gZCCL: Compression-Accelerated Collective Communication Framework for GPU Clusters

Jiajun Huang
jhuan380@ucr.edu
University of California, Riverside
Riverside, United States of America

Sheng Di
sdi1@anl.gov
Argonne National Laboratory
Lemont, United States of America

Xiaodong Yu
xyu38@stevens.edu
Stevens Institute of Technology
Hoboken, United States of America

Yujia Zhai
yzhai015@ucr.edu
University of California, Riverside
Riverside, United States of America

Jinyang Liu
jliu447@ucr.edu
University of California, Riverside
Riverside, United States of America

Yafan Huang
yafan-huang@uiowa.edu
University of Iowa
Iowa City, United States of America

Ken Raffenetti
raffenet@anl.gov
Argonne National Laboratory
Lemont, United States of America

Hui Zhou
zhouh@anl.gov
Argonne National Laboratory
Lemont, United States of America

Kai Zhao
kzhao@cs.fsu.edu
Florida State University
Tallahassee, United States of America

Xiaoyi Lu
xiaoyi.lu@ucmerced.edu
University of California, Merced
Merced, United States of America

Zizhong Chen
chen@cs.ucr.edu
University of California, Riverside
Riverside, United States of America

Franck Cappello
cappello@mcs.anl.gov
Argonne National Laboratory
Lemont, United States of America

Yanfei Guo
yguo@anl.gov
Argonne National Laboratory
Lemont, United States of America

Rajeev Thakur
thakur@anl.gov
Argonne National Laboratory
Lemont, United States of America

ABSTRACT

GPU-aware collective communication has become a major bottleneck for modern computing platforms as GPU computing power rapidly rises. A traditional approach is to directly integrate lossy compression into GPU-aware collectives, which can lead to serious performance issues such as underutilized GPU devices and uncontrolled data distortion. In order to address these issues, in this paper, we propose *gZCCL*, a *first-ever* general framework that designs and optimizes GPU-aware, compression-enabled collectives with an accuracy-aware design to control error propagation. To validate our framework, we evaluate the performance on up to 512 NVIDIA A100 GPUs with real-world applications and datasets. Experimental results demonstrate that our *gZCCL*-accelerated collectives, including both collective computation (Allreduce) and collective data movement (Scatter), can outperform NCCL as well as Cray MPI by up to 4.5× and 28.7×, respectively. Furthermore, our accuracy evaluation with an image-stacking application confirms the high reconstructed data quality of our accuracy-aware framework.

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656636

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; **Parallel algorithms**; • **General and reference** → **Performance**; • **Computer systems organization** → **Distributed architectures**.

KEYWORDS

GPU, Collective Communication, Compression

ACM Reference Format:
Jiajun Huang, Sheng Di, Xiaodong Yu, Yujia Zhai, Jinyang Liu, Yafan Huang, Ken Raffenetti, Hui Zhou, Kai Zhao, Xiaoyi Lu, Zizhong Chen, Franck Cappello, Yanfei Guo, and Rajeev Thakur. 2024. gZCCL: Compression-Accelerated Collective Communication Framework for GPU Clusters. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656636

1 INTRODUCTION

In the exascale computing era, efficient large-message collective communications are crucial for the performance of modern GPU-based supercomputers and clusters. This is particularly true for scientific applications and deep learning tasks that involve extensive

data processing and exchange [1, 2, 4, 5, 15]. For example, the classic LSTM [11] model used in the language modeling task can contain more than 66 million parameters and the communication overhead can be as high as 94% [2], increasing the need for optimizing GPU-aware collective communication for large messages [6, 7].

For GPU-aware collective communication, numerous researchers are actively working on mitigating network congestion in large-message collectives. Network saturation is often the major bottleneck because of limited network bandwidth. For example, even with advanced networks, such as HPE Slingshot 10, the network bandwidth is only about 100 Gbps [22]. A straightforward solution is designing large-message collective communication algorithms that can minimize the transferred data volume instead of latency [3, 20, 26]. Another promising solution is shrinking the message size by error-bounded lossy compression techniques [9, 13, 18, 19, 25, 27], as it can significantly reduce the data volume and maintain the data quality.

Previous lossy-compression-integrated approaches can be divided into two categories. The first is *compression-enabled point-to-point communication* (namely CPRP2P) [30], which directly uses the 1D fixed-rate ZFP [18] to compress the data before it is sent and decompresses the received data after it is received. This method may cause significant overheads and unbounded errors in the collective communications as shown in [12, 31]. The other category is to particularly optimize the *compression-enabled collectives*. Zhou et al. [31] integrated the 1D fixed-rate ZFP [18] into MPI_Alltoall on GPUs; however, this approach is limited to the Alltoall operation and CPU-centric staging algorithm and also results in the issue of unbounded error. Huang et al. [12] designed an optimized general framework for compression-enabled collectives that can realize high performance for all MPI collectives with controlled errors. Nevertheless, this approach suffers from suboptimal performance on modern GPU clusters because of under-utilized GPU devices.

Designing a GPU-aware compression-enabled collective communication system that realizes both high performance and controlled error propagation is non-trivial. There are three key challenges to address.

(1) How can we co-design and implement a compression-enabled collective algorithm that optimizes performance within modern GPU clusters? For Allreduce operations, for example, state-of-the-art GPU-aware collective communication libraries, such as NCCL [8] and MPICH [17], adopt ring-based algorithms to optimize the transmission of large messages. However, it is unclear whether the ring-based model is the best fit when we include lossy compression techniques. In fact, unlike CPU, the GPU-based compression may easily face a low utilization issue, because of the inevitable GPU kernel-launch overhead and limited parallel design in GPU-based compression algorithms, which significantly lowers the performance.

(2) How can we optimize the redesigned algorithms to increase GPU utilization and decrease the required synchronizations and data transfers? This is because unnecessary data transfers and synchronization can considerably increase the overall runtime and eliminate the opportunity for overlapping in the coordination of the host and device.

(3) How can we devise an accuracy-aware co-design that maintains data quality without sacrificing performance? The accuracy of collective operations is at risk due to the data loss from GPU lossy compression. It is important to balance performance with accuracy.

To address the challenges mentioned above, this paper introduces a *first-ever* generic high-performance framework, namely *gZCCL*, specifically designed for GPU-aware compression-accelerated collective communications. Our contributions can be summarized in four key aspects:

- To tackle challenge (1), we present two innovative algorithm design frameworks for classic collective operations, encompassing both collective computation and collective data movement. This proposal stems from a thorough analysis of the limitations in traditional large-message algorithms. This is fundamental to various co-designed compression-enabled collective algorithms, which can increase device utilization, decrease times of compression/decompression, and maximize performance.
- To address challenge (2), we develop a series of optimization strategies to improve performance. Specifically, we improve the error-bounded lossy compressor (cuSZp [14]) and develop a multi-stream version to suit the context of the two collective performance optimization frameworks. For the data movement framework, we overlap the compression/decompression, kernel launching, and data movement, respectively. For the collective computation framework, we enable possible overlapping between compression, decompression, and communication, which can further reduce the collective runtime.
- To address challenge (3), we design various strategies to considerably control the error accumulation in the *gZCCL* framework. We carefully design the *gZCCL* framework with the error-bounded lossy compressor that always causes an unknown compressed data size instead of the fixed-rate compressor that leads to a pre-known output data size to ensure a bounded error. We also decrease the number of compression operations on purpose, which can effectively decrease the number of stacked errors during the communication pattern.
- We integrate *gZCCL* framework into numerous collective operations, including Allgather, Reduce_scatter, Allreduce, and Scatter, and meticulously evaluate their performance using different real-world scientific datasets. Experiments with up to 512 NVIDIA A100 GPUs reveal that other related works suffer from undesirable performance degradation in both Allreduce and Scatter due to significant compression overhead, inefficient GPU utilization, or larger data transfer volume. In contrast, our *gZCCL*-based Allreduce (referred to as gZ-Allreduce) outperforms the Allreduce in Cray MPI and NCCL by 20.2× and 4.5×, respectively. Our *gZCCL*-based Scatter (gZ-Scatter) operates 28.7× faster than the MPI_Scatter in Cray MPI. We also utilize a real-world use case (i.e., image stacking analysis) to validate the practical effectiveness of gZ-Allreduce. It demonstrates a 1.69× performance gain over NCCL, while still preserving a high level of data integrity.

The rest of the paper is organized as follows: we introduce background and related work in Section 2 and detail our design and

optimization in Section 3. Evaluation results are presented in Section 4 followed by conclusion and future work in Section 5.

2 BACKGROUND AND RELATED WORK

Researchers have long been interested in utilizing compression to enhance MPI communication performance, based on the two communication categories – point-to-point communication and collective communication.

For the first category, a typical latest related work is utilizing 1D fixed-rate ZFP to boost MPI communications on GPU clusters [30]. Their approach, however, focuses on enhancing MPI point-to-point communication performance, yielding suboptimal performance in collective scenarios. Furthermore, their solution could not provide a bounded error due to its fixed-rate design that fixes the compressed data size rather than ensuring accuracy. In contrast, our collective framework integrates error-bounded lossy compression, guaranteeing both high-quality compression and high collective performance. Hence, we regard this work as orthogonal to ours.

As for the second category, several existing studies explored how to optimize the MPI collective performance particularly, while they are limited to either CPU-centric communication (i.e., all the data are transferred through the CPU essentially) and/or have the uncontrolled error propagation. Zhou et al. proposed several optimized MPI collective operations [28, 29, 31] using fixed-rate compression, which leads to inferior compression quality and unbounded error aggregation. On the contrary, our general framework provides a detailed guideline for designing and optimizing compression-accelerated collective algorithms, maximizing the performance of both collective computation and collective data movement while featuring well-controlled data distortion. Hence, we categorize these works as orthogonal works to ours. In addition, Huang et al. proposed an error-controlled compression-enabled framework that is capable of achieving a high performance across all MPI collectives [12]. Their method, however, fails to solve the inefficient GPU utilization, synchronization, and device-host data transfer issues, resulting in suboptimal performance on GPU clusters. In contrast, our GPU-centric framework is capable of fully utilizing the computational power of GPUs, significantly lowering the amounts of required compression, synchronization, and device-host data transfer, leading to a remarkable performance improvement.

In the following text, we mainly focus on optimizing the performance of collective communications on GPU clusters by error-bounded lossy compression. This is because prior research [12] already demonstrated that the error-bounded lossy compression brings a limited and controllable impact on the final accuracy of collective communications by both theoretical and experimental analysis.

3 *GZCCL* DESIGN AND OPTIMIZATION

In this section, we present our design and optimization strategies. Figure 1 shows the design architecture of *gZCCL*, where the newly designed modules are highlighted in purple boxes. We develop an adapter that can run cuSZp [14] more efficiently in regard to collective communications, to be detailed in Section 3.3.2. We discuss our algorithm design as well as a series of performance optimization strategies, which are meticulously crafted for the two classic types of collectives – collective computation and collective data movement. Details are described in Sections 3.3.3 and 3.3.4.

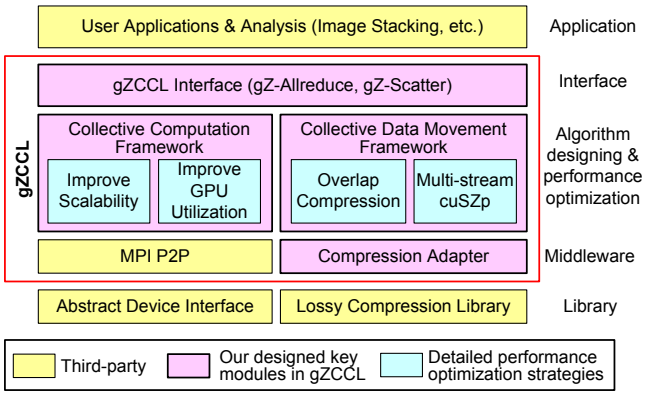


Figure 1: *gZCCL* design architecture.

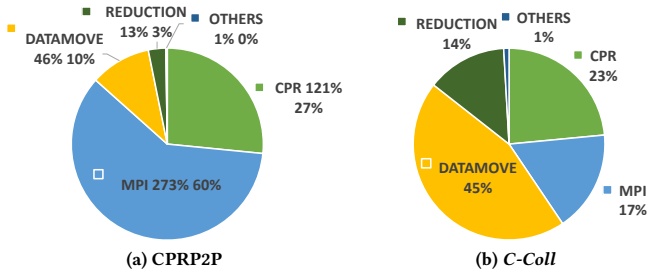
3.1 Analysis of existing compression-enabled GPU-aware collectives

In this section, we analyze the problems of prior solutions and provide a comprehensive performance breakdown to identify potential bottlenecks.

3.1.1 ***Inefficient prior solutions in GPU-aware collectives.*** Lossy compression-enabled point-to-point communication (CPRP2P) can decrease the transferred data volume [30], however, it faces huge accuracy loss and performance degradation in the collective scenario [31]. To solve these issues, *C-Coll* framework was proposed with two sub-frameworks: data movement framework and collective computation framework [12]. In the data movement framework, the data is pre-compressed and then sent along the communication patterns. Through this method, the huge compression overhead brought by the CPRP2P could be avoided. In the collective computation framework, the compression and communication costs are overlapped with each other, resulting in a better overall runtime. However, the direct implementation of the *C-Coll* framework may experience a huge performance degradation on modern GPU clusters due to two facts: 1. The current MPI collectives result in sub-optimal performance because all the temporary buffers are allocated on CPU, which means the data needs to be moved from GPU to CPU for the data to be transmitted over networks. Even though integrated compression can reduce the transferred message size, the device-host data movement cost can be significant. 2. The *C-Coll* framework does not address the inefficient GPU utilization problem and host-device synchronization issue, which may substantially degrade the collective performance.

3.1.2 ***Identification of the bottlenecks in prior related works.*** The ring-based Allreduce is a method commonly used in numerous state-of-the-art GPU-aware collective communication libraries such as NCCL [8] and MPICH [17], particularly when optimizing large-message communications. This technique is composed of both data movement collective (Allgather) and collective computation (Reduce_scatter), both of which have been optimized in

C-Coll [12]. Figure 2 presents a performance breakdown for the CPRP2P and *C-Coll* within the GPU-aware ring-based Allreduce algorithm. The evaluation is conducted utilizing 64 NVIDIA A100 GPUs, with 4 GPUs per node. When comparing CPRP2P versus *C-Coll*, it is evident that the latter significantly decreases the time cost in compression and decompression (CPR), resulting in overall enhanced performance. However, it is notable that in *C-Coll*, the time required for host-device data transfer (DATAMOVE) is significant, accounting for nearly 45% of the total runtime. In addition, the time consumed by compression and decompression (CPR) still remains substantial, occupying more than 23% of the total time. This can be attributed to the inefficient utilization of GPUs. To rectify these problems, we present the *gZCCL* framework, whose design and implementation are detailed in subsequent sections.



(a) CPRP2P (b) *C-Coll*

Figure 2: Performance breakdown of Allreduce using CPRP2P and *C-Coll*: CPRP2P's first percentage is scaled to *C-Coll*'s runtime, and the second is scaled to its own.

3.2 Characterization of ring-based compression-enabled GPU-aware collectives

3.2.1 ***Traditional ring-based algorithms for long messages.*** Ring algorithms are widely acknowledged as the state-of-the-art solution for large-message collective communications such as Allgather, Reduce_scatter, and Allreduce. In scenarios involving pure collective communications, ring approaches can significantly control the total data transfer volume, which can effectively control the network congestion when message sizes are large, thereby delivering optimal performance. When CPU compression is employed, the CPU can be fully utilized for large message sizes, and the communication data volume can be substantially reduced, leading to a vast increase in overall collective performance. Prior research has shown that compression cost can be a dominant bottleneck in compression-enabled collectives. The reduction in communication volume in the ring-based algorithm design can lower the workload on the compressor, resulting in optimal performance. Hence, ring-based approaches are considered the most suitable algorithms for collectives integrated with CPU compression.

Taking into account modern GPUs [23] features very high performance because of its performant single instruction, multiple threads (SIMT) architecture, adopting GPU-based lossy compression may further reduce the compression overhead intuitively, however, a key question arises: Can GPUs still be fully utilized in the compression-enabled ring-based algorithm? In fact, unlike CPUs which are often saturated, GPU performance is heavily dependent on the utilization rate. That is, a low utilization rate on GPU will increase the compression cost and lead to sub-optimal collective performance. To answer the above question, we need to characterize the performance of the lossy compressor.

3.2.2 ***Characterization of GPU lossy compressor.*** In this section, we detail the characterization of the GPU lossy compressor – cuSZp [14], and this process is also applicable to other GPU compressors. Utilizing 646MB (the data size of the largest scientific dataset we use later) of synthetic data where all data points are uniformly distributed, we characterize the performance of cuSZp on an NVIDIA A100 GPU as shown in Figure 3. We observe that as data size decreases, execution time decreases for both compression and decompression kernels with a declining rate, and even stagnates when the data size is smaller than 5MB. This indicates that the GPU is not fully utilized, especially when the input data size is relatively small, and the utilization rate continues to drop with a decrease in message size. However, an input message larger than 1MB is already considered a large message in collective communications, and the actual message to be sent/received or compressed during ring-like communication patterns is much smaller than the input message. This is because the original data is divided into small blocks for communications. Consequently, ring-based algorithms may result in relatively low GPU utilization, and we provide a more detailed discussion in the following text.

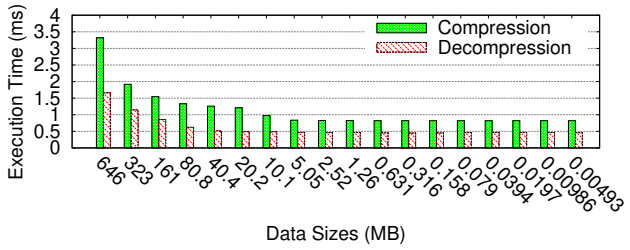


Figure 3: Characterization of cuSZp compression and decompression execution time with uniform data.

3.2.3 ***Ring-based collective computation.*** In this section, we explore the limitations of the ring-based algorithms integrated with GPU compression, using the ring-based Reduce_scatter operation as an illustrative example. In the ring-based Reduce_scatter operation, the input data, denoted by size D, is divided into N small chunks, with N being the process count. Each of these chunks undergoes a ring-like communication pattern for reduction across $N-1$ rounds. When the GPU compression is incorporated, each round provides a data chunk of size D/N to the compression kernel, while an equal-sized output is produced by the decompression kernel. This mechanism necessitates a total of $N-1$ rounds of both compression and decompression. Consequently, even when dealing with large message sizes like 646MB, the GPU experiences significantly poor utilization when the process count reaches approximately 128 ($646/5.05 \approx 128$), according to our previous analysis in Section 3.2.2. This results in compromised scalability. Further exacerbating this issue is the fact that the total number of decompression

and compression operations is $N-1$, which scales linearly with the process count N. Notably, this problem is not exclusive to the ring-based Reduce_scatter operation. The widely-used ring-based Allreduce operation, which is composed of ring-based Allgather and Reduce_scatter, is also plagued by these scalability and performance shortcomings. Therefore, the direct application of ring-based algorithms for collective computation with GPU compression may not always yield optimal results. It is hence vital to explore other algorithms that may offer superior performance.

3.3 Proposing the novel *gZCCL* framework

In this section, we delve into the details of our *gZCCL* framework. Our primary goal is to address and overcome the performance issues noted in the previous GPU-aware MPI collective framework that incorporates compression, such that a superior performance can be reached.

3.3.1 ***Getting rid of the traditional host-centric design.*** To circumvent the high cost of device-to-host data transfer inherent in traditional CPU-centric designs, we implement a GPU-centric design. Specifically, when GPU support is enabled, a sufficiently large GPU buffer pool is pre-allocated during the `MPI_Init` function call. The size of this GPU buffer pool can be adjusted based on user input. Hence, GPU-aware MPI collectives can leverage these pre-allocated device buffers directly during function calls, rather than repeatedly allocating them amidst intensive communications. This is not only resource-intensive but also causes undesired host-device synchronization. Moreover, current MPI implementations tend to use the host for carrying out reduction operations in collective computations. In response to this, we designed and implemented a GPU reduction kernel capable of processing data entirely on the device. With these optimizations, we successfully transition from the original host-centric algorithms and elevate the compression-enabled collectives to the device-centric level.

3.3.2 ***Adapting lossy compression to achieve high collective performance.*** To improve collective performance in compression-enabled collectives, it is critical to adapt the lossy compression to suit the requirements of collective communications. We illustrate our customization and optimization strategies based on cuSZp, and the improvement strategies can also be applied to other lossy compressors.

In the following, we analyze the potential performance issue of cuSZp, and then describe our improvement strategies. In the cuSZp function `cuSZp_compress_deviceptr`, an initial step involves the allocation of a unified memory buffer known as `d_cmpOffset`, accessible from both the device and host. This joint accessibility incurs implicit host-device data transfer, leading to suboptimal performance. To counteract this issue, we redesign cuSZp's data allocation process, liberating cuSZp from the constraints of unified memory. This modification results in a reduction of necessary data transfers, subsequently improving performance. Moreover, cuSZp allocates temporary buffers to store compression-related parameters upon any invocation of the `cuSZp_compress_deviceptr` function. This procedure may block the host and also generates unwanted device overheads in collective scenarios where compression is frequently executed. To address this issue, our solution allocates a temporary buffer, which will be cleared and reused for any compression operations, so that the memory allocation costs can be reduced significantly also with data integrity.

3.3.3 ***Two algorithm design frameworks.*** In this section, we describe the algorithm design inherent to our *gZCCL*.

Exploring new metrics regarding GPU compression-enabled collective performance. As for the GPU compression-enable collective algorithms, there are several important new metrics that need to be addressed in particular.

Total compression cost. The compression cost is determined by two critical factors: per-compression time cost and the number of compression executions. As for the per-compression cost on GPU, it may face a low utilization issue when the input data is not large enough, as discussed in Section 3.2.2. For example, 10 times of compression of 1 MB data can be much more expensive than 1 compression of 100 MB data as shown in Figure 3. As such, we should pay much attention to the number of times the data need to be compressed, in order to minimize the total compression cost. As verified in Section 3.2.3, we demonstrate that large-message algorithms such as ring-based algorithms may result in low scalability with compression in some cases, which is due to the fact that they can result in more compression operations each with low GPU utilization. In the compression-enabled collectives, how often the compression is executed is closely related to the times of the data communications, which are generally optimized by the small-message algorithms. Thus, the conclusion is that, with GPU compression integrated, the small-message algorithms may outperform the large-message algorithms.

Accuracy loss. Apart from the compression-related overheads, another concern of integrating lossy compression in the collectives is the accuracy loss caused by accumulated errors along with the intensive communications. Again, the large-message algorithms like the ring-based approach can introduce larger errors compared with the small-message algorithms such as the one based on the recursive-doubling algorithm, further degrading the reconstructed data quality. This is due to the fact that the ring-based algorithm requires $N-1$ times of compression/decompression and the recursive-doubling-based algorithm only needs $logN$ compression/decompression operations. Fortunately, the increased times of compression/decompression may not bring a huge accuracy difference statistically because the mathematical expectation of all accumulated errors is 0. Thus, we can achieve a high reconstructed data quality with the integration of lossy compression in the collective communications, which will be demonstrated later in Section 4.5.

Collective computation algorithm design framework. In the following discussion, we will employ the typical Allreduce operation as a case study to describe the algorithm design of our *gZCCL* framework in collective computation scenarios. In general, the recursive doubling algorithm is employed for short messages due to its optimized latency, whereas the previously-mentioned ring-based algorithm is used for large messages in Allreduce because of its ability to control the data transfer volume [26]. The ring-based Allreduce operation consists of a Reduce_scatter stage and an Allgather stage.

In the Reduce_scatter stage, $N-1$ compression/decompression operations are required, while the Allgather stage necessitates one compression and $N-1$ decompression operations [12]. When compared with the N compression operations and $N-1$ decompression operations required by the ring-based Allreduce algorithm, the recursive doubling algorithm involves only $log\ N$ communication steps or compression/decompression operations, where N is the process count. As such, the recursive doubling algorithm exhibits superior scalability in terms of compression cost, especially when $D/N < 5MB$, where D denotes the input data size. However, when compressing the data with the data size being D/N and the GPU utilization is high, the ring-based algorithm still outperforms the recursive doubling one as it can minimize both compression and communication workloads. In conclusion, the recursive doubling-based Allreduce algorithm delivers high scalability, while the ring-based one projects a high performance when GPU utilization is high.

Collective data movement algorithm design framework. In this section, we delve into the algorithm design of our *gZCCL* framework in collective data movement scenarios. Generally, there are three types of collective data movement: one-to-all, all-to-one, and all-to-all. The all-to-all communication pattern is the most complex as it encapsulates both one-to-all and all-to-one communications. Accordingly, we select the extensively-used all-to-all communication operation – Allgather – as a case study to demonstrate the algorithm selection process in *gZCCL*. Note that this design can also be applied to other collectives. In essence, the Bruck algorithm and the recursive doubling algorithm are optimized toward lowering latency, while the ring-based algorithm prioritizes minimizing data transfer volume. Unlike collective computation scenarios, data compression only happens at the beginning and the end of the collective data movement. For instance, the data in the Allgather operation of each process should be compressed first, then the compressed data is communicated between processes. After all communications are completed, each process decompresses the gathered compressed data to retrieve the original data.

In what follows, we extensively analyze which compression-enabled algorithm is the best fit for the Allgather operation. Although the ring-based Allgather requires $N-1$ communication steps to finish, it only necessitates one compression and $N-1$ decompression operations. In addition, the $N-1$ decompression operations can be overlapped using multi-stream techniques to improve GPU utilization, which will be detailed in Section 3.3.4. In conclusion, the ring-based Allgather only suffers from inefficient GPU utilization in one compression operation, and it benefits from optimized data transfer volume. Therefore, although the Bruck and recursive doubling algorithms exhibit the least communication steps or compression operations, they cannot further improve scalability and suffer from sub-optimal data transfer volume compared to the ring-based algorithm. As a result, the ring-based approach emerges as the optimal choice for the compression-integrated Allgather operation.

3.3.4 ***Two performance optimization frameworks.*** In this section, we give a comprehensive discussion of the intricate optimization techniques that are integral to our *gZCCL* framework. By unveiling the technical underpinnings of our framework, we aim to provide an in-depth understanding of how our methods contribute to improved performance and efficiency in GPU-based computational systems.

Developing multi-stream lossy compression. To facilitate multi-stream compression and decompression within collective communication, we need to tailor the lossy compressor, which originally operates using a single default GPU stream. For illustrative purposes, we mainly describe the compression procedure based on the state-of-the-art GPU-based compressor – cuSZp as an example. We begin by delving into the source code to modify the cuSZp compression process, enabling it to accept a user-defined stream rather than operating exclusively on the default stream. This new stream-supported compression API is henceforth referred to as `cuSZp_compress_stream`. To effectively overlap compression across different streams, it is imperative to ensure the absence of data races and undesired conflicts. Accordingly, we conduct a meticulous analysis and testing of the critical paths and data dependencies within cuSZp. During this investigation, we find that beyond the standard `d_oriData` (buffer of original data) and `d_cmpBytes` (buffer of compressed data), cuSZp requires several distinct device buffers to store temporary information, including offsets of various compression blocks and flag bits. Consequently, we independently allocate buffers for each stream to avoid data conflicts in the multi-stream scenario. The decompression as well as other lossy compressors can be multi-streamed similarly, and we omit details due to space limit.

Collective computation performance optimization framework. In this section, we illustrate the *gZCCL* optimizations in the collective computation routines using the recursive doubling-based Allreduce as an example. Similar optimizations can be applied to other collective computation algorithms such as Reduce_scatter. Figure 4 illustrates the *gZCCL* implementation on the recursive doubling-based compression-enabled Allreduce operation (we call it gZ-Allreduce (ReDoub)). We first create one non-default stream and a set of temporary device buffers then reuse these GPU buffers for all the compression and decompression to avoid extra overheads. Then, the design contains two main stages, which will be described in the following text, where N is the number of processes, and r refers to the remainder of the process count taking away the maximum power of two: i.e., $r = \min(N - 2^k)$, where $k \in \mathbb{Z}_+$ and $k \leq \log_2 N$.

In the first stage, we mainly handle the remainder processes (r processes). In the case where the number of processes is not a power of two, all even-numbered processes with a rank (denoted i) lower than $2r$ first asynchronously clear the temporary GPU buffers and launch the compression kernel on the non-default stream to compress their whole data and sending their compressed data to the process of rank i+1. Meanwhile, the odd-numbered processes pose non-blocking receive operations to obtain the compressed data and clear the GPU buffers for decompressing them on the non-default stream. Then, these even-numbered processes are suspended until the final stages, and the odd-numbered processes half their ranks (i=$i/2$).

In the second stage, we handle the remaining power of two processes (i.e., 2^k). For the processes with ranks $i \geq 2r$, we update the ranks by i=$i-r$. Then, in each recursive doubling communication

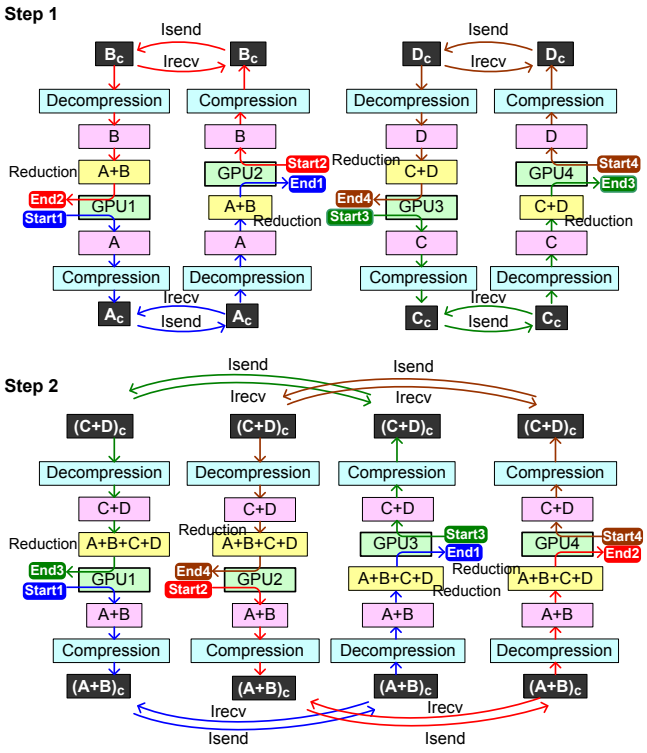


Figure 4: Design of our *gZCCL* collective computation framework on compression-accelerated gZ-Allreduce. This example uses four GPUs/Processes.

step, each process asynchronously memsets the temporary device buffers and launches the compression kernel on the non-default stream to compress the data. The compressed data is sent through a non-blocking send operation and another non-blocking receive operation is posed to receive the compressed data from another process. Upon the receiving of data, a clear operation and decompression kernel are launched to obtain the original data. Thereafter, the reduction kernel is launched on the non-default stream to reduce the decompressed data and data in the receive buffer. Unlike the ring-based case, each communication step requires sending/receiving the whole data instead of the divided data blocks, ensuring high GPU utilization.

Collective data movement performance optimization framework. In this section, we describe how we optimize collective data movements to enhance GPU utilization. We use the binomial tree-based *gZCCL*-accelerated Scatter/Scatterv as an example. Similar optimization can be applied to other collective data movement operations, such as Allgather.

We design our gZ-Scatter based on the binomial tree-based Scatter algorithm that is utilized in both short and long messages[26]. In Figure 5, we present the overall design of our gZ-Scatter. In this algorithm, the original data on the root process is distributed to other processes in a binomial tree communication pattern. An intuitive solution is compressing the original data as a whole and sending the compressed data by blocks to other processes, which however introduces two challenges. On the one hand, the compressed bytes contain some metadata that are essential for decompression. If the compressed data are directly divided into smaller blocks, the vital information will be lost. On the other hand, the original data distribution might not be uniform and the compressed data sizes for each block are not equal. As a result, it is impossible for us to correctly separate the compressed data into data blocks in this case. Thus, we need to individually compress the corresponding data blocks and then distribute them.

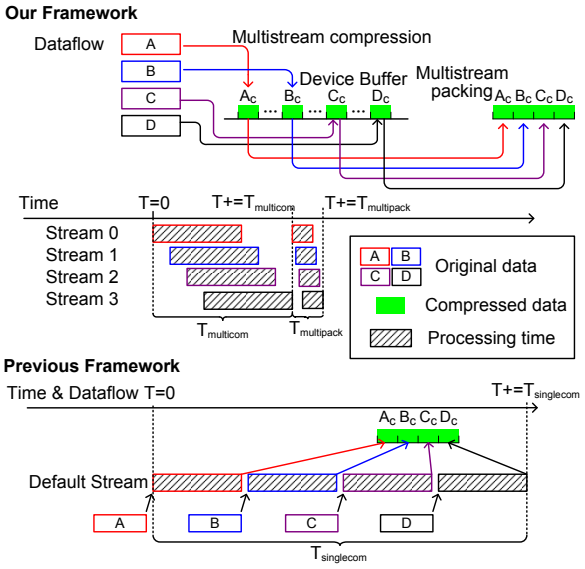


Figure 5: Design of our *gZCCL* data movement framework on compression-accelerated gZ-Scatter. This example uses four GPUs/Processes.

To better explain our optimization and design, more details are shown as follows. First, we create helper arrays on the CPU to store the compressed data sizes and the related global offsets on each process. Then, in each process, we create a stream array of size N, where N is the size of the communicator. Additionally, we allocate two device buffer pointer arrays, also of size N, to store the offsets of compressed bytes and flag information for differing streams, respectively. In the root process, we launch the multi-stream compression kernel utilizing the independent device buffers and streams from 0 to $N-1$ in the stream array. The compressed data for each stream is stored in the same device buffer based on the designated offset so that there are no data races. Then, we synchronize these streams with the host to make sure the multi-stream compression has finished. After that, we obtain the compressed data sizes and offsets of different streams and synchronize the information with other non-root processes. Then, we use asynchronous memcpys with different streams to pack these compressed data based on the compressed data offsets into another device buffer, so that they can be sent out in a continuous format. Finally, the data is distributed in a binomial tree communication pattern and the non-root processes utilize a non-default stream to decompress its own part of compressed data. In a nutshell, we have optimized the compression-enabled Scatter

algorithm with overlapped compression, kernel launching, and data movements, resulting in improved performance.

4 EXPERIMENTAL EVALUATION

We present and discuss the evaluation results as follows.

4.1 Experimental Setup

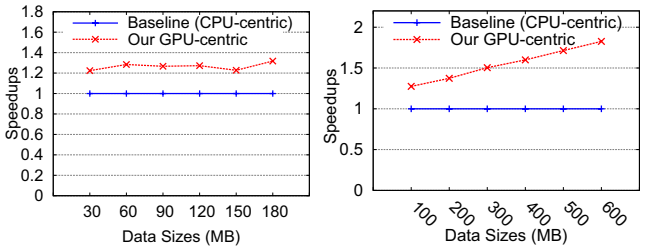
We perform the evaluation on a GPU supercomputer that involves 512 NVIDIA A100 80G GPUs (128 nodes each with 4 GPUs, specifically), which features both internode communication and intranode communication. These computational nodes are interconnected via the HPE Slingshot 10 interconnect, providing a network bandwidth of 100 Gbps. Unless specified, the absolute error bound of compression is set to 1E-4, because the image reconstruction quality is already superior with 2E-4 error bound, which will be demonstrated later in Figure 13. Two distinct RTM datasets [16], originating from the real-world 3D SEG/EAGE Overthrust model, are generated under two different simulation settings. Table 1 exhibits the average compression ratio and PSNR that cuSZp can achieve for these datasets, where ABS denotes the absolute error bound.

Table 1: Compression ratio (CPR) and quality (PSNR).

	Simulation Setting 1		Simulation Setting 2	
Dimensions	**449X449X235**		**849X849X235**	
ABS	**CPR**	**PSNR**	**CPR**	**PSNR**
1E-3	92.28	53.23	94.41	53.41
1E-4	73.35	65.67	63.94	70.38
1E-5	55.65	78.83	46.74	88.57

4.2 Evaluating the GPU-centric design

First of all, we present the performance evaluation of our proposed GPU-centric design compared with the traditional CPU-centric solution on 64 NVIDIA A100 GPUs across 16 nodes, using two different scientific datasets and the Allreduce collective operation. As mentioned in Section 1 and Section 2, many of the existing related works [12, 31] are dependent on the CPU-centric communication design. As shown in Figure 6b, it is noticeable that the speedups of GPU-centric design over the CPU-centric solution increase with the expansion of the data sizes, culminating in a 1.82× performance improvement for the data size of 600 MB. This trend is also observed in Figure 6a, where the speedup can reach up to 1.32× with the largest 180 MB data size. As data size increases, the demand of intensive host-device data movement escalates in the CPU-centric design, which may cause an increasing PCIe congestion and reduction cost. This creates a pronounced bottleneck for the overall collective performance. To mitigate the substantial cost of host-device data transfer, our GPU-centric design does not depend on CPU-based communication, totally eliminating the data movement cost between CPU and GPU. Moreover, our design can significantly mitigate reduction operation cost, further boosting the speedup, especially with the growth of data size.



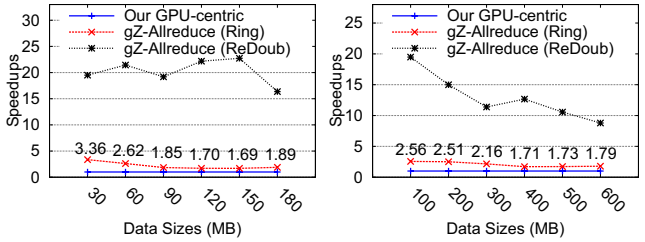
(a) Simulation Setting 1 (b) Simulation Setting 2

Figure 6: Performance evaluation of our GPU-centric design using two different scientific datasets.

4.3 Evaluating the optimized redesigned GPU compression-enabled collective algorithms

We evaluate the performance of our optimized, compression-integrated collective algorithms using 64 NVIDIA A100 GPUs.

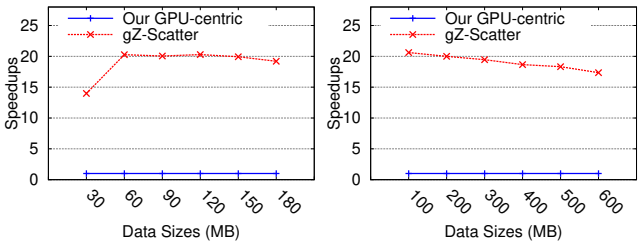
4.3.1 Collective computation. In this section, we evaluate our optimized redesigned compression-enabled collective computation algorithms using the widely-used Allreduce operation. Both Figure 7a and 7b reveal that our optimized solution – gZ-Allreduce (Ring) surpasses our original GPU-centric approach by up to 3.36×. This is because our solution improves GPU utilization. Specifically, we overlap the decompression and kernel launching in the Allgather stage and facilitate potential overlapping among compression, decompression, and communication in the Reduce_scatter stage. Furthermore, the newly designed gZ-Allreduce (ReDoub) achieves an even higher performance enhancement compared to gZ-Allreduce (Ring), attaining up to 22.7× speedup compared to our original GPU-centric approach. We explain the reasons as follows. To tackle the inefficient device utilization in ring-based Allreduce, we design and optimize a novel recursive doubling-based compression-enabled algorithm, with the aim of improving scalability, maximizing performance, and preserving accuracy. However, it is worth noting that the speedup of both gZ-Allreduce methods generally decreases as the data size increases. This is because the problem of inefficient GPU utilization can be mitigated by larger message sizes, and the performance improvement resulting from higher GPU utilization would consequently decrease.



(a) Simulation Setting 1 (b) Simulation Setting 2

Figure 7: Performance evaluation of our *gZCCL* collective computation framework using Allreduce operation.

4.3.2 Collective data movement. In this section, the performance of our optimized, redesigned compression-integrated collective data movement algorithms is demonstrated, using the classic Scatter operation. From Figure 8a and Figure 8b, we notice that gZ-Scatter exhibits substantial speedups in both datasets, obtaining up to 20.3× and 20.6× improved performance in the two simulation settings, respectively. This is because, in the gZ-Scatter algorithm, we overlap compression, kernel launching, and data movements, leading to enhanced device utilization, diminished host-device synchronization, and reduced device-device data movement cost. Similar to the collective computation scenario, with increasing data sizes, the performance boost slightly diminishes, with a minimum of 17.4× at 600 MB as depicted in Figure 8b. This reason is that a larger input data size can better saturate the device, thereby mitigating the performance enhancement derived from our *gZCCL* design.



(a) Simulation Setting 1 **(b) Simulation Setting 2**

Figure 8: Performance evaluation of our *gZCCL* collective data movement framework using Scatter operation.

4.4 Comparisons of *gZCCL* with other collective communication libraries

In this section, we compare the performance of our *gZCCL* framework with other state-of-the-art GPU communication libraries, such as the widely-utilized NCCL and CUDA-aware Cray MPI.

4.4.1 Collective computation. In this section, the performance of our *gZCCL* collective computation framework is compared with both NCCL and Cray MPI, using the prevalent Allreduce operation.

Evaluation with different message sizes. We evaluate the performance of our gZ-Allreduce algorithm using various data sizes up to 600 MB on a configuration of 64 NVIDIA A100 GPUs across 16 nodes. As observed in Figure 9, our recursive doubling-based gZ-Allreduce (ReDoub) consistently outperforms across all data sizes, achieving up to a speedup of 18.7× compared to Cray MPI and a 3.4× performance improvement over NCCL. Furthermore, with increasing data sizes, the speedup generally rises, demonstrating high scalability with respect to data size. The performance improvement originates from the significantly reduced message size and compression-related overheads in our *gZCCL* design, which can further mitigate network congestion with enlarging message sizes. However, the ring-based gZ-Allreduce (Ring), despite surpassing Cray MPI for the data size with 50+ MB, fails to outpace NCCL. This is attributed to the inefficient GPU utilization in gZ-Allreduce (Ring), which incurs substantial compression-related costs, outweighing the benefits of reduced message size.

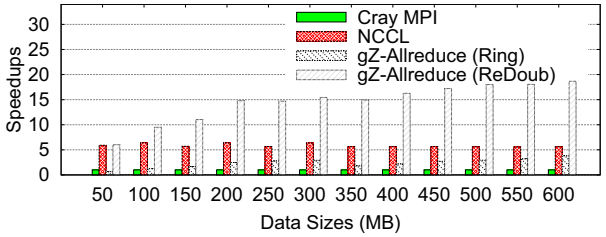


Figure 9: Performance evaluation of our gZ-Allreduce with Cray MPI and NCCL in different data sizes.

Evaluation with different GPU counts. In this section, we assess the scalability of our gZ-Allreduce algorithm with the complete RTM dataset of 646 MB data size, utilizing up to 512 NVIDIA A100 GPUs across 128 nodes. We start from 8 GPUs, as it is the minimal amount to have both internode and intranode communication with 4 GPUs per node.

As depicted in Figure 10, our recursive doubling-based gZ-Allred uce (ReDoub) consistently performs the best, achieving up to 20.2× and 4.5× speedups compared to Cray MPI and NCCL respectively, across varying GPU counts. This superior performance stems from the substantial reduction in message size with relatively low compression cost achieved by our *gZCCL* framework. When the GPU count is at 8, Cray MPI appears to suffer from significant performance degradation, as compared to the other three counterparts. Apart from the 8-GPU case, as the number of GPUs increases, both gZ-Allreduce (ReDoub) and NCCL tend to exhibit a greater performance boost compared to Cray MPI, indicating robust scalability with respect to the GPU count. This is because both gZ-Allreduce (ReDoub) and NCCL are optimized for large GPU count scenarios. However, the trend differs for the ring-based gZ-Allreduce (Ring), which outperforms NCCL when the GPU count is 32 or less. As the GPU count increases, its performance deteriorates, ending up with the worst performance compared with other solutions in the case of 512 GPUs. The declining performance is attributed to the reduced input data size for each compression/decompression with an increase of GPU count, leading to lower device utilization and prolonged runtime, thus subpar scalability.

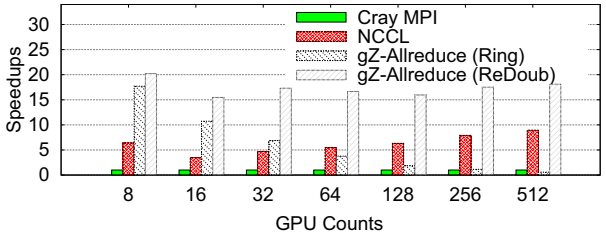


Figure 10: Scalability evaluation of our gZ-Allreduce with Cray MPI and NCCL in different GPU counts.

4.4.2 Collective data movement. In this section, we assess the performance of our *gZCCL* collective data movement framework using the widely-used Scatter operation, comparing it with Cray MPI. We exclude NCCL from this comparison as it has no implementation for Scatter.

Evaluation with different message sizes. We evaluate the performance of our gZ-Scatter with data sizes up to 600 MB, using 64 NVIDIA A100 GPUs on 16 nodes. Figure 11 indicates that our gZ-Scatter is able to consistently outperform Cray MPI across all data sizes. The speedup of gZ-Scatter enhances as the data size increases, achieving its maximum (20.2×) at 600 MB. This demonstrates superior scalability with respect to data sizes, which can be attributed to the reduced message sizes and overlapping of compression, kernel launching, and data movement in our *gZCCL* framework.

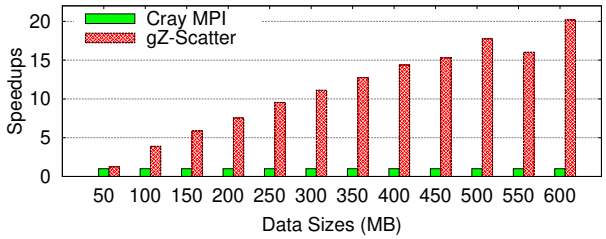


Figure 11: Performance evaluation of our gZ-Scatter with Cray MPI in different data sizes.

Evaluation with different GPU counts. We assess the scalability of our gZ-Scatter with the complete RTM dataset, with a data size of 646 MB, using up to 512 NVIDIA A100 GPUs spread across 128 nodes. From Figure 12, it is evident that our gZ-Scatter outperforms Cray MPI in all cases. As the GPU count increases, the speedup of gZ-Scatter first increases, peaking at 28.7×, and then gradually decreases to 4.75× when the GPU count reaches 512. Unlike the Allreduce scenario, the message size distributed to each non-root GPU in the Scatter communication pattern linearly decreases as the GPU count rises. When the GPU count is less than or equal to 16, the message size on the non-root GPU allows for high GPU utilization, hence the speedup grows with the increasing GPU count. However, when the GPU count exceeds or equals 32, the GPU utilization continues to drop, thereby reducing the collective performance and leading to a decrease in performance enhancement.

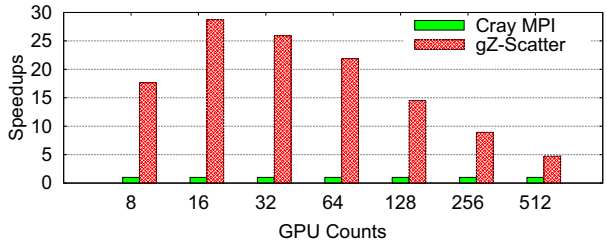


Figure 12: Scalability evaluation of our gZ-Scatter with Cray MPI in different GPU counts.

4.5 Image Stacking Performance Evaluation with Accuracy Analysis

In this section, we employ the image stacking application to evaluate both the performance and accuracy of our *gZCCL*. Image stacking, a technique widely used in various scientific fields such as atmospheric science and geology, is employed to generate high-quality images by stacking multiple individual images, which essentially constitutes an Allreduce operation. As demonstrated by Gurhem in 2021 [10], researchers use MPI to merge these individual images into a comprehensive final image. As can be seen from Table 2, our ring-based *gZCCL* (Ring) outperforms Cray MPI by a factor of 3.99× when using an absolute error bound of 1E-4. Moreover, our recursive doubling-based *gZCCL* (ReDoub) offers even higher performance with speedups of up to 9.26× and 1.69× compared with Cray MPI and NCCL, respectively. This significant performance enhancement arises from the markedly reduced message sizes and compression-related overheads brought by our *gZCCL* framework.

The following text presents a performance breakdown analysis. For *gZCCL* (Ring), 84.08% of the total runtime is consumed by compression, whereas *gZCCL* (ReDoub) has comparable compression and communication costs at 42.61% and 46.28% respectively. This substantial reduction in compression cost is due to higher GPU utilization and fewer compression operations in our optimized gZ-Allreduce (ReDoub) algorithm compared with *gZCCL* (Ring).

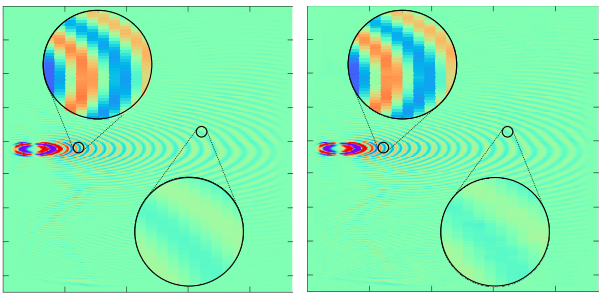
Table 2: Image stacking performance analysis (The speedups are based on Cray MPI. The last four columns are performance breakdowns of our *gZCCL*).

	Speedups	Cmpr.	Comm.	Redu.	Others
***gZCCL* (Ring)**	3.99	84.08%	14.08%	1.26%	0.58%
***gZCCL* (ReDoub)**	9.26	42.61%	46.28%	11.04%	0.06%
NCCL	5.47	No breakdown because of complexity			

In addition to performance analysis, we thoroughly evaluate the accuracy using both visualization method and numerical metrics such as the widely-used peak signal-to-noise ratio (PSNR) [21] and normalized root mean squared error (NRMSE) [24]. Our accuracy-aware design allows *gZCCL* (ReDoub) to achieve excellent reconstructed image quality, even with an error bound of 2E-4, as shown in Figure 13. The reconstructed image of *gZCCL* (Ring) also exhibits high visual quality, similar to that shown in Figure 13b, hence it is not presented separately here. When the error bound is tightened to 1E-4, as used in our performance analysis, *gZCCL* (Ring) reaches a great PSNR of 56.83 and an NRMSE of 1E-3. Meanwhile, *gZCCL* (ReDoub) demonstrates better data quality, achieving a PSNR of 57.80 and an NRMSE of 1E-3. The high accuracy of *gZCCL* confirms a controllable error propagation, which matches the theoretical analysis in [12]. *gZCCL* (ReDoub) exhibits a higher quality of reconstructed data over *gZCCL* (Ring), because of fewer error propagation steps as mentioned in Section 3.3.3.

5 CONCLUSION AND FUTURE WORK

This paper presents *gZCCL*, an innovative framework that optimizes GPU-aware collective communications, offering minimized compression-related overheads and controlled accuracy. We devise



(a) Cray MPI/NCCL (lossless) (b) *gZCCL* (2E-4)

Figure 13: Visualization of final stacking image.

two algorithm design frameworks and two collective optimization frameworks for both compression-enabled collective computation and collective data movement. We integrate the framework into a variety of collective communications including Allgather, Reduce_scatter, Allreduce, and Scatter, demonstrating its generality. Our experiments with up to 512 NVIDIA A100 GPUs illustrate that our gZ-Allreduce surpasses the Allreduce operation in Cray MPI and NCCL by up to 20.2× and 4.5× respectively. In addition, our gZ-Scatter outperforms the Scatter operation in Cray MPI by 28.7×, while NCCL lacks a Scatter implementation. In a nutshell, our work not only addresses the concerns of previous related efforts, such as inefficient GPU utilization, significant compression-related overheads, and inferior performance but also provides a groundwork for further studies in this domain. Our future work will evaluate our *gZCCL* framework with more collective operations and we plan to extend *gZCCL* to more hardware such as FPGAs and AI accelerators.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under Contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant OAC-2003709, OAC-2104023, and OAC-2311875. This research used resources from the Argonne Leadership Computing Facility, a U.S. DOE Office of Science user facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Ahmed M. Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. 2021. An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems. arXiv:2101.10761 [cs.LG]

[3] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. 2005. Optimization of MPI Collective Communication on BlueGene/L Systems. In *Proceedings of the 19th Annual International Conference on Supercomputing* (Cambridge, Massachusetts) *(ICS '05)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/1088149.1088183

[4] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 2017. S-Caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 193–205.

[5] Alan Ayala, Stanimire Tomov, Xi Luo, Hejer Shaeik, Azzam Haidar, George Bosilca, and Jack Dongarra. 2019. Impacts of Multi-GPU MPI collective communications on large FFT computation. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*. IEEE, 12–18.

[6] M. Bayatpour and M. A. Hashmi. 2018. SALaR: Scalable and Adaptive Designs for Large Message Reduction Collectives. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–10. https://doi.org/10.1109/CLUSTER.2018.00009

[7] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of MPI usage on a production supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 386–400.

[8] NVIDIA Corp. 2023. NCCL – Optimized primitives for inter-GPU communication. https://github.com/NVIDIA/nccl.

[9] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 730–739.

[10] Jérôme Gurhem, Henri Calandra, and Serge G. Petiton. 2021. Parallel and Distributed Task-Based Kirchhoff Seismic Pre-Stack Depth Migration Application. In *2021 20th International Symposium on Parallel and Distributed Computing (ISPDC)*. 65–72. https://doi.org/10.1109/ISPDC52870.2021.9521599

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[12] Jiajun Huang, Sheng Di, Xiaodong Yu, Yujia Zhai, Zhaorui Zhang, Jinyang Liu, Xiaoyi Lu, Ken Raffenetti, Hui Zhou, Kai Zhao, Zizhong Chen, Franck Cappello, Yanfei Guo, and Rajeev Thakur. 2023. An Optimized Error-controlled MPI Collective Framework Integrated with Lossy Compression. arXiv:2304.03890 [cs.DC]

[13] Jiajun Huang, Jinyang Liu, Sheng Di, Yujia Zhai, Zizhe Jian, Shixun Wu, Kai Zhao, Zizhong Chen, Yanfei Guo, and Franck Cappello. 2023. Exploring Wavelet Transform Usages for Error-bounded Scientific Data Compression. In *2023 IEEE International Conference on Big Data (BigData)*. 4233–4239. https://doi.org/10.1109/BigData59044.2023.10386386

[14] Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. 2023. cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Article 43, 13 pages. https://doi.org/10.1145/3581784.3607048

[15] Arpan Jain, Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. 2019. Scaling TensorFlow, PyTorch, and MXNet using MVAPICH2 for High-Performance Deep Learning on Frontera. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 76–83.

[16] Suha Kayum et al. 2020. GeoDRIVE – A high performance computing flexible platform for seismic applications. *First Break* 38, 2 (2020), 97–100.

[17] Argonne National Laboratory. 2023. MPICH – A high-performance and widely portable implementation of the MPI-4.0 standard. https://www.mpich.org.

[18] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20 (2014), 2674–2683.

[19] Jinyang Liu, Sheng Di, Kai Zhao, Xin Liang, Sian Jin, Zizhe Jian, Jiajun Huang, Shixun Wu, Zizhong Chen, and Franck Cappello. 2024. High-performance Effective Scientific Error-bounded Lossy Compression with Auto-tuned Multi-component Interpolation. *Proc. ACM Manag. Data* 2, 1, Article 4 (mar 2024), 27 pages. https://doi.org/10.1145/3639259

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

[21] Sanjith S., R. Ganesan, and Rimal Isaac. 2015. Experimental Analysis of Compacted Satellite Image Quality Using Different Compression Methods. *Advanced Science* 7 (03 2015). https://doi.org/10.1166/asem.2015.1673

[22] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An In-Depth Analysis of the Slingshot Interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. https://doi.org/10.1109/sc41405.2020.00039

[23] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[24] Maxim Vladimirovich Shcherbakov, Adriaan Brebels, Nataliya Lvovna Shcherbakova, Anton Pavlovich Tyukov, Timur Alexandrovich Janovsky, Valeriy Anatol'evich Kamaev, et al. 2013. A survey of forecast error measures. *World applied sciences journal* 24, 24 (2013), 171–176.

[25] Dingwen Tao, Sheng Di, and Franck Cappello. 2017. Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization. https://doi.org/10.1109/IPDPS.2017.115

[26] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[27] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2020. Significantly improving lossy compression for HPC datasets with second-order prediction and parameter optimization. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 89–100.

[28] 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)*. 22–31. https://doi.org/10.1109/HiPC56025.2022.00016

[29] 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)*. 134–144. https://doi.org/10.1109/IPDPS54959.2023.00023

[30] Q. Zhou, C. Chu, N. S. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. 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)*. 444–453. https://doi.org/10.1109/IPDPS49936.2021.00053

[31] 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 *High Performance Computing: 37th International Conference, ISC High Performance 2022, Hamburg, Germany, May 29 – June 2, 2022, Proceedings* (Hamburg, Germany). Springer-Verlag, Berlin, Heidelberg, 3–25. https://doi.org/10.1007/978-3-031-07312-0_1