Accelerating MPI AllReduce Communication with Efficient GPU-Based Compression Schemes on Modern GPU Clusters

Qinghua Zhou, Bharath Ramesh, Aamir Shafi, Mustafa Abduljabbar,
Hari Subramoni and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, *The Ohio State University*
{zhou.2595, ramesh.113, shafi.16, abduljabbar.1, subramoni.1, panda.2}@osu.edu

***Abstract*—With the increasing scale of High-Performance Computing (HPC) and Deep Learning (DL) applications through GPU adaptation, the seamless communication of data stored on GPUs has become a critical factor in enhancing overall application performance. AllReduce is a communication collective operation that is commonly used in HPC applications and distributed DL training, especially Data Parallelism. Data Parallelism is a common strategy where parallel GPUs are used to process the partitioned training dataset using a replica of the DL model. However, AllReduce operation for large GPU data still performs poorly due to the limited interconnect bandwidth between the GPU nodes. Some strategies of Gradient Quantization or Sparse AllReduce modifying the Stochastic Gradient Descent (SGD) algorithms may not support different training scenarios. Recent research shows integrating GPU-based compression into MPI libraries is efficient to achieve faster data transmission. In this paper, we propose optimized Recursive-Doubling and Ring AllReduce algorithms that encompass efficient collective-level GPU-based compression schemes in a state-of-the-art GPU-Aware MPI library. At the microbenchmark level, the proposed Recursive-Doubling and Ring algorithms with compression support achieve benefits of up to 75.3% and 85.5% respectively compared to the baseline, and 24.8% and 66.1% respectively compared to naive point-to-point compression on modern GPU clusters. For distributed DL training with PyTorch-DDP, these two approaches yield up to 32.3% and 35.7% faster training than the baseline, while maintaining similar accuracy.**

***Index Terms*—AllReduce, Compression, GPU-Aware MPI, Deep Learning, Data Parallelism**

I. Introduction

Over the past few years, deep neural networks (DNNs) have enabled tremendous advances in areas like natural language processing, image classification, and self-driving cars. DL frameworks like PyTorch [1] and TensorFlow [2] make training—the compute-intensive part of developing DNNs—efficient by supporting parallel execution on systems with graphics processing units (GPUs). These GPU systems are key to training the large DNNs that can automatically extract features from complex datasets and uncover non-linear relationships between them. On the other hand, GPU vendors—including NVIDIA, AMD, and Intel—are building HPC systems to meet the computational needs of both traditional scientific and DL applications. Given the distributed nature and significant scale of these systems, the communication performance between GPUs is crucial for ensuring optimal overall performance of parallel applications.

The Message Passing Interface (MPI) [3], widely recognized as the dominant parallel programming model, offers a wide range of communication primitives to facilitate parallel and distributed execution of applications on HPC systems. MPI communication libraries [4], [5] have been commonly employed to parallelize traditional scientific applications in the past. They have also been extensively embraced recently by DL frameworks as a communication backend for distributed training [6], [7] that involves intensive computation.

A. Problem Statement

AllReduce is a communication collective operation that is commonly used in HPC applications as well as distributed DL training. *Data Parallelism* is a common strategy used in distributed DL training where parallel GPUs are used to process the partitioned training dataset using a replica of the DL model. DDP (Distributed Data Parallel) training [8] extends the Data Parallelism to multiple computing nodes—a strategy adopted by PyTorch. This parallelism can combine with other parallel strategies [9], [10] to form hybrid parallelism.

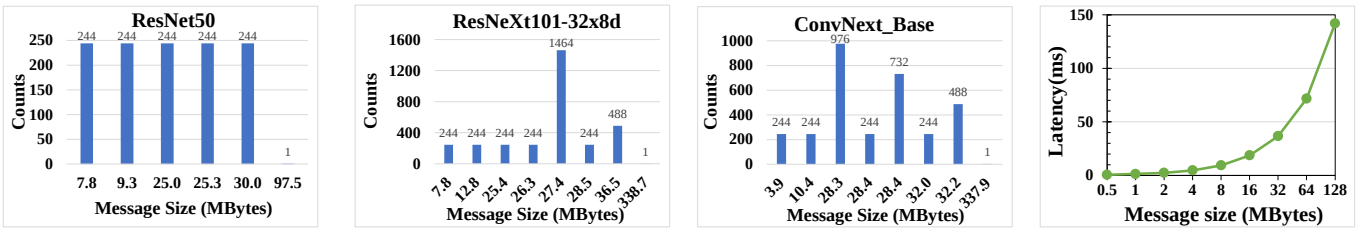
Many Allreduce algorithms have been developed in the past, such as Baidu Ring [11], NCCL Ring [12] and double binary tree [13], Link-Efficient NVGroup algorithm [14], etc. However, Allreduce operation can still be a bottleneck, especially for large GPU data during distributed DL training. Recent studies explored the Gradient Quantization [15] or Sparse AllReduce [16] based on the sparsity of the gradient data. These solutions usually require specific modified Stochastic Gradient Descent(SGD) algorithms [16]. However, the modification at the application level may not support different training scenarios. The effectiveness of these solutions is contingent on their compatibility with the specific SGD algorithms.

Therefore, **this paper aims to optimize the AllReduce communication operation by designing general GPU-based compression schemes at the communication middleware level while preserving similar accuracy and without requiring modifications to the applications**.

B. Motivation

To analyze the communication bottleneck in DNN training using DDP, we conduct profiling of representative DL models

*This research is supported in part by NSF grants #1818253, #1854828, #2007991, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002.



(a) ResNet50 (b) ResNeXt101-32x8d (c) ConvNeXt_Base (d) AllReduce communication latency

Fig. 1: Message sizes of Allreduce in PyTorch DDP Training with 8 GPUs (4 nodes) on Pitzer [17]. Large message sizes are transferred for three DNN models. The communication latency increases rapidly with increasing message sizes in this range.

including ResNet50 [18], ResNext101-32x8d [19], and ConvNeXt_Base [20] using CIFAR10 [21] dataset. The goal is to determine the message sizes of AllReduce operation. Figures 1(a), 1(b) and 1(c) represent a frequency histogram of the message sizes for AllReduce communication primitives in the training of 5 epochs for the identified models using a state-of-the-art GPU-Aware MPI library with 8 V100 GPUs on Pitzer [17] cluster. These figures show that large messages (>1MB) need to be exchanged between GPUs. For large message sizes, the communication latency of AllReduce increases rapidly with increasing message sizes as shown in Figure 1(d). We observe that AllReduce communication primitives add large overheads to the DNN training—this is due to the limited interconnect bandwidth between the GPU node.

C. Challenges and Proposed Solutions

In the state-of-the-art GPU-aware MPI libraries for GPU communication [14], [22], inter-node bandwidth has become a primary bottleneck [23] for transferring large data between multiple GPU nodes. Due to the saturated network bandwidth, other optimization strategies could be explored to enhance AllReduce communication performance.

Recent research [24] proposed On-the-fly compression by leveraging advanced GPU-based compression algorithms [25], [26] to accelerate point-to-point GPU communication for the state-of-the-art MPI library. The AllReduce in MPI libraries [4], [5] may benefit from using naive compression for point-to-point communications. However, inside the advanced AllReduce algorithms, non-blocking send and receive operations are usually utilized to transfer the GPU data. Such non-blocking operations could be blocked by compression/decompression operations triggered inside the send or receive operations. Naive point-to-point compression for each transmission operation may introduce redundant compression/decompression operations with unnecessary overheads.

To enable compression for AllReduce to achieve optimal performance, this paper addresses the following research challenge: **What is the limitation of using naive point-to-point compression technique for advanced AllReduce algorithms?** First, we need to select the appropriate advanced AllReduce algorithms for transferring large GPU data and analyze the limitations of using naive point-to-point compression.

To implement compression at the collective level, recent work [27] and [28] proposed optimization strategies for MPI_Alltoall and MPI_Bcast operations, respectively. However, the proposed optimizations cannot be directly applied to MPI_AllReduce since the communication patterns in MPI_AllReduce are different than the other two operations.

We need to explore new implementations to improve the performance of AllReduce with GPU-based compression and accelerate the distributed DL training. This paper tackles the second challenge: **How to co-design and optimize the GPU-based compression at the collective level along with the communication patterns of advanced AllReduce algorithms for distributed DNN training?**

D. Contributions

This paper makes the following main contributions:

1) Analyze the shortcomings of using the naive Point-to-Point compression in the Ring and Recursive-Doubling AllReduce algorithms. We point out optimization opportunities for optimizing the compression at the collective level for AllReduce. (Section III-B)
2) Optimize the data flow of AllReduce by co-designing compression operations with the communication pattern of the Ring and Recursive-Doubling algorithms. This is to remove redundant compression operations in the naive point-to-point compression solution and enable overlapping compression/decompression kernels with send and receive operations.(Section III-C, III-D, III-E, III-F)
3) Evaluate the communication latency of compression-enabled MPI_Allreduce operations using the OSU Micro Benchmark (OMB) suite. Our proposed Recursive-Doubling and Ring algorithms with compression support achieve benefits of up to 75.3% and 85.5% respectively compared to the baseline in a state-of-the-art MPI library on modern GPU clusters. (Section IV)
4) Evaluate the performance benefits of the proposed designs on the DDP training of representative DNN models with PyTorch. With the proposed Ring and Recursive-Doubling AllReduce with Collective-level compression designs, the training time can be reduced by up to 32.3% and 35.7% compared to the baseline, respectively (while keeping similar convergent accuracy). (Section V)
5) To the best of our knowledge, this is the first work that leverages the GPU-based compression technique and optimizes the compression at the collective level to accelerate AllReduce communication for DL training performance. (Section VI)

II. Background

In this section, we introduce the background knowledge including GPU-Aware MPI libraries, MPI_Allreduce algorithms in MPI libraries, and GPU-based compression libraries.

A. GPU-Aware MPI libraries with GPUDirect technology

MPI libraries like MVAPICH2 [5] and OpenMPI [4] offer direct support for passing GPU buffers to MPI primitives. Applications using these *GPU-Aware* MPI libraries no longer require explicit copying of GPU buffers to CPUs. These MPI libraries have implemented various optimization strategies including CUDA IPC (Inter-Process Communication), Pipelining, and GPUDirect RDMA [22], specifically for point-to-point GPU-based communication. NVIDIA GPUDirect technology [29] enables Peer-to-Peer access, allowing data to be directly shared between GPUs on the same PCIe bus or via the advanced NVLink interconnect. The RDMA (Remote Direct Memory Access) facilitates extended memory access between GPUs and other PCIe devices. With the optimization schemes, the communication performance of GPU data transfer is significantly accelerated across various communication paths.

B. Algorithms for AllReduce Operation

The AllReduce operation allows multiple processing units to exchange data and combine them into a global result using a specified operator. Many advanced Allreduce schemes have been developed such as Baidu Ring [11], NCCL Ring [12] and double binary tree algorithm [13], Link-Efficient NVGroup algorithm [14], etc. The selection of suitable AllReduce algorithms can be based on factors such as message sizes, the number of processes, and the system configuration.

In state-of-the-art MPI libraries [4], [5] and NVIDIA's Collective Communication Library (NCCL) [12], Ring and Recursive-Doubleing algorithms [30] are often selected for large message transfer in AllReduce Communication.

C. General-purpose GPU-based Compression Libraries

The performance of compression algorithms MPC [25], nvComp [31], ZFP [26], SZ [32] has significantly improved thanks to the advanced computing power of GPUs, surpassing CPU-based algorithms. Existing lossless GPU-based compression libraries like MPC [25] are often hindered by a low compression ratio when applied to generic data. A recent study [33] has shown that for certain HPC applications, lossy compression algorithms with higher compression ratios have been utilized, as opposed to lossless compression libraries. The ZFP [26] library offers numerous interfaces that can accommodate multi-dimensional scientific data and different compression modes. The cuZFP, implemented with CUDA, enables lossy compression with a fixed compression rate. In this mode, a d-dimensional array of values is divided into 4^d blocks. The fixed compression rate mode guarantees the expected compression ratio regardless of the sparsity of the data.

III. Proposed Collective-level Online Compression Designs for AllReduce Communication

In this section, we will first review the existing AllReduce algorithms and pick up the appropriate algorithms for the DL workloads. We also analyze the limitations of using naive point-to-point compression solutions in these algorithms. Then we propose new compression designs at the collective level to further improve the performance of AllReduce operation.

A. Existing Allreduce algorithms

The Ring-based AllReduce algorithm [30] establishes a logical ring structure, facilitating the exchange of data between each process and its adjacent neighbors. Each rank partitions the data into N (the total number of processes) multiple chunks. Then each rank sends the chunks to its right neighbor rank, receive the data chunk from its left neighbor rank and run a reduction operation to aggregate the received data chunk with the corresponding chunk in its own data buffer. Finally, each rank receives the reduction of chunks across other ranks. Since the element-wise reduction operation is expensive, state-of-art GPU-Aware MPI libraries [4], [5] have developed GPU reduction kernels [34] to accelerate the performance of reduction. Nevertheless, the Ring algorithm may not always be the optimal solution when the size of each data chunk becomes much smaller for smaller message sizes or more processes.

On the other hand, in the Recursive-Doubling algorithm, specific pairs of processes exchange messages with each other in a pairwise manner. The Recursive-Doubling algorithm with a reduction kernel has been proven to achieve good performance [34] since fewer data exchanges are needed across the rank. The whole data on each rank is aggregated with the data on other ranks without partitioning it into multiple chunks.

In this paper, we pick up the Ring and Recursive-Doubling algorithms with GPU reduction kernel in the state-of-the-art GPU-Aware MPI libraries to study the impact of applying compression for AllReduce communication.

B. Limitations of using naive Point-to-Point Compression

Although the naive point-to-point compression solution [24] has proven to be an effective way to reduce the size of transferred data through the network, directly applying it to the Ring and Recursive-Doubling AllReduce algorithms may not achieve optimal performance. Because these collectives are usually layered on top of Point-to-Point operations, such compression inside the point-to-point may lose visibility into what the higher-level collective algorithm wants to achieve.

1) ***Limitations in Ring algorithm****:* During the Ring AllReduce operation, the data chunks present in the send buffer of each rank will be moved to the receive buffer of all the other ranks. When using a basic point-to-point compression method, every send and receive operation involves performing compression on the sender's side and decompression on the receiver's side. Upon receiving compressed data from its left neighbor rank, each rank will decompress the data within the MPI_Irecv operation. However, when the next MPI_Isend

sends this data to its right neighbor, compression will still be initiated. Since the received data is already compressed, such compression operation becomes redundant for exchanging the final aggregated results in the second phase of Ring algorithm.

In addition, the original non-blocking MPI_Isend and MPI_Irecv operations often offer opportunities for overlap. However, when compression is introduced, each send or receive operation needs to wait for the completion of inside compression/decompression which causes the delay of subsequent send or receive operations Consequently, the compression kernel within one send operation cannot overlap with the compression kernels in other send/receive operations.

*2) **Limitations in Recursive-Doubling algorithm**:* In the existing Recursive-Doubling AllReduce algorithm, each pair of two GPUs exchange data between each other. On each rank, the reduction operation follows the receive operation to aggregate the received data with the data on its own GPU.

Applying the naive point-to-point compression to the recursive-doubling algorithm incurs similar limitations. In the original recursive-doubling algorithm, the non-blocking MPI_Isend and MPI_Irecv operation between two processes could achieve some overlapping. Similarly, these non-blocking send and receive operations will be blocked by the point-to-point compression/decompression solution. These compression/decompression overheads can not be reduced by overlapping with the send/receive operations.

Furthermore, in both algorithms, the decompression kernel inside the receive operation is usually launched into a different CUDA stream other than that for the following reduction operation. Explicit stream synchronization must be added before the reduction operation, which introduces extra overheads.

To tackle the limitation of using point-to-point compression for MPI_Allreduce, we redesign both the Ring and Recursive-Doubling MPI_Allreduce and algorithms with collective-level online compression in a state-of-the-art MPI library to accelerate the communication of GPU data.

C. Overview of Ring-based Allreduce Communication with Collective-level Online Compression

Figure 2 shows the data flow in the Ring MPI_Allreduce operation with Collective-level compression. In this example, 4 GPUs have data A, B, C, and D on their own buffers respectively. These data will be aggregated for all the GPUs. Each GPU partitions the data into N (N=4) chunks. There are two phases in this algorithm. In the first phase, the data chunk with the same index on each GPU will be aggregated together and stored in the corresponding GPU. For example, data chunk A0, B0, C0, D0 will be aggregated into GPU0.

Each GPU transfers data chunks with different indices to its logical right rank at each iteration and receives data chunks from the logical left rank. For example, in the first iteration, A3 on GPU0 is compressed using a compression kernel and sent to GPU1 using MPI_Isend while D2 is compressed on GPU3 and sent to GPU0. Once GPU1 receives the A3', it will first launch the decompression kernel to restore the data followed by a reduction kernel to aggregate A3 and B3. The

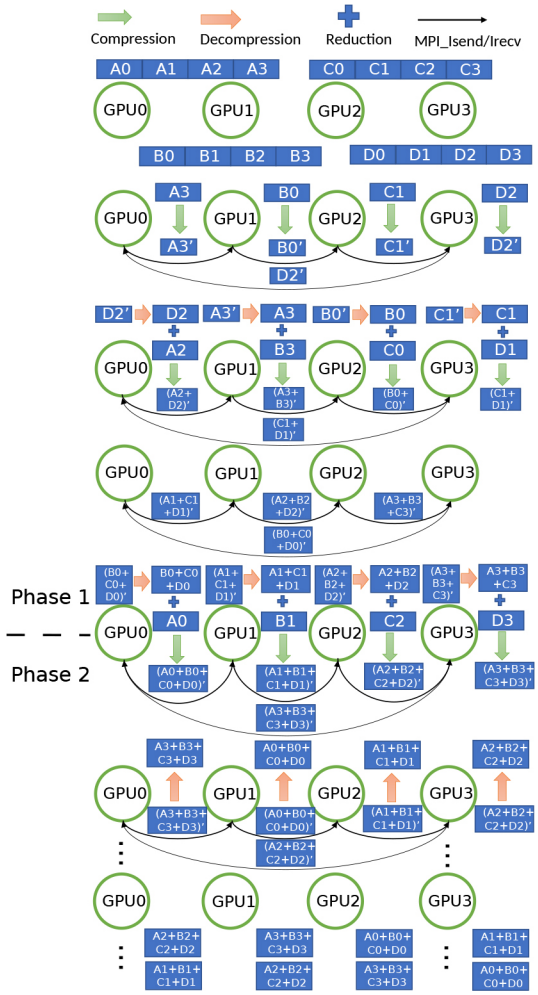


Fig. 2: Data flow of Ring-based MPI_Allreduce with Collective-level online compression.

result of (A3+B3) is compressed and then transferred to GPU2. In the next iteration, the received compressed (A2+B2)' will be decompressed first and aggregated with D2 (This process is skipped in the figure). The result (A2+B2+D2) is then compressed to (A2+B2+D2)' and sent to GPU2. In the next iteration, GPU1 receives the (A1+C1+D1)' and repeats the same procedure to get the aggregated result of (A1+B1+C1+D1). Similar procedures are executed on other GPUs.

In the second phase, the aggregated results of different chunks on each GPU (e.g., (A0+B0+C0+D0) on GPU0) need to be transferred to all the other GPUs. They will only be compressed once in this phase. Once other GPUs receive these compressed data chunks with aggregated results, the MPI_Isend is posted immediately to directly send the received compressed data chunk to the logical right rank. A following decompression kernel is immediately launched on a non-default CUDA stream to restore the data. Since no reduction operation is involved in this phase, these decompression kernels launched on multiple CUDA streams for different received data chunks can have opportunities to overlap.

Figure 3 depicts the detailed operations inside the GPU0

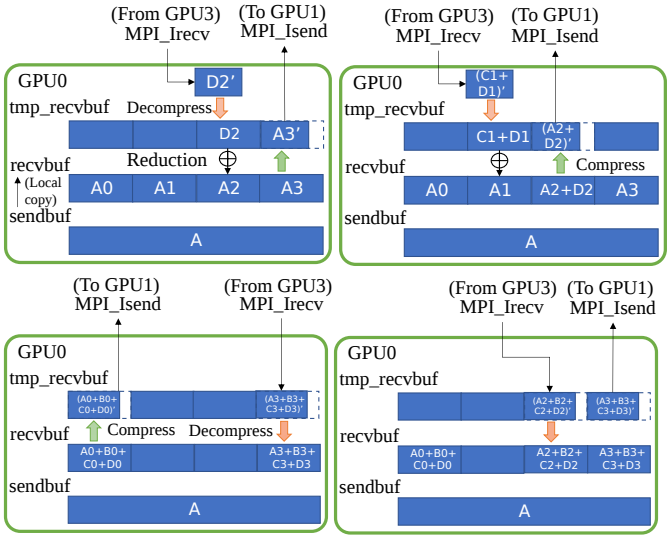


Fig. 3: Detailed operations inside GPU of Ring MPI_Allreduce with Collective-level online compression.

of the Ring MPI_Allreduce with Collective-level online compression. First, data is copied from the send buffer $sendbuf$ to the local receive buffer $recvbuf$ directly using a Device to Device cudaMemcpy. A pre-allocated temporary receive buffer $tmp\_recvbuf$ on GPU stores the intermediate compressed data and the decompressed data. When data chunk A3 is compressed on a non-default CUDA stream, an MPI_Irecv operation is posted for receiving compressed data D2' from GPU3. The compression kernel has the opportunity to overlap with the receive operation. Once the compression kernel is completed, the compressed A3' is sent out to GPU1 by MPI_Isend. At the same time, a decompression kernel is launched immediately on a non-default CUDA stream for D2' after the receive operation for D2' is finished. The following reduction kernel is launched on the same CUDA stream without the need to add stream synchronization before the reduction operation. These decompression and reduction kernels have opportunities to overlap with the MPI_Isend operation to hide partial of the computing overheads. In the next iteration, the compression kernel is launched on this same CUDA stream for the previous reduction to generate the result (A2+D2)'. GPU0 repeats the same procedure to receive compressed (C1+D1)' and send out the (A2+D2)'.

In the second phase(shown in the lower two subfigures), we achieve similar overlap. We will not wait for the completion of MPI_Isend and decompression kernel to start the next iteration. In the next iteration, MPI_Isend directly sends out the previously received compressed (A3+B3+C3+D3)' to GPU1. In the naive point-to-point compression solution, there is redundant compression to regenerate the (A3+B3+C3+D3)' from (A3+B3+C3+D3). The decompression kernel follows the receive operation to restore the (A2+B2+C2+D2)' and has opportunities to overlap with the decompression kernel in the previous iteration. After all the iterations, we add wait for all the decompression kernels and MPI_Isend operations.

D. Algorithm of Ring-based Allreduce Communication with Collective-level Online Compression

Algorithm 1: Collective-level Online Compression Design for Ring MPI_Allreduce

Input : Send buffer S, Data size M, Control parameters A, Preallocated GPU buffer S_tmp, R_tmp, Send Request Array $Sreq$, Receive Request Array $Rreq$, Reduction Operation Op, GPU counts N
Output: Receive buffer R, Compressed data size B

1 **for** $i = 0$ *to* $N-2$ **do**
2 $\quad s_i = (rank - i + N)\%N; //send\_index$
3 $\quad r_i = (rank - i - 1 + N)\%N; //receive\_index$
4 $\quad$ **if** $i == 0$ **then**
5 $\quad\quad (B, R\_tmp_{si})$=zfp_compress_coll($R_{si}$, A, $Stream_{si}$);
6 $\quad$ MPI_Irecv(R_tmp_{ri}, B, left, $Rreq_i$, ...); //Receive from left
7 $\quad$ **if** $i == 0$ **then**
8 $\quad\quad$ cudaStreamSynchronize($Stream_{si}$);
9 $\quad$ **else**
10 $\quad\quad$ cudaStreamSynchronize($Stream_{ri}$);
11 $\quad\quad (B, R\_tmp_{si})$=zfp_compress_coll($R_{ri}$, A, $Stream_{si}$);
12 $\quad\quad$ cudaStreamSynchronize($Stream_{si}$);
13 $\quad$ MPI_Isend(R_tmp_{si}, B, right, $Sreq_i$, ...); // Send to right
14 $\quad$ Wait for $Rreq_i$;
15 $\quad S\_tmp_{ri}$ = zfp_decompress_coll(R_tmp_{ri}, B, A, $Stream_{ri}$);
16 $\quad R_{ri}$ = reduction_coll(S_tmp_{ri}, R_{ri}, M, Op, $Stream_{ri}$);
17 $r_i = (rank + 1)\%N; //receive\_index$
18 cudaStreamSynchronize($Stream_{ri}$);//Wait for last reduction kernel
19 Wait for all $Sreq_i$;
20 **for** $i = 0$ *to* $N-2$ **do**
21 $\quad s_i = (rank - i + 1 + N)\%N; //send\_index$
22 $\quad r_i = (rank - i + N)\%N; //receive\_index$
23 $\quad$ **if** $i == 0$ **then**
24 $\quad\quad (B, R\_tmp_{si})$=zfp_compress_coll($R_{si}$, A, $Stream_{si}$);
25 $\quad$ MPI_Irecv(R_tmp_{ri}, B, left, $Rreq_i$, ...); //Receive from left
26 $\quad$ **if** $i == 0$ **then**
27 $\quad\quad$ cudaStreamSynchronize($Stream_{si}$);
28 $\quad$ MPI_Isend(R_tmp_{si}, B, right, $Sreq_i$, ...); // Send to right
29 $\quad$ Wait for $Rreq_i$;
30 $\quad R_{ri}$ = zfp_decompress_coll(R_tmp_{ri}, B, A, $Stream_{ri}$);
31 **for** $i = 0$ *to* $N-2$ **do**
32 $\quad$ cudaStreamSynchronize($Stream_{ri}$);
33 Wait for all $Sreq_i$;

Algorithm 1 is a high-level overview of the Ring-based AllReduce with Collective-level online compression. We leverage the ZFP compression library for compression/decompression. We use runtime control parameters to specify the compression-related parameters. The wrapper functions $zfp\_compress\_coll$ and $zfp\_decompress\_coll$ calls ZFP APIs to execute the compression and decompression kernels, respectively. The two for loops correspond to the two phases in the Ring algorithm. On each rank, the compression/decompression kernels are launched on multiple non-default CUDA streams (Line 5,11,15,24,30). The reduction kernel is launched on the same stream of decompression (Line 16). In the first phase, we wait for all the send requests after all the iterations (Line 19). Similarly, we move the wait after all the iterations in the second phase (Line 33). The cudaStreamSynchronize is launched (Line 32) for the completion of all the decompression kernels after the iterations so that decompression will not block the send or receive operations.

E. Overview of Recursive-Doubling Allreduce Communication with Collective-level Online Compression

Figure 4 depicts the data flow in the Recursive-Doubling MPI_AllReduce operation with Collective-level compression. For simplicity, we show the example of 4 GPUs. In the following section, Algorithm 2 explains the case that the total number of GPUs is non-power of 2. The whole data on each rank will be compressed. Each pair of GPUs exchange the compressed data. For example, data A is compressed to A' on GPU0 and sent to GPU1 by MPI_Isend. GPU0 also posts MPI_Irecv to receive data B' from GPU1. The pair of GPU2 and GPU3 follow the same flow to exchange data C and D.

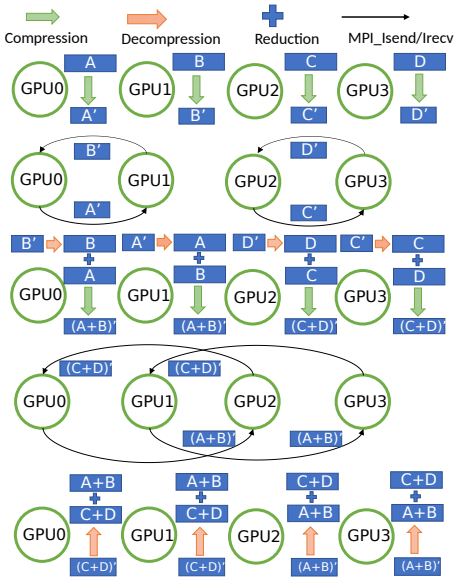


Fig. 4: Data flow of Recursive-Doubling MPI_Allreduce with Collective-level online compression.

Once GPU0 receives the B' from GPU1, a decompression kernel is launched on a non-default CUDA stream to restore the compressed B' to B. The following reduction kernel is launched on the same CUDA stream to aggregate B with A. Then we launch the compression kernel to generate the compressed (A+B)', which executes on the same CUDA stream as the reduction kernel so that no explicit synchronization is needed between them. Once the compression kernel is completed, MPI_Isend is posted to send (A+B)' to GPU2. Then we wait for the completion of the receive operation for (C+D)' and launch the decompression kernel on a non-default CUDA stream to restore (C+D)' to C+D followed by a reduction kernel on the same CUDA stream. Finally, we wait for the last reduction kernel and all the send operations. The flow on GPU1, GPU2, and GPU3 are similar.

Figure 5 depicts the detailed operations inside GPU0 of the Recursive-Doubling MPI_Allreduce with Collective-level online compression. The pre-allocate temporary send buffer $tmp\_sendbuf$ and temporary receive buffer $tmp\_recvbuf$ on GPU is used to send out the compressed data and store the received data. Similar to the Ring algorithm, the receive operation for B' is not blocked by the compression kernel. Once compression is finished, MPI_Isend operation is posted to send out the compressed data A'. In naive point-to-point compression, the MPI_Irecv operation for B' can only be issued after the completion of compression. Once B' is received, a decompression kernel is launched immediately on a non-default CUDA stream to store data B' to B. A following reduction kernel is launched on the same CUDA stream to aggregate B with A in the $recvbuf$. We do not wait for the completion of the reduction kernel and MPI_Isend operation to post the next MPI_Irev to receive the data from GPU2. During this period, the compression kernel is launched to generate (A+B)'. This compression kernel has the opportunity to overlap with the MPI_Irev operation. Next, when the MPI_Isend is posted to send out the (A+B)', the decompression kernel restoring the (C+D)' and reduction kernel aggregating (C+D) with (A+B) will have the opportunity to overlap with the send operation.

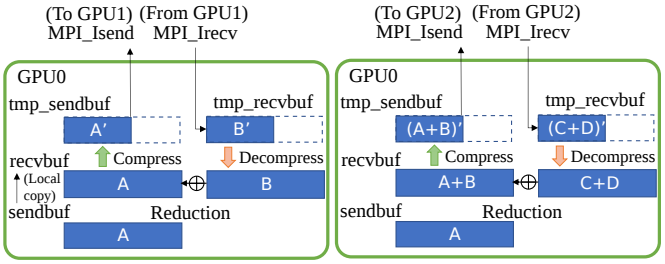


Fig. 5: Detailed operations inside GPU of Recursive-Doubling MPI_Allreduce with Collective-level online compression.

F. Algorithm of Recursive-Doubling Allreduce Communication with Collective-level Online Compression

Algorithm 2 is a high-level overview of the Recursive-Doubling AllReduce with Collective-level online compression. We also use the same wrapper functions $zfp\_compress\_coll$ and $zfp\_decompress\_coll$. The nearest lower power of 2 is first calculated based on the GPU counts N (Line 1). If there are remainder processes ($rem > 0$), all even-numbered processes of $rank$ ($< 2 \times rem$) compress and send data to $rank + 1$ (Line 6). The odd-numbered process of $rank$ ($< 2 \times rem$) receives the compressed data(Line 9), run decompression and reduction(Line 10). After a nice power-of-two recursive-doubling exchange data of the remaining processes, the odd-numbered process of $rank$ ($< 2 \times rem$) sends the compressed result to $rank - 1$ (Line 45). Similar to the Ring algorithm, we launch the reduction kernel to the same stream for decompression (Line 10, 34). We wait for all the send requests after the while loop for all the iterations (Line 39).

IV. Microbenchmark Results and Analysis

We run experiments on four GPU clusters: Frontera Liquid [35] on the Texas Advanced Computing Center (TACC) Frontera supercomputer, Pitzer [17] on the Ohio Supercomputer Center, Lassen [36] Supercomputer of the Lawrence Livermore National Laboratory and MRI cluster. Pitzer is equipped with Dual or Quad NVIDIA V100 GPUs and Infiniband EDR(one way 100 Gb/s) between nodes. Frontera Liquid is

equipped with NVIDIA Quadro RTX 5000 GPUs. Lassen is equipped with Quad NVIDIA V100 GPUs and Infiniband EDR between nodes. More technical details can be found on their official websites. MRI is an in-house cluster where each A100 node is fitted with dual-socket AMD Milan 7713 processors (64 cores) and 2 NVIDIA A100 GPUs. The nodes are interconnected using Mellanox ConnectX-6 HDR (200 Gb/s).

Algorithm 2: Collective-level Online Compression Design for Recursive-Doubling MPI_Allreduce

Input : Send buffer S, Data size M, Control parameters A, Preallocated GPU buffer S_tmp, R_tmp, Send Request Array $Sreq$, Receive Request Array $Rreq$, Reduction Operation Op, GPU counts N, current rank $rank$
Output: Receive buffer R, Compressed data size B

pof2 = $2^{\lfloor \log_2(N) \rfloor}$;
rem = N - pof2;
newrank = 0;
if $rank < 2 \times rem$ **then**
  if *rank % 2 == 0* **then**
    Run compression and send compressed data to rank+1;
    newrank = -1;
  else
    Receive compressed data from rank-1;
    Run decompression and reduction with its own data;
    newrank = rank / 2;
else
  newrank = rank - rem;
if $newrank != -1$ **then**
  mask = 0x1;
  while $mask < pof2$ **do**
    newdst = newrank $\oplus mask$;
    dst = (newdst < rem) ? newdst * 2 + 1 : newdst + rem;
    if $mask == 0x1$ **then**
      (B, S_tmp)=zfp_compress_coll(S, A, $Stream_{dst}$);
    mask2 = mask << 1;
    newdst2 = newrank $\oplus mask2$;
    dst2 = (newdst2 < rem) ? newdst2 * 2 + 1 : newdst2 + rem;
    MPI_Irecv(R_tmp, B, dst, $Rreq_{dst}$, ...);
    if $mask > 0x1$ **then**
      cudaStreamSynchronize($Stream_{rdst}$);
      (B, S_tmp)=zfp_compress_coll(R, A, $Stream_{dst}$);
      cudaStreamSynchronize($Stream_{dst}$);
    else
      cudaStreamSynchronize($Stream_{dst}$);
    MPI_Isend(S_tmp, B, dst, $Sreq_{dst}$, ...);
    Wait for $Rreq_{dst}$;
    R_tmp2 = zfp_decompress_coll(R_tmp, B, A, $Stream_{rdst2}$);
    R = reduction_coll(R_tmp2, R, M, Op, $Stream_{rdst2}$);
    mask <<= 1;
  newdst2 = newrank $\oplus mask2$;
  dst2=(newdst2 < rem)? newdst2*2 + 1:newdst2 + rem;
  cudaStreamSynchronize($Stream_{rdst2}$); // Wait for last reduction
  Wait for all $Sreq_{dst}$;
if $rank < 2 \times rem$ **then**
  if *rank % 2 == 0* **then**
    Receive compressed data from rank+1;
    Run decompression;
  else
    Run compression and send compressed data to rank-1;

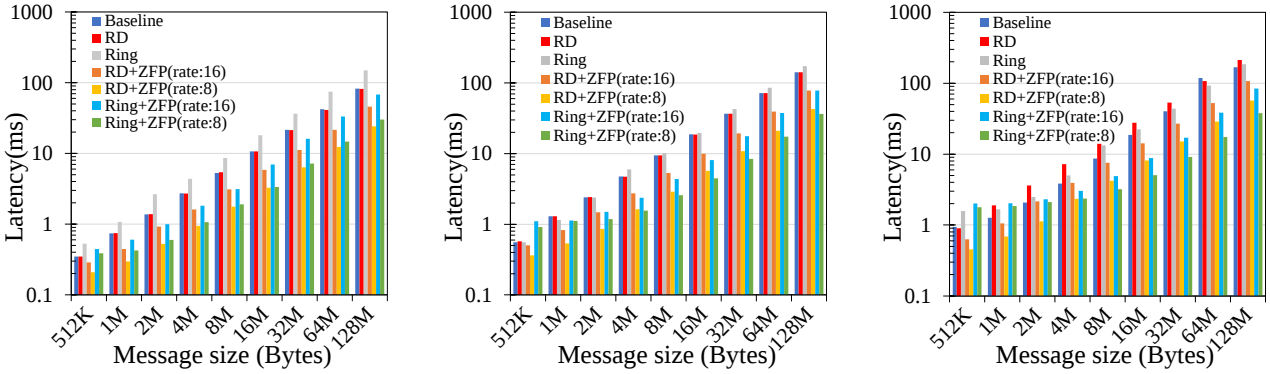
A. MPI_Allreduce Communication Latency

We run the OSU Micro-Benchmark suite (OMB) to evaluate the AllReduce communication latency of GPU data. Figure 6, 7 and 8 show the AllReduce communication latency of GPU-resident data with sizes from 512KB to 128MB on Pitzer, MRI, and Frontera Liquid systems. Figure 9 shows results for message sizes up to 512MB on the Lassen system. We do not show smaller message sizes since we focus on the large GPU data transfer in AllReduce as shown in Figure 1. For smaller message sizes (e.g., < 512KB), the inherited overheads of compression/decompression kernels may exceed the communication benefits of reduced data size. This has been studied in the paper [24]. "RD" means the basic Recursive-Doubling AllReduce algorithm without compression. "Ring" is for the Ring algorithm. "RD+ZFP" and "Ring+ZFP" stand for our proposed designs with collective-level compression using ZFP algorithm. The "Baseline" corresponds to default algorithms automatically selected by the MPI library based on the different message sizes and architecture of the system.

Figure 6 shows the AllReduce communication latency on the Pitzer system. The proposed Recursive-Doubling AllReduce with collective-level compression can achieve benefits from 512KB to 128MB compared to the basic Recursive-Doubling. For the Ring algorithm, the proposed compression design achieves benefits starting from 512KB, 1MB, and 2MB on 4, 8, and 16 GPUs respectively. For a lower rate number, ZFP compresses the floating point data to fewer bits, therefore such a configuration will achieve more communication benefits of the reduced data size. For smaller message ranges (e.g., 512KB to 2MB), the overheads of compression operations are more critical. Since Recursive-Doubling requires fewer data transmission and thus fewer compression operations, the RD+ZFP achieves better performance than the Ring+ZFP. For larger message ranges (e.g., 4MB to 128MB), the proposed Ring algorithm with compression archives better performance than Recursive-Doubling with compression, especially on more GPUs(e.g., 8, 16 GPUs).

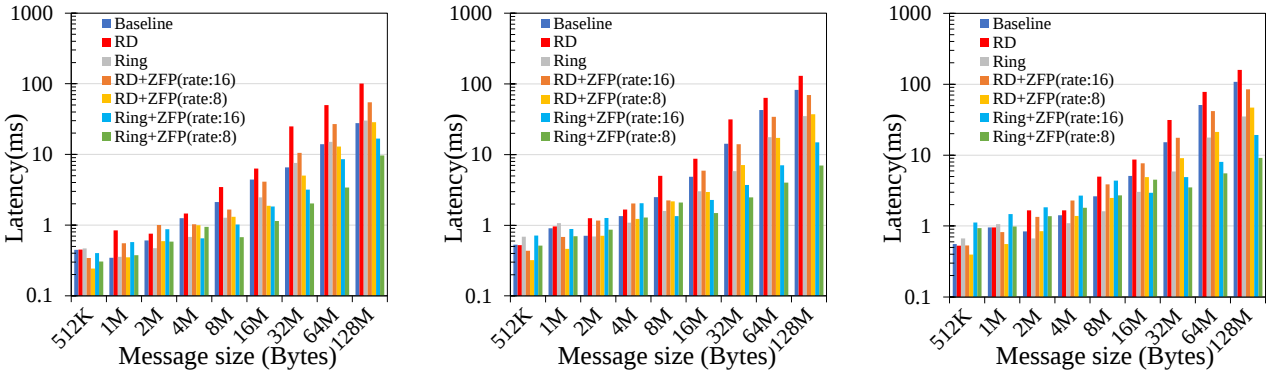
Compared to the basic RD algorithm, RD+ZFP reduces the communication latency on 4 GPUs by 17.5% (512KB) to 47.7% (32MB) with rate:16 and by 39.7% (512KB) to 70.6% (128MB) with rate:8. On 8 GPUs, the latency reduces by 12.0% (512KB) to 47.2% (32MB) with rate:16 and by 36.4% (512KB) to 70.8% (64MB) with rate:8. On 16 GPUs, the latency reduces by 30.1% (512KB) to 51.2% (64MB) with rate:16 and by 49.6% (512KB) to 73.1% (64MB) with rate:8. Compared to the basic Ring algorithm, the Ring+ZFP reduces the communication latency on 4 GPUs by 16.1% (512KB) to 63.7% (8MB) with rate:16 and by 26.8% (512KB) to 80.4% (64MB) with rate:8. On 8 GPUs, the latency reduces by 37.0% (2MB) to 58.8% (32MB) with rate:16 and by 50.3% (2MB) to 80.4% (32MB) with rate:8. On 16 GPUs, the latency reduces by 7.1% (2MB) to 61.0% (32MB) with rate:16 and by 15.8% (2MB) to 81.2% (64MB) with rate:8.

Compared to the baseline algorithm, RD+ZFP reduces the communication latency by 9.6% (512KB on 8 GPUs) to 55.7%



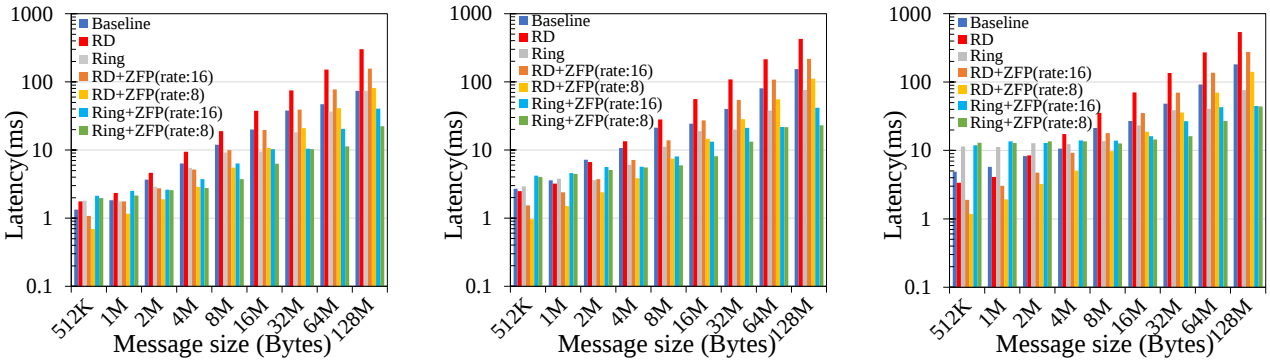
(a) Pitzer: 4 GPUs (2nodes, 2ppn) (b) Pitzer: 8 GPUs (4nodes, 2ppn) (c) Pitzer: 16 GPUs (8nodes, 2ppn)

Fig. 6: Latency of MPI_Allreduce on Pitzer system (V100 GPUs).



(a) MRI: 4 GPUs (2nodes, 2ppn) (b) MRI: 8 GPUs (4nodes, 2ppn) (c) MRI: 16 GPUs (8nodes, 2ppn)

Fig. 7: Latency of MPI_Allreduce on MRI system (A100 GPUs).

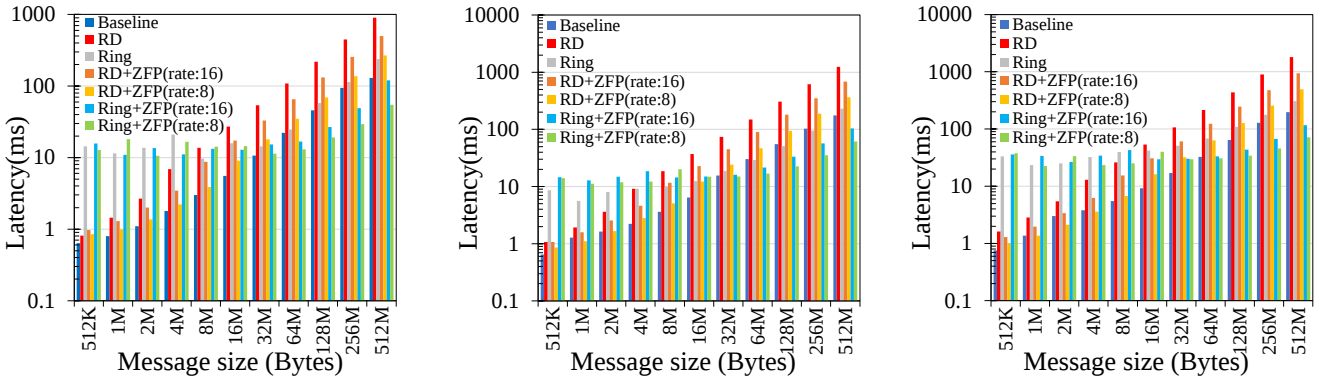


(a) Frontera: 16 GPUs (4nodes, 4ppn) (b) Frontera: 32 GPUs (8nodes, 4ppn) (c) Frontera: 64 GPUs (16nodes,4ppn)

Fig. 8: Latency of MPI_Allreduce on Frontera Liquid system (RTX5000 GPUs).

(64MB on 16 GPUs) with rate:16 and by 34.7% (512KB on 8 GPUs) to 75.5% (64MB on 16 GPUs) with rate:8. The Ring+ZFP reduces the latency by 13.3% (1MB on 8 GPUs) to 67.5% (64MB on 16 GPUs) with rate:16 and by 14.6% (1MB on 8 GPUs) to 85.3% (64MB on 16 GPUs) with rate:8.

Figure 7 shows the AllReduce communication latency on the MRI system. We observe similar trends with the proposed designs. The Recursive-Doubling AllReduce with the proposed compression design can achieve benefits for almost all message sizes from 512KB to 128MB compared to the basic Recursive-Doubling, except for some message sizes with a ZFP compression rate of 16. The Ring algorithm with the proposed compression design achieves benefits compared to the basic ring algorithm starting from 4MB, 8MB, and 16MB on 4, 8, and 16 GPUs respectively. The difference we observed on the MRI system is probably due to the faster HDR interconnect compared to the EDR interconnect on the Pitzer system. Similarly, we observe that Recursive-Doubling algorithm with compression achieves better performance than the Ring algorithm with compression for smaller message ranges (e.g., 512KB to 1MB) as we discussed in Figure 6. The Ring algorithm with compression is better for larger messages (e.g., 4MB to 128MB on 8GPUs, 8MB to 128MB on 16 GPUs, and 16M to 128MB on 32 GPUs).



(a) Lassen: 64 GPUs (16nodes, 4ppn) (b) Lassen: 128 GPUs (32nodes, 4ppn) (c) Lassen: 256 GPUs (64nodes,4ppn)

Fig. 9: Latency of MPI_Allreduce on Lassen system (V100 GPUs).

Compared to the basic RD algorithm, RD+ZFP reduces the communication latency on 4 GPUs by 23.7% (512KB) to 57.8% (32MB) with rate:16 and by 21.5% (2MB) to 74.0% (64MB) with rate:8. On 8 GPUs, the latency reduces by 7.7% (2MB) to 55.4% (32MB) with rate:16 and by 38.9% (512KB) to 77.4% (32MB) with rate:8. On 16 GPUs, the latency reduces by 14.2% (1MB) to 46.6% (128MB) with rate:16 and by 16.9% (4MB) to 72.7% (64MB) with rate:8. Compared to the basic Ring algorithm, the Ring+ZFP reduces the communication latency on 4 GPUs by 4.7% (4MB) to 58.1% (32MB) with rate:16 and by 34.8% (512KB) to 77.3% (64MB) with rate:8. On 8 GPUs, the latency reduces by 17.5% (1MB) to 60.1% (64MB) with rate:16 and by 24.9% (512KB) to 80.1% (128MB) with rate:8. On 16 GPUs, the latency reduces by 16.5% (32MB) to 54.7% (64MB) with rate:16 and by 7.4% (1MB) to 74.0% (128MB) with rate:8.

Compared to the baseline algorithm, the RD+ZFP reduces the latency by 6.3% (16MB on 4 GPUs) to 24.5% (1MB on 8 GPUs) with rate:16 and by 5.4% (8MB on 16 GPUs) to 59.7% (64MB on 8 GPUs) with rate:8. The Ring+ZFP reduces the latency by 9.8% (512KB on 4 GPUs) to 74.0% (128MB on 8 GPUs) with rate:16 and by 10.9% (16MB on 16 GPUs) to 82.6% (128MB on 8 GPUs) with rate:8.

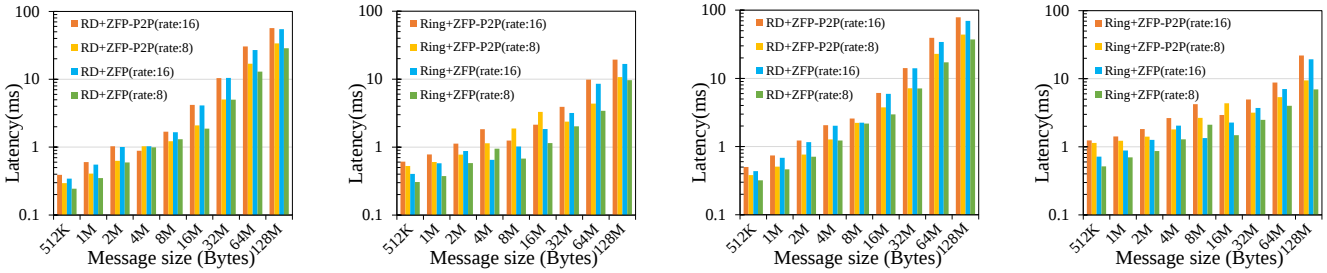
Figure 8 shows the AllReduce communication latency on the Frontera-Liquid system. The proposed Recursive-Doubling AllReduce design can achieve benefits for almost all message sizes compared to the basic Recursive-Doubling. The proposed Ring algorithm with compression achieves benefits starting from 2MB, 4MB, and 8MB on 16, 32, and 64 GPUs respectively, compared to the basic ring algorithm. Note that, for the ring algorithm, each data chunk is only 128KB for these message sizes. Similarly, the proposed Recursive-Doubling with compression achieves better performance in smaller ranges (e.g., 512KB to 4MB) as we discussed in Figure 6. The Ring algorithm with compression is better for larger messages (e.g., 8MB to 128MB).

Compared to the basic RD algorithm, RD+ZFP reduces the latency on 16 GPUs by 25.2% (1MB) to 48.7% (64MB) with rate:16 and by 50.8% (1MB) to 73.0% (128MB) with rate:8. On 32 GPUs, the latency reduces by 25.4% (1MB) to 51.3% (16MB) with rate:16 and by 53.2% (1MB) to 74.1% (64MB) with rate:8. On 64 GPUs, the latency reduces by 25.5% (1MB) to 50.0% (16MB) with rate:16 and by 52.9% (1MB) to 74.2% (64MB) with rate:8. Compared to the basic Ring algorithm, the Ring+ZFP reduces the latency on 16 GPUs by 10.0% (2MB) to 45.2% (128MB) with rate:16 and by 11.5% (2MB) to 69.7% (128MB) with rate:8. On 32 GPUs, the latency reduces by 6.1% (4MB) to 45.3% (128MB) with rate:16 and by 8.8% (4MB) to 69.6% (128MB) with rate:8. On 64 GPUs, the latency reduces by 30.6% (16MB) to 41.3% (128MB) with rate:16 and by 7.2% (8MB) to 58.2% (32MB) with rate:8.

Compared to the baseline algorithm, the RD+ZFP reduces the latency by 13.1% (4MB on 64 GPUs) to 61.2% (512KB on 64 GPUs) with rate:16 and by 12.9% (64MB on 16 GPUs) to 66.5% (1MB on 64 GPUs) with rate:8. The Ring+ZFP reduces the latency by 21.5% (2MB on 32 GPUs) to 75.4% (128MB on 64 GPUs) with rate:16 and by 29.3% (2MB on 32 GPUs) to 85.0% (128MB on 32 GPUs) rate:8.

Figure 9 shows the AllReduce communication latency on the Lassen system. The proposed Recursive-Doubling AllReduce design can achieve benefits for almost all message sizes (e.g., 1MB to 512MB) compared to the basic Recursive-Doubling. The proposed Ring algorithm with compression achieves benefits starting from 16MB, 32MB, and 16MB on 64, 128, and 256 GPUs respectively, compared to the basic ring algorithm. For these message sizes, each data chunk is only 256KB, 256KB, and 64KB respectively in the Ring algorithm. Similarly, the proposed Recursive-Doubling with compression achieves better performance in smaller ranges (e.g., 512KB to 8MB) as we discussed in Figure 6. The Ring algorithm with compression is better for larger messages (e.g., 16MB to 512MB).

Compared to the basic RD algorithm, RD+ZFP reduces the latency on 64 GPUs by 10.6% (1MB) to 43.1% (256MB) with rate:16 and by 31.4% (1MB) to 69.3% (256MB) with rate:8. On 128 GPUs, the latency reduces by 17.5% (1MB) to 44.8% (512MB) with rate:16 and by 20.0% (512KB) to 70.9% (512MB) with rate:8. On 256 GPUs, the latency reduces by 19.5% (512KB) to 47.6% (512MB) with rate:16 and by 37.0% (512KB) to 72.5% (512MB) with rate:8. Compared to the basic Ring algorithm, the Ring+ZFP reduces the latency on 64 GPUs by 19.0% (16MB) to 57.1% (128MB) with



(a) Recursive-Doubling (4 GPUs) (b) Ring (4 GPUs) (c) Recursive-Doubling (8 GPUs) (d) Ring (8 GPUs)

Fig. 10: Compare with Point-to-Point compression for Ring and Recursive-Doubling MPI_AllReduce on MRI system.

rate:16 and by 8.6% (16MB) to 74.3% (256MB) with rate:8. On 128 GPUs, the latency reduces by 14.4% (32MB) to 54.9% (512MB) with rate:16 and by 20.5% (32MB) to 73.5% (512MB) with rate:8. On 256 GPUs, the latency reduces by 29.6% (16MB) to 62.3% (512MB) with rate:16 and by 4.8% (16MB) to 76.6% (512MB) with rate:8.

Compared to the baseline algorithm, the Ring+ZFP can achieve benefits from 32MB to 512MB. Ring+ZFP reduces the latency by 7.1% (512MB on 64 GPUs) to 47.6% (256MB on 256 GPUs) with rate:16 and by 4.0% (32MB on 128 GPUs) to 65.7% (256MB on 128 GPUs) with rate:8. Since the baseline algorithm is much better than the basic RD algorithm for large numbers of GPUs, the RD+ZFP gets performance benefits on some small message sizes (e.g., 13.4% at 1MB on 128 GPUs, 29.5% at 2MB on 256 GPUs) with rate:8.

B. Compare with existing Point-to-Point Compression

In Figure 10 we compare the performance of the proposed collective-level compression designs for Recursive-Doubling and Ring AllReduce algorithms with naive point-to-point compression with a postfix "P2P". We turn on the point-to-point compression in the MPI library by the runtime parameters. As discussed earlier, our collective-level compression design addresses the limitation of using naive point-to-point compression. For the performance of the Recursive-Doubling AllReduce shown in Figures 10(a) and 10(c), the proposed RD+ZFP achieves benefits by 5.2% (2MB on 8 GPUs) to 13.1% (64MB on 8 GPUs) with rate:16 and by 5.6% (2MB on 4GPUs) to 24.8% (64MB on 8 GPUs) with rate:8 compared to RD+ZFP-P2P. For the Ring AllReduce shown in Figures 10(b) and 10(d), the proposed Ring+ZFP achieves benefits by 11.9% (128MB on 8 GPUs) to 64.5% (4MB on 4 GPUs) with rate:16 and by 10.0% (128MB on 4 GPUs) to 66.1% (16MB on 8 GPUs) with rate:8 compared to Ring+ZFP-P2P.

The proposed Ring AllReduce collective-level compression design achieves more performance benefits mainly because we cut down the redundant compression operations and archives more opportunities to overlap the compression/decompression kernels as discussed in the section III-C.

V. Application Results and Analysis

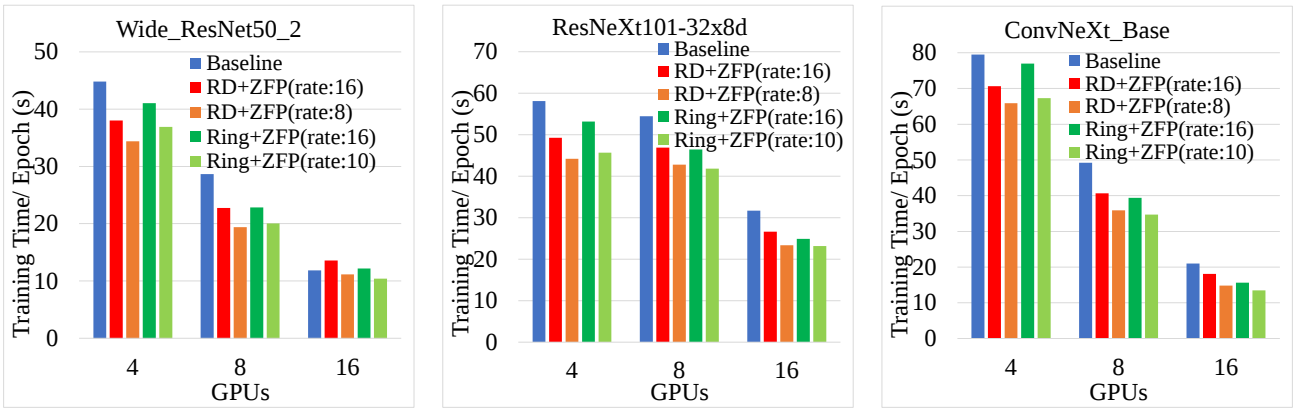
This section evaluates the proposed Ring and Recursive-Doubling AllReduce designs in the distributed DL training with PyTorch. We use PyTorch (v1.13) with MPI as the communication backend. We run PyTorch DDP [8] training of Wide_ResNet50_2 [37], ResNeXt101-32x8d [19], and ConvNeXt_Base [20] on CIFAR10 [21] dataset. We turn on DDP by adding a $DistributedDataParallel$ wrapper. We use a Batch Size(BS) of 128 and a Learning Rate(LR) of 0.001.

A. Training performance

Figure 11 shows the DDP training performance of three DNN models with PyTorch on 4 GPUs (2 nodes), 8 GPUs (4 nodes), and 16 GPUs (8 nodes) of the Pitzer system. We report the average training time per epoch for the baseline as well as the proposed designs with different compression rates. For model Wide_ResNet50_2, the proposed RD+ZFP can reduce the training time by 15.2% (4 GPUs) to 20.6% (8 GPUs) with rate:16 and by 5.9% (16 GPUs) to 32.3% (8 GPUs) with rate:8 compared to the baseline default algorithm. The proposed Ring+ZFP reduces the training time by 8.4% (4 GPUs) to 20.2% (8 GPUs) with rate:16 and by 12.0% (16 GPUs) to 30.1% (8 GPUs) with rate:10. For ResNeXt101-32x8d, the proposed RD+ZFP can reduce the training time by 13.9% (8 GPUs) to 15.8 (16 GPUs) with rate:16 and by 21.5% (8 GPUs) to 26.3% (16 GPUs) with rate:8. The proposed Ring+ZFP reduces the training time by 8.5% (4 GPUs) to 21.4% (16 GPUs) with rate:16 and by 21.4% (4 GPUs) to 26.8% (16 GPUs) with rate:10. For ConvNeXt_Base, the proposed RD+ZFP can reduce the training time by 11.2% (4 GPUs) to 17.4% (8 GPUs) with rate:16 and by 17.2% (4 GPUs) to 29.4% (16 GPUs) with rate:8. The proposed Ring+ZFP reduces the training time by 3.2% (4 GPUs) to 25.6% (16 GPUs) with rate:16 and by 15.4% (4 GPUs) to 35.7% (16 GPUs) with rate:10.

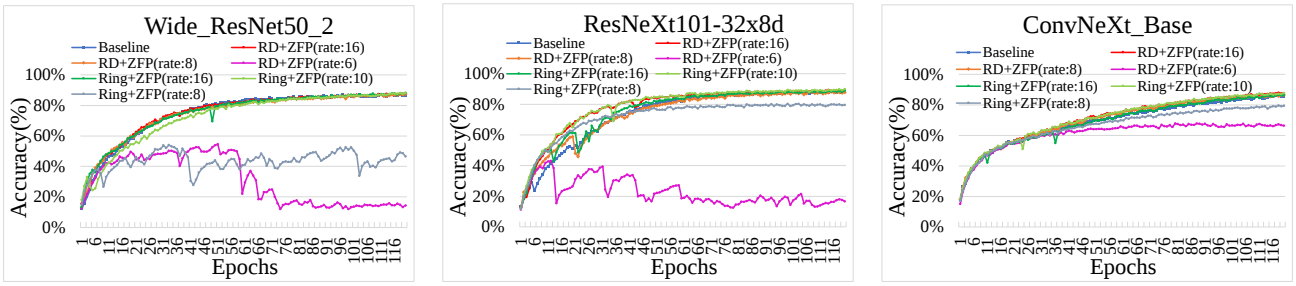
B. Training accuracy

We conduct experiments to study the impact of compression errors on the gradient tensors. we verify the training accuracy with different compression rates in the proposed Recursive-Doubling and Ring AllReduce designs for all three DNN models as shown in Figure 12. The accuracy is calculated by comparing the model predictions with the ground truth labels. In Figure 12(a) and 12(b), for both RD and Ring algorithms with compression, although we observe deviations at early epochs, the training of two models can converge to similar accuracy as the baseline with a close convergence speed. Figure 12(c) shows the training of ConvNeXt_Base with



(a) Wide_ResNet50_2 (BS=128, LR=0.001) (b) ResNeXt101-32x8d (BS=128, LR=0.001) (c) ConvNext_Base (BS=128, LR=0.001)

Fig. 11: DDP training performance with 4 nodes of 8 GPUs on Pitzer system.



(a) Wide_ResNet50_2 (b) ResNeXt101-32x8d (c) ConvNeXt_Base

Fig. 12: DDP training accuracy with 4 nodes of 8 GPUs on Pitzer system.

the proposed designs with compression can achieve similar training accuracy as the baseline from the very early epochs. We also observe that the RD+ZFP can accept slightly lower compression rates (e.g., 8) than rate 10 for Ring+ZFP. As previously analyzed, the Recursive-Doubling Allreduce does not split data into chunks, resulting in reduced compression operations and the potential accumulation of fewer errors at the same compression rate during data transmission. For even lower compression rates (e.g., rate:6 for RD+ZFP or rate:8 for Ring+ZFP), we start to observe an obvious accuracy drop probably due to the larger accumulated compression errors.

These results demonstrate that the compression approaches with appropriate compression rates can speed up distributed deep-learning training while maintaining training quality.

VI. Related Work

For the distributed Deep Learning training, state-of-the-art MPI libraries [4], [5] and NCCL (NVIDIA's Collective Communication Library) [12] Support AllReduce communication across multiple GPUs and multiple nodes. Advanced Allreduce schemes have been developed in the past few years, such as Baidu Ring algorithm [11], NCCL Ring-based algorithm [12] and double binary tree algorithm [13], Link-Efficient NVGroup algorithm [14], etc. Recent studies explored the Gradient Quantization such as QSGD [15] and Sparse AllReduce [16] based on the sparsity of the gradient for distributed DL training. The effectiveness of these solutions is contingent on their compatibility with the specific SGD algorithms.

Advanced GPUs from vendors like NVIDIA [38], AMD [39], and Intel [40] have greatly accelerated the computing taskes. The advanced GPU-based lossless compression algorithms (e.g., MPC [25], nvCOMP [31]) are much higher in computing throughput compared to the CPU-based algorithms. GPU-based lossy compression algorithms, such as cuSZ [41] and ZFP [26], can typically provide a high compression ratio and error-bounded performance in scientific applications, as demonstrated in recent study [33]. A recent work [24] has integrated MPC [25] and ZFP [26] into an MPI library to achieve high performance communication of large GPU data.

Recent research proposed optimization strategies for using compression in MPI_Alltoall [27] and MPI_Bcast communication [28]. Research [42] proposed collective operations with cuSZ [41] compression library for scientific HPC application.

VII. Conclusion

In this paper, we propose two collective-level compression schemes in the MPI library for efficient MPI_AllReduce communication of large GPU data, overcoming the limitations of naive point-to-point compression.

In the benchmark level evaluation, the proposed Recursive-Doubling and Ring AllReduce designs with compression demonstrate up to 75.5% and 85.3% reduced communication latency compared to the baseline. Compared to the existing point-to-point compression solution, the new Recursive-Doubling and Ring AllReduce achieve up to 24.8% and 66.1% reduced latency compared to the naive point-to-point compression, respectively. In the DDP training with PyTorch,

the proposed Recursive-Doubling and Ring AllReduce with compression reduce the training time by up to 32.3% and 35.7% respectively compared to the baseline while keeping a similar convergent training accuracy. Our approach, which operates at the communication middleware level, does not necessitate modifications to the applications.

As part of future work, we intend to design compression schemes for other parallel strategies to accelerate the distributed training of larger DL models.

REFERENCES

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[2] Martín Abadi and others, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[3] "MPI-4 Standard Document," https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[4] Open MPI, "Open MPI: Open Source High Performance Computing," https://www.open-mpi.org/, 2004, Accessed: March 19, 2024.

[5] Network-Based Computing Laboratory, "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE," http://mvapich.cse.ohio-state.edu/, 2001, Accessed: March 19, 2024.

[6] A. Jain, A. A. Awan, A. M. Aljuhani, J. M. Hashmi, Q. G. Anthony, H. Subramoni, D. K. Panda, R. Machiraju, and A. Parwani, "Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[7] A. Castelló, E. S. Quintana-Ortí, and J. Duato, "Accelerating distributed deep neural network training with pipelined mpi allreduce," *Cluster Computing*, vol. 24, no. 4, p. 3797–3813, dec 2021. [Online]. Available: https://doi.org/10.1007/s10586-021-03370-9

[8] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," 2020. [Online]. Available: https://arxiv.org/abs/2006.15704

[9] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[10] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," 2021.

[11] D. Amodei and Others, "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin," *CoRR*, vol. abs/1512.02595, 2015. [Online]. Available: http://arxiv.org/abs/1512.02595

[12] NVIDIA, "NCCL2," https://developer.nvidia.com/nccl, 2017, Accessed: March 19, 2024.

[13] Sylvain Jeaugey, "Massively Scale Your Deep Learning Training with NCCL 2.4," https://devblogs.nvidia.com/massively-scale-deep-learning-training-nccl-2-4/, Feb. 2019, Accessed: March 19, 2024.

[14] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. Panda, "NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.

[15] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," 2017.

[16] S. Li and T. Hoefler, "Near-optimal sparse allreduce for distributed deep learning," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 135–149. [Online]. Available: https://doi.org/10.1145/3503221.3508399

[17] "Pitzer system - Ohio Supercomputer Center," https://www.osc.edu/resources/technical_support/supercomputers/pitzer.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[19] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," *arXiv preprint arXiv:1611.05431*, 2016.

[20] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," 2022.

[21] A. Krizhevsky, "CIFAR10," https://www.cs.toronto.edu/ kriz/cifar.html, 2010, Accessed: March 19, 2024.

[22] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," in *42nd International Conference on Parallel Processing (ICPP), 2013*. IEEE, 2013, pp. 80–89.

[23] S. S. Sharkawi and G. A. Chochia, "Communication protocol optimization for enhanced GPU performance," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 9:1–9:9, 2020.

[24] Q. Zhou, C. Chu, N. S. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda, "Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters*," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 444–453.

[25] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, "MPC: A Massively Parallel Compression Algorithm for Scientific Data," in *IEEE Cluster Conference*, September 2015.

[26] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, 08 2014.

[27] Q. Zhou, P. Kousha, Q. Anthony, K. Shafie Khorassani, A. Shafi, H. Subramoni, and D. K. Panda, "Accelerating mpi all-to-all communication with online compression on modern gpu clusters," in *High Performance Computing*, A.-L. Varbanescu, A. Bhatele, P. Luszczek, and B. Marc, Eds. Cham: Springer International Publishing, 2022, pp. 3–25.

[28] Q. Zhou, Q. Anthony, A. Shafi, H. Subramoni, and D. K. D. Panda, "Accelerating broadcast communication with gpu compression for deep learning workloads," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2022, pp. 22–31.

[29] NVIDIA, "NVIDIA GPUDirect," https://developer.nvidia.com/gpudirect, 2011, Accessed: March 19, 2024.

[30] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, D. Laforenza, and S. Orlando, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 257–267.

[31] NVIDIA, "nvCOMP," https://github.com/NVIDIA/nvcomp, 2020, Accessed: March 19, 2024.

[32] S. Di and F. Cappello, "Fast Error-bounded Lossy HPC Data Compression with SZ," in *International Parallel and Distributed Processing Symposium(IPDPS)*, 2016.

[33] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. P. Ahrens, "Understanding gpu-based lossy compression for extreme-scale cosmological simulations," *ArXiv*, vol. abs/2004.00224, 2020.

[34] C.-H. Chu, K. Hamidouche, A. Venkatesh, A. A. Awan, and D. K. Panda, "Cuda kernel based collective reduction operations on large-scale gpu clusters," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 726–735.

[35] "Liquid Submerged System - Texas Advanced Computing Center, Frontera - Specifications," https://www.tacc.utexas.edu/systems/frontera.

[36] "Lassen - Livermore Computing center - Specifications," https://hpc.llnl.gov/hardware/platforms/lassen.

[37] S. Zagoruyko and N. Komodakis, "Wide residual networks," 2017.

[38] NVIDIA, "NVIDIA H100 Tensor Core GPU," https://www.nvidia.com/en-us/data-center/h100, 2022, Accessed: March 19, 2024.

[39] AMD, "MI200 Instinct Server Accelerators," https://www.amd.com/en/graphics/instinct-server-accelerators, 2021.

[40] Intel, "GAUDI2 Processor For Deep Learning Training And Inference Workloads," https://habana.ai/products/gaudi2/, 2021.

[41] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello, "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*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3–15. [Online]. Available: https://doi.org/10.1145/3410463.3414624

[42] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, Y. Huang, K. Raffenetti, H. Zhou, K. Zhao, Z. Chen, F. Cappello, Y. Guo, and R. Thakur, "gzccl: Compression-accelerated collective communication framework for gpu clusters," 2023.