CereSZ: Enabling and Scaling Error-bounded Lossy Compression on Cerebras CS-2

Shihui Song
shihui-song@uiowa.edu
The University of Iowa
Iowa City, IA, USA

Yafan Huang
yafan-huang@uiowa.edu
University of Iowa
Iowa City, IA, USA

Peng Jiang
peng-jiang@uiowa.edu
University of Iowa
Iowa City, IA, USA

Xiaodong Yu
xyu38@stevens.edu
Stevens Institute of Technology
Hoboken, NJ, USA

Weijian Zheng
wzheng@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Sheng Di*
sdi1@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Qinglei Cao
qinglei.cao@slu.edu
Saint Louis University
St. Louis, MO, USA

Yunhe Feng
Yunhe.Feng@unt.edu
University of North Texas
Denton, TX, USA

Zhen Xie
zxie3@binghamton.edu
Binghamton University
Binghamton, NY, USA

Franck Cappello
cappello@mcs.anl.gov
Argonne National Laboratory
Lemont, IL, USA

ABSTRACT

Today's scientific applications running on supercomputers produce large volumes of data, leading to critical data storage and communication challenges. To tackle the challenges, error-bounded lossy compression is commonly adopted since it can reduce data size drastically within a user-defined error threshold. Previous work has shown that compression techniques can significantly reduce the storage and I/O overhead while retaining good data quality. However, the existing compressors are mainly designed for CPU and GPU. As new AI chips are being incorporated into supercomputers and increasingly used for accelerating scientific computing, there is a growing demand for efficient data compression on the new architecture. In this paper, we propose an efficient lossy compressor, CereSZ, based on the Cerebras CS-2 system. The compression algorithm is mapped onto Cerebras using both data parallelism and pipeline parallelism. In order to achieve a balanced workload on each processing unit, we propose an algorithm to evenly distribute the pipeline stages. Our experiments with six scientific datasets demonstrate that CereSZ can achieve a throughput from 227.93 GB/s to 773.8 GB/s, 2.43x to 10.98x faster than existing GPU compressors.

*Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 Cass Avenue, Lemont, IL 60439, USA.

HPDC '24, June 3–7, 2024, Pisa, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0413-0/24/06...$15.00
https://doi.org/10.1145/3625549.3658691

CCS CONCEPTS

• **Theory of computation → Data compression**; • **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

Error-bounded Lossy Compression, AI-Optimized Architecture, Parallel Computing, Scientific Simulation, High-speed Compressor

ACM Reference Format:
Shihui Song, Yafan Huang, Peng Jiang, Xiaodong Yu, Weijian Zheng, Sheng Di*, Qinglei Cao, Yunhe Feng, Zhen Xie, and Franck Cappello. 2024. CereSZ: Enabling and Scaling Error-bounded Lossy Compression on Cerebras CS-2. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3625549.3658691

1 INTRODUCTION

Scientific computing applications often generate large volumes of data rapidly. For instance, Reverse Time Migration (RTM), which is an advanced seismic imaging technique for intricate subsurface structures, can generate as much as 2,800 TB of data from a 10x10x8 aperture in a single time-stamp [14, 33]. Linear Coherent Light Source (LCLS), which is a leading X-ray free-electron laser located at Stanford Linear Accelerator Center, generates raw photon snapshots at 250 GB/s [31]. Storing and processing such a great amount of data within a short time impose considerable challenges, even on high-performance computing systems [35, 37, 38, 45, 47].

To tackle this big data challenge, lossy compression techniques [8, 22, 26, 28, 39, 49] have been commonly used in scientific applications to reduce the data size while maintaining a user-specified error limit. Beyond the traditional compressors on CPU, accelerating data compression on heterogeneous processors, such as FPGA [41] and

GPU [13, 42, 48, 50], has become increasingly important for real-time compression tasks (e.g. reducing data stream intensity and accelerating program execution). For example, cuSZ [42] parallelizes quantization, prediction, and Huffman encoding on NVIDIA GPU, benefiting the runtime performance of large-scale cosmic simulation [17] and deep learning training systems [18].

In recent years, there has been a boom in AI chips to meet the high computation demand of AI workloads. Among the new AI accelerators, Cerebras has emerged as a powerful and flexible platform for general-purpose high-performance computing. Besides AI workloads, Cerebras has been increasingly adopted for scientific computing applications. For example, [32] implements 3D fast Fourier transforms of a 512^3 complex input array on the Cerebras CS-2 and achieves 959 microseconds, which is the largest ever parallelization for this problem size and the first implementation that breaks the millisecond barrier. Ltaief et al. [30] utilize the substantial memory bandwidth of the Cerebras CS-2 systems for seismic data processing. As massive data is generated by these applications on the AI chip, there is a growing demand for efficient data compression on the new architecture.

In this work, we present the first error-bounded lossy compressor called CereSZ for the Cerebras CS-2 system. Different from CPU and GPU, which are designed based on the control flow architecture (i.e. von Neumann architecture), the computing units on Cerebras construct a dataflow architecture. Each computing unit can only access data from a small local memory and its neighbors. The restricted memory access imposes new challenges in the design and implementation of compression algorithms.

First, the dataflow architecture does not allow the maintenance of any global status in the compression process, unless all computation is conducted on a single unit. To scale out the computation, we adopt a *block-wise* compression algorithm. The data is divided into blocks and compressed within each block independently. This allows the computation on different blocks to be naively mapped to different computing units on the Cerebras chip, without requiring any communication between the units. We also adopt a *stage-wise* design where the compression procedure is divided into multiple stages with each stage only processing the output of the previous stage. This allows the computation on each block to be mapped to a group of contiguous computing units and parallelized as a pipeline.

With the block-wise and stage-wise algorithm design, we can scale out the compression procedure to a 2D mesh of computing units on the Cerebras system. However, algorithm design alone is not enough to fully utilize the computing power of Cerebras. Since the performance of a pipeline is bottlenecked by the slowest stage, it is important to achieve a balanced workload among stages. The problem is nontrivial because different steps in the compression algorithm can vary significantly in execution time and the execution time can even be input-dependent. To deal with the issue, we propose to divide each step into finer-grained sub-stages and redistribute the sub-stages evenly into stage groups. Another issue is that the number of connected computing units on the chip is much larger than the number of pipeline stages. To utilize all the computing units, we must run multiple pipelines. However, it is nontrivial to run pipelines on the computing units where the input data are not initially available. We propose a technique that keeps forwarding data along the computing units to enable parallel execution of multiple pipelines.

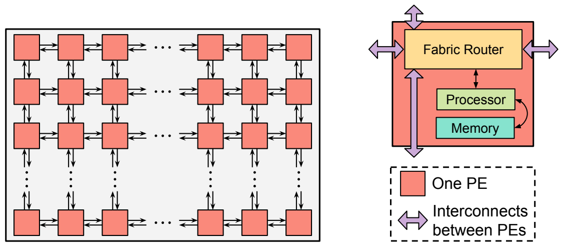
The contributions of this paper are summarized as follows.

- We proposed and implemented the first end-to-end data compressor on the Cerebras CS-2 system.
- We studied the parallelization of the compression/decompression algorithm on the dataflow architecture and proposed three strategies to scale out the computation.
- We theoretically and empirically show that our algorithm with the proposed parallelization strategies achieves linear speedups across the rows and columns of the 2D mesh of computing units on Cerebras.
- We conduct comprehensive experiments to evaluate CereSZ on 6 real-world scientific datasets. The results show that CereSZ can achieve on average 457.35 GB/s and 581.31 GB/s for compression and decompression throughput, respectively, which is 4.97 and 4.84 times faster compared with exiting GPU compressors without any losses of data quality.

2 BACKGROUND

In this section, we first give some background on the Cerebras AI chip and its programming model. Then, we introduce the terminologies related to error-bounded lossy compression.

2.1 Cerebras CS-2 System



(a) Cerebras WSE, where each node denotes one PE (processing element).
(b) Structure of a single PE on Cerebras WSE.

Figure 1: An illustration of the wafer-scale engine (WSE), which is the central processor of Cerebras CS-2 system.

Cerebras CS-2 is considered one of the most competitive systems for accelerating deep learning and scientific simulation tasks today [30, 32, 55]. The Cerebras AI chip is manufactured on a single *wafer-scale engine (WSE)*, which contains a two-dimensional mesh of 757×996 *processing elements (PEs)* with fast access to data in its local memory and nearest neighbors, as shown in Fig. 1(a). Each PE has its own program counter and thus can operate independently from other PEs.

Dataflow Architecture. The connected PEs on Cerebras WSE construct a *dataflow* architecture. As shown in Fig. 1(b), each PE has: 1) a *fabric router* which sends and receives data from its neighboring PEs, 2) a *processor*, and 3) a local *memory* that stores all the code and data used by the processor. The processor is connected to the fabric router via an internal link called the *RAMP*. The RAMP, along with the external links to the four neighbors (east, west, north, and south), are referred to as the five cardinal dataflow directions of one PE. A PE can exchange a 32-bit message, known as a *wavelet*, with its neighbors in one clock cycle. The programmer can assign one or more tasks to a PE, and a task will only be executed if its input data are available (usually received from its neighboring PEs). The programmer can also terminate a running task and initialize a new one from the set of tasks on a PE. Compared to the shared memory single-instruction multiple-data (SIMD) architecture (e.g., GPUs), Cerebras is more flexible in handling irregular control flows but has more restricted data flows.

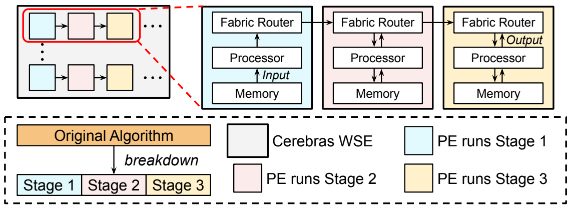


Figure 2: Pipeline parallelization on Cerebras WSE.

Parallel Processing on Cerebras. The dataflow architecture of Cerebras allows two types of parallel processing: MIMD and pipeline. If the computation is embarrassingly parallel, we can simply execute it in a multiple-instruction multiple-data (MIMD) fashion, typically by using PEs in different rows as the input data can be flowed into different rows simultaneously. However, because the computing and memory capacity of each PE is limited, we often need to split the computation into multiple stages and run different stages on consecutive PEs in a pipeline fashion. As shown in Fig. 2, we can map three computing stages into three consecutive PEs in the same row. After one PE finishes its computation, the intermediate data is transferred through the fabric router to the next PE. The data transfer between PEs can be conducted asynchronously with the computation.

Implementing a Pipeline. Cerebras provides an SDK that allows programmers to develop native code running on the PEs using a language called *CSL*. To implement a pipeline, the programmer needs to use CSL to explicitly define the communication between PEs. As mentioned earlier, there are five cardinal dataflow directions of one PE, including four neighbors and the local RAMP. To route a wavelet through the fabric, the programmer needs to define a logical channel called *color*. There are 24 colors available in total. For every color, the programmer needs to configure its input and output direction. Figure 3 shows an example of routing an array from one PE to its neighboring PE on the right and the pseudocode in Figure 4 shows the implementation of a pipeline on PE 2. We configure the two PEs with the same color and then define routing directions respectively. The left PE sends data from its RAMP to the east (right), and the right PE receives data from the west (left) and sends it to its RAMP. The data on the fabric can be accessed by the processor through *Data Structure Descriptors (DSDs)* (line 2 to 4 in Figure 4), which can be considered as pointers to the memory allocated in the fabric for holding the wavelets.

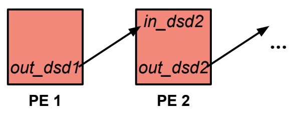


Figure 3: Routing an array from PE 1 to PE 2.

```
1  // Define the input dsd which receives N continuous elements
2  const in_dsd2 = @get_dsd(fabin_dsd, .{
3                      .fabric_color = color, .extent = N),
4                      .output_queue = @get_output_queue(1)});
5
6  // Receive data from the left PE and once it finishes, activate
7  // computeColor to run compute task
8  task read() void {
9      @mov32(input_compute_dsd, in_dsd2, .{ .async = true,
10            .activate = computeColor});}
11
12 // Run compute function, then activate readColor to run read task
13 // again to start receiving the next N elements
14 task compute() void {
15     compute(); @activate(readColor);}
16     // Send out_dsd2 to the right PE here}
17
18 // Bind two colors to their corresponding tasks
19 @bind_task(read, readColor);
20 @bind_task(compute, computeColor);
```

Figure 4: Routing an array from the left PE to the right PE.

Once the in_dsd2 is available on the right PE, its computeColor will be triggered, and the compute task bound with the color will be activated. The compute task then runs the compute function and activates the readColor to run the read task again and receive the next array. This data-triggering mechanism allows different PEs to run different stages simultaneously on different data, thus achieving pipeline parallelization.

2.2 Error-bounded Lossy Compression

Error-bounded lossy compression can reduce data size drastically while ensuring a small difference between the original and reconstructed data. The controllable data distortion can benefit visualization and post-hoc analysis and hence has been widely adopted in modern HPC scientific simulations and large-scale AI systems [7, 12, 14, 17, 18]. More formally, given a dataset $O = (e_1, e_2, ..., e_N)$ where e_i denotes the i-th element and N is the total number of elements, we can obtain the reconstructed data $R = (e'_1, e'_2, ..., e'_N)$ after compression and decompression. This reconstructed data should satisfy $|e_i - e'_i| \leq \epsilon, \forall i \in [1, N]$, where ϵ is a specified error bound.

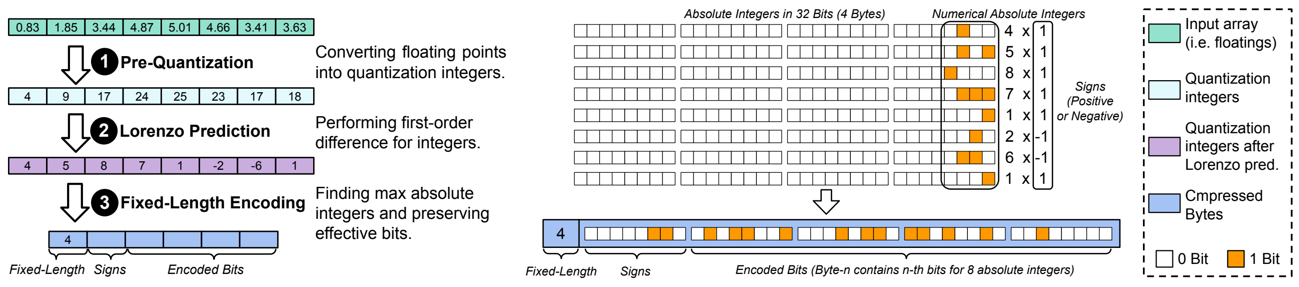
A lossy compression is often evaluated in three metrics.

1) *Throughput*: Compression/decompression throughput represents how much data can be processed during a certain time period;
2) *Compression ratio*: The compression ratio, calculated as the original data size divided by the compressed data size, indicates a compressor's efficiency in condensing information from the original data;
3) *Data quality*: This metric denotes how well a lossy compressor preserves the original data from a domain expert's perspective.

In this work, we consider both visualization quality and quantitative metrics, such as PSNR and SSIM, which will be further explained in Section 5.

3 CERESZ: COMPRESSION ALGORITHM

To better utilize the dataflow architecture in Cerebras WSE, we propose a stage-wise compression algorithm, CereSZ, that operates on blocks of data. Given an input dataset, which is generally represented as an array of floating-point numbers, we first divide it into


0.83
1.85
3.44
4.87
5.01
4.66
3.41
3.63
1 Pre-Quantization
Converting floating points into quantization integers.
4
9
17
24
25
23
17
18
2 Lorenzo Prediction
Performing first-order difference for integers.
4
5
8
7
1
-2
-6
1
3 Fixed-Length Encoding
Finding max absolute integers and preserving effective bits.
4
Fixed-Length
Signs
Encoded Bits
Absolute Integers in 32 Bits (4 Bytes)
Numerical Absolute Integers
4 x 1
5 x 1
8 x 1
7 x 1
1 x 1
2 x -1
6 x -1
1 x 1
Signs (Positive or Negative)
4
Fixed-Length
Signs
Encoded Bits (Byte-n contains n-th bits for 8 absolute integers)
Input array (i.e. floatings)
Quantization integers
Quantization integers after Lorenzo pred.
Cmpressed Bytes
0 Bit
1 Bit

(a) Three stages of CereSZ to compress this data block.
(b) Details of Fixed-Length Encoding (❸).
(c) Caption

Figure 5: A running example of CereSZ compression algorithm for one data block.

a set of data blocks, with each block comprising the same number of consecutive elements. As shown in Fig. 5, the data is divided into blocks of the same number of elements (eight in this example). Next, CereSZ compresses each data block independently through three major stages, including Pre-Quantization (❶), Lorenzo Prediction (❷), and Fixed-Length Encoding (❸):

Pre-Quantization (❶). This step converts floating-point numbers into quantized integers. Given a float number e_i and a user-defined error bound ϵ, the quantized integer p_i is calculated as the following formula:

$$p_i = \text{round}\left(\frac{e_i}{2\epsilon}\right). \tag{1}$$

The original number can be reconstructed as $p_i \cdot 2\epsilon$. This data quantization is the only step that introduces errors in the compression procedure. Since $\left|p_i - \frac{e_i}{2\epsilon}\right| \leq 0.5$, it is guaranteed that the error is bounded: $|p_i \cdot 2\epsilon - e_i| \leq \epsilon$. In the example in Fig. 5(a), suppose the error bound $\epsilon = 0.1$, the value 0.83 is converted to $\text{round}(0.83/0.02) = 4$. The error of the reconstructed data is $|4 \times 2 \times 0.01 - 0.83| = 0.03$, which is within the specified error bound.

Lorenzo Prediction (❷). The quantized data are then passed to a Lorenzo prediction procedure. Suppose a block of quantized data is denoted as $(p_1, p_2, ..., p_L)$, where p_i is i-th quantized number and L is the block size. The output of Lorenzo prediction is calculated as the first-order difference of the input: $(p_1, p_2 - p_1, ..., p_L - p_{L-1})$. Since many scientific datasets exhibit high smoothness [8, 13, 28, 50], this step can reduce the repeated bit patterns efficiently. As shown in Figure 5(a), the absolute value of the output is smaller than the input, thus requiring fewer bits to store the data. Note that beyond the first-order difference (i.e. 1D Lorenzo prediction) in CereSZ, there are higher dimensional Lorenzo prediction methods [39, 42] that consider more spatial information, which can lead to a higher compression ratio. Although CereSZ can support such prediction methods, in this work, we prioritize high throughput, making ❷ the preferred choice due to its lower computational requirements (utilizing just the preceding data point) and its facilitation of coalescing memory access, as evidenced in prior studies [13, 42, 48, 50].

Fixed-Length Encoding (❸). Finally, we store the output of Lorenzo prediction (which are small integers) into a byte stream using as few bits as possible. We adopt a fixed-length encoding technique for this task. It computes the absolute values of all numbers in a block. The number of effective bits in the maximum absolute value is enough for accommodating the absolute value of any number in the block. We need one more bit to store the sign (i.e. positive or negative) for each number. In Figure 5(b), the maximum absolute value in the block is 8, which can be stored in four bits. Thus, the "fixed-length" of this block is four, and we can store every element in four bits and all the eight elements with four bytes. All the signs of the eight numbers can be compacted into one byte. In this example, the original data block has 32 bytes (8 single-precision floating data points), and the encoding procedure compresses it to 6 bytes, achieving a 5.33 compression ratio.

Decompression Steps. During the decompression phase, CereSZ follows the same block-wise design and performs the above three steps in reverse order. Given the compressed bytes for one data block, the first byte records the "fixed-length" information. If this length is f and the block size is L, then the subsequent $L/8$ and the last $f \times L/8$ bytes can be interpreted as signs and encoded bits, respectively. Based on this, CereSZ can decode the compressed byte array into L integers. Then, CereSZ reverses the Lorenzo prediction and generates quantization integers. This step can be formulated as a sequential prefix sum task within each data block. Finally, each quantization integer p_i can be restored into its corresponding floating value by $e'_i = p_i \cdot 2\epsilon$, where ϵ denotes error bound and e'_i denotes the reconstructed data of e_i while satisfying the error bound $|e_i - e'_i| \leq \epsilon$. In general, the decompression process in CereSZ bypasses the need to identify the maximum quantization integer due to the pre-known "fixed-length", resulting in fewer computations and thereby leading to higher throughput.

Rationale in CereSZ Algorithm Designs. (1) Prediction Strategy Selection: Besides Lorenzo Prediction adopted in CereSZ, there are other prediction methods such as spline interpolation [52] and block-wise linear regression [24, 54]. While the former can cause strided memory access pattern, the latter requires extensive computations to retrieve the regression coefficients, hence negatively impacting throughput. For instance, obtaining 10 coefficients via quadratic regression on a single data block involves floating-point computations equivalent to a 10x10 matrix multiplication. (2) Lossless Encoding Selection: In CereSZ, we select fixed-length encoding rather than other variable-length encoding methods such as Huffman encoding [36, 43, 46], for which reasons are two-fold. First, building a Huffman encoding tree is expensive for parallel algorithms and violates the high-throughput design in CereSZ. Second, in fixed-length encoding, the compressed data length of each block can be calculated once "fixed-length" is known. Since the unique dataflow design in Cerebras WSE preserves the block processing order, CereSZ avoids synchronization overheads, which is a device-level scan while concatenating compressed bytes across data blocks. This benefits runtime throughput in turn compared with existing compressors such as GPULZ [51], FZ-GPU [50], and cuSZp [13].

4 MAPPING CERESZ ONTO CEREBRAS WSE

In this section, we explain how our proposed compression algorithm CereSZ is parallelized and executed on Cerebras WSE. An overview is shown in Fig. 6.

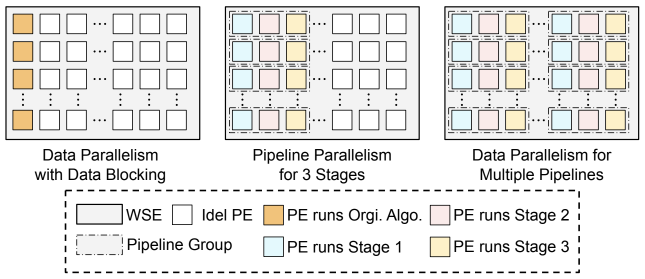

Data Parallelism with Data Blocking
Pipeline Parallelism for 3 Stages
Data Parallelism for Multiple Pipelines
WSE
Idel PE
PE runs Orgi. Algo.
PE runs Stage 2
Pipeline Group
PE runs Stage 1
PE runs Stage 3

Figure 6: Three parallelization strategies for mapping our compression algorithm to 2D mesh of PEs.

4.1 Data Parallelism for Different Blocks

Since the computations in different blocks are independent, it is straightforward to assign the processing of different blocks to different PE rows. Fig. 6 (left) illustrates this parallelization paradigm with an assumption that the entire compression procedure is executed on the first PE of each row.

Linear Speedup Across Rows. As different rows of PEs run independently without any communication, it is expected that increasing the number of rows will result in a linear speedup. To verify this point, we run the compression algorithm on the temperature field of the NYX dataset [2] using the first PE of each row. We set the data block size to 32 and keep flowing data blocks to each row. Fig. 7 shows the throughput (MB/s) of compressing the entire data set with different numbers of PE rows. The execution time is measured as the clock cycles needed for the last PE to finish processing its data. We can see the throughput indeed increases linearly w.r.t the number of PE rows. A similar pattern is observed in other datasets.

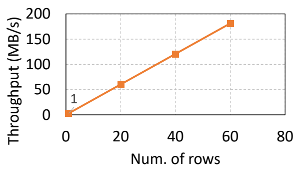


Figure 7: Throughput with different numbers of PE rows.

4.2 Pipeline Parallelism for Different Stages

To further improve the performance, we want to utilize more columns of PEs. This can be achieved by dividing the compression procedure into multiple stages and mapping different stages to consecutive PE columns. Fig. 6 (middle) illustrates the idea by mapping the three compression steps (quantization, prediction, and encoding) onto the first three PEs of each row.

As a PE can only start execution when its input data are available, the pipeline execution will be bottlenecked by the PE with the longest job. Thus, it is important to achieve a balanced workload distribution among PEs. To demonstrate the problem, we profile the execution of quantization, prediction, and encoding on the first three PEs. Table 1 shows the execution time (in clock cycles) of the three steps for one data block. The experiment is conducted on the CESM-ATM [19], HACC [9], and QMCPack [20] datasets, and the execution time is obtained by the maximum execution cycles across all data blocks within each dataset. We run the experiments 10 times and report the average execution time and cycles. We can see that the quantization and prediction steps need a stable number of cycles and the prediction needs much fewer cycles than quantization and encoding. This is because quantization and prediction involve a fixed number of arithmetic operations and the subtraction in the prediction step is much faster than the division operation in quantization. Encoding is much slower than quantization and prediction, and its performance is dependent on the number of effective bits in the input. Considering that quantization and encoding have multiple operations, we can divide each of them into sub-stages to achieve a more balanced workload.

Dataset	Pre-Quant.	Loren. Pred.	FL Encd.
CESM-ATM	6051	975	37124
HACC	6101	975	29181
QMCPack	6111	975	27188

Table 1: Execution cycles for three steps.

Dividing Pre-Quantization into Two Sub-stages. In our real implementation of the quantization in Formula (1), the division operation is implemented as a multiplication with the reciprocal of 2ϵ and the round operation is implemented as an addition with 0.5 followed by a floor operation. The breakdown cycles of the multiplication and addition operations are shown in Table 2. As expected, the execution times of the two operations are consistent across different datasets. Multiplication takes approximately 80% of the quantization time.

Dataset	Pre-Quant.	Multiplication	Addition
CESM-ATM	6051	5078	1033
HACC	6101	5081	1038
QMCPack	6111	5063	1049

Table 2: Breakdown cycles for Pre-Quantization.

Dividing Fixed-Length Encoding. Fixed-length encoding can be decomposed into four sub-stages: *Sign*, *Max*, *GetLength*, and *Bit-shuffle*. The *Sign* stage involves storing signs and calculating the absolute values of numbers in a data block. The *Max* stage obtains the maximum of the absolute values. The *GetLength* stage obtains the number of effective bits of the maximum value (i.e., the encoding length). The *Bit-shuffle* stage transforms effective bits of integers into a set of aligned bytes according to the encoding length. Among the four sub-stages, *Sign*, *Max*, and *GetLength* perform fixed numbers of operations and thus have stable execution time. This is validated by our profiling results in Table 3.

Algorithm 1: Evenly distributing n sub-stages across m PEs

```
Input: The stages: $s_1, s_2, ..., s_n$; Total cycles of all stages: $C$; Number of PEs: $m$;
Output: Stage group assigned to PEs: $G_1, G_2, ..., G_m$
1 Initialize $G_1 = \{\}, G_2 = \{\}, ..., G_m = \{\}$
2 for each stage group $G_i$ in $\{G_1, G_2, ..., G_{m-1}\}$ do
3     while The sum of runtime of the stages in $G_j < \frac{C}{m}$ do
4         move the next $s_j$ to $G_i$
5 $G_m = \{s_1, s_2, ..., s_n\} - (G1 \cup G2 \cup ... \cup G_{m-1})$
```

Dataset	FL Encd.	Sign	Max	GetLength	Bit-shuffle
CESM-ATM	37124	1044	1037	1386	33609
HACC	29181	1041	1032	1370	25675
QMCPack	27188	1048	1041	1385	23694

Table 3: Breakdown cycles for Fixed-Length Encoding.

The *Bit-shuffle* stage, however, requires a varying runtime for different datasets. This variation is attributed to their different encoding lengths (17, 13, and 12 for CESM-ATM, HACC, and QMCPack respectively). Figure 8 gives an example of Bit-shuffle with 8 integers from I_0 to I_7. For the k−th effective bit, we rearrange this bit of all integers into Byte k. This variation suggests a uniform encoding overhead per effective bit, as evidenced by the approximate equivalences $33609/17 \approx 25675/13 \approx 23694/12$. Since the Bit-shuffle of each effective bit is independent of each other, this stage can be further segmented into *1-bit Shuffle*.

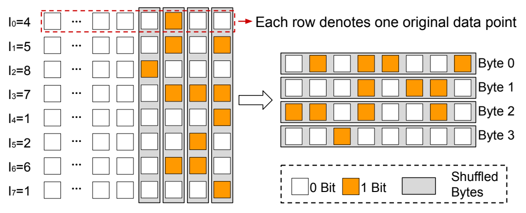


Figure 8: Illustrating Bit-shuffle using block from Fig. 5.

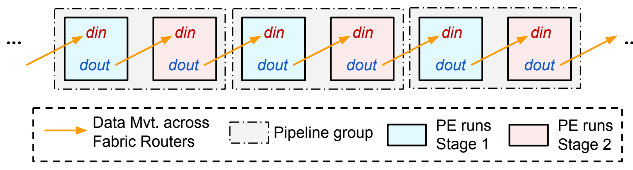
The division of the steps in decompression follows a similar pattern. The reverse of Bit-shuffle which breaks down the rearranged Bytes one by one can be subdivided into operations of each Byte. Reversing Lorenzo Prediction involving calculating the prefix sum of all integers, cannot be further divided. Similarly, the reverse Pre-Quantization step, comprised solely of multiplication operations, remains indivisible. Having segmented the three reverse operations, Algorithm 3 can also be effectively employed to distribute these steps uniformly across PEs.

Distributing Sub-stages to PEs. We now describe a simple greedy algorithm that evenly distributes m sub-stages across n PEs. As shown in Alg. 1, we start with m empty stage groups $G_1, G_2, \ldots, G_m$. Suppose the total execution time of all sub-stages is C. We iterate over all the sub-stages from s_1 to s_n and keep adding s_i to the current group until the group exceeds an execution time of C/m. When the execution time limit is reached, we stop adding sub-stages to the group and move to the next group.

Notice that the Multiplication step has the longest runtime, so it bottlenecks the performance of the Pipeline. Suppose the runtime of Multiplication is t_1, the maximum feasible length for the Pipeline can be obtained by $\lfloor C/t_1 \rfloor$. Using a Pipeline with a longer length than this cannot achieve better performance. Since the length is determined by the total execution time, the distribution strategy should be the same for two datasets that have the same fixed length. In our experiments, 5% of the data points are randomly sampled to approximate the fixed length for various configurations, allowing for an estimation of the total execution time C.

4.3 Data Parallelism for Different Pipelines

Because Cerebras has many more PE columns than the number of stages of our compression algorithm, it will be beneficial to further divide the data and use more PE columns to process multiple compression pipelines in each row, as shown in Fig. 6 (right). Without loss of generality, we assume that the input data is generated on the first PE of each row. To start a pipeline from a certain column, we need to send the data from the first PE to that column.



(a) Passing data with 2-length pipeline.

```
// Define the input dsd which receives N continuous elements
const din = @get_dsd(fabin_dsd, .{
                .fabric_color = recvColor, .extent = N),
                .output_queue = @get_output_queue(1)});

// Define the output dsd which sends N continuous elements
const dout = @get_dsd(fabout_dsd, .{
                .fabric_color = sendColor, .extent = N),
                .output_queue = @get_output_queue(0)});

// Define the input and its dsd
var input: [N]i32;
var data = @get_dsd(mem1d_dsd, .{.tensor_access = |i|{N}->input[i]});

// Define the number of data blocks received
const nblock : u32 = 0;

task relay() void {
    // Receive the input dsd and activate computeColor once the
    // current PE receives it own data block
    if (nblock == (total_cols-cur_col)/pipeline_length)){
        @mov32(data, din, .{.async = true, .activate =
                computeColor}); nblocks = 0;}}
    // Pass the data blocks for right PEs and activate relayColor again
    else{
        @mov32(dout, din, .{.async = true});
        nblocks += 1;
        @activate(relayColor);}

task compute() void {
    // Activate relayColor to run relay task again
    @activate(relayColor);}
    // Execute substages assigned to the PE
    // Send results to next PE in the pipeline}

// Bind two colors to their corresponding tasks
@bind_task(relay, relayColor);
@bind_task(compute, computeColor);
```

(b) Pseudocode runs on the first PE of each pipeline.

Figure 9: Illustrating data preparation for parallel pipelines.

Data Preparation for Parallel Pipelines. Fig. 9(a) shows how the data is passed through multiple pipelines in a row. Since a PE can only communicate with the PEs next to it, the data must be relayed by every PE along the path. Fig. 9(b) gives a pseudocode that runs on the first PE of each pipeline for relaying the input data. We define an input DSD (*din*) and an output DSD (*dout*) on each PE (line 2 to 9). The *relay* task keeps receiving data in *din* and passing it to *dout* (line 26 to 28). A counter *nblocks* is used for recording the number of data blocks relayed through the PE (line 16). Suppose the total number of PE columns is TC. Since the number of data blocks required by the PEs following PE-i is $(TC-i)/pipeline\_length$, once *nblocks* reaches this number, we move the data to the local memory of PE-i and activate the computation on the PE (line 21 to 23). Before executing the sub-stages, the *compute* task activates the *relay* task so that the PE can keep forwarding data to the PEs on the right. The code running on the remaining PEs of a pipeline is similar, except that they need another set of input/output descriptors to receive/send intermediate data.

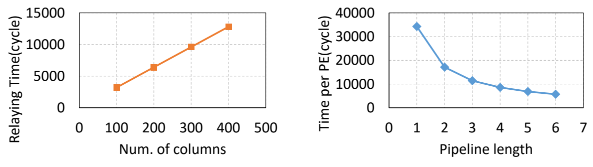
According to the code in Fig. 9(b), the execution time of a PE comprises two parts: the time for relaying data needed by the PEs on its right and the time for executing the sub-stages on its data block. Consider the PE-i. It needs to wait for $(TC-i)/pipeline\_length$ data blocks to be passed through it between every two executions of the *compute* task. The waiting time equals the time that PE-$(i-1)$ needs for $(TC-i+1)/pipeline\_length$ data blocks to be passed through it plus the time to send a block from PE-$(i-1)$ to PE-i. By induction, we know that the data relaying time on each PE is

$$TC \cdot C_1 \tag{2}$$

where C_1 is the number of cycles needed for sending a data block from one PE to the next PE. We profile the data relaying time with varying numbers of columns on QMCPack and report. The result in Fig. 10(a) shows a linear correlation between the number of rows and the relaying time on each PE, which verifies (2). Suppose the total number of cycles for the entire compression procedure is C and the cycles needed for sending the intermediate data of one block to the next PE is C_2. Assuming that the computation can be evenly distributed across PEs, the computation time on each PE can be represented as

$$\frac{C}{pipeline\_length} + pipeline\_length \cdot C_2. \tag{3}$$

It is worth noting that C_2 is different from C_1. While C_1 only involves routing a data block on the fabric, C_2 includes the time of moving data from local memory to the fabric and routing data on the fabric. The idea of (3) can be verified in Fig. 10(b), which is conducted on QMCPack and shows an inversely proportional between the execution time per PE and pipeline length.



(a) Relaying time on each PE.

(b) Execution time on each PE.

Figure 10: Profiling the relaying time and execution time on each PE.

4.4 Complexity Analysis

Given a fixed number of PE columns, the number of parallel pipelines in a row is $TC/pipeline\_length$. When processing the same dataset, the number of execution rounds in each pipeline is proportional to $pipeline\_length/TC$. Multiplying it with the sum of Formula (2) and (3), we can obtain the total execution time as

$$O\left(\frac{C}{TC} + pipeline\_length \cdot C_1 + pipeline\_length^2 \cdot C_2\right). \tag{4}$$

The first term in the time complexity suggests that, by combining our second and third parallelization strategies, we achieve an almost linear speedup w.r.t the total number of PE columns. The second term suggests that a small overhead proportional to the pipeline length is incurred for propagating data through the PEs to multiple pipelines. Another overhead quadratic to the pipeline length is incurred for flowing the intermediate data through the PEs within each pipeline. **Selection of Pipeline Length.** According to Formula (4), the optimal performance is achieved with $pipeline\_length = 1$. That is, we should execute the entire compression procedure on a single PE for each data block. However, there are two assumptions here: 1) the data is generated fast enough to saturate all the TC pipelines in a row and 2) the local memory is large enough to hold the intermediate data of the entire compression procedure. If either of the assumptions does not hold, we need to split the computation and use a longer pipeline. The best configuration of pipeline length depends on the data generation rate and the memory consumption of the compression algorithm. Since the number of sub-stages in the compression procedure is limited, usually less than 10, the optimal configuration can be easily obtained by tuning.

5 EVALUATIONS

In this section, we first present experimental setups. Then we analyze CereSZ from three perspectives, including throughput, compression ratio, and data quality, based on Section 2.2.

5.1 Experimental Setups

5.1.1 Platform and Settings for CereSZ. We evaluate CereSZ on the Cerebras CS-2 system, comprising a grid of 757×996 processing elements (PEs), with a clock frequency of 850MHz. We can use up to 750×994 PEs for implementation, since the rest PEs are used for routing data on and off the WSE. Each PE has 48 KB of SRAM and there is no global memory for the whole WSE. We use the hardware cycle counters at each PE to measure runtime and use counters divided by clock frequency (i.e. 850M) to obtain the runtime in seconds. In this section, we present the maximum cycles during our throughput evaluation.

In the Cerebras CS-2 system, the minimum data transfer units are 16-bit and 32-bit, necessitating a block size divisible by 16. We choose a block size of 32 in our implementation, as it yields the highest compression ratio among the options considered [13]. To comply with data transfer requirements, we use 32 bits (i.e. 4 bytes), rather than 8, for storing each block's fixed-length. While this approach does incur a slight compression ratio penalty, our evaluations demonstrate it is negligible for most cases and can significantly benefit throughput. Our code is implemented using Cerebras SDK of version 0.8.0.

5.1.2 Dataset. We evaluate CereSZ on six real-world HPC datasets from SDRBench [53]. These datasets are from various scientific domains, including seismic imaging [5, 14], climate simulation [1, 19], quantum computing [20], and cosmic simulation [4, 9, 10]. Dataset details can be found in Table 4. These datasets are commonly utilized for assessing lossy compressors in the data reduction community [13, 16, 22, 34, 48, 50].

Dataset	No. of Fields	Dim. per Field	Domain
CESM-ATM [19]	79	1,800x3,600	Climate Simulation
Hurricane [1]	13	500x500x100	Weather Simulation
QMCPack [20]	2	33120x69x69	Quantum Monte Carlo
NYX [4]	6	512x512x512	Cosmic Simulation
RTM [5, 14]	36	449x449x235	Seismic Imaging
HACC [10]	6	280,953,867	Cosmic Simulation

Table 4: Datasets for evaluating CereSZ.

5.1.3 Baseline Compressors and Settings. We compare CereSZ with several state-of-the-art CPU and GPU lossy compressors, including SZ [26], cuSZ [42], SZp [21], and cuSZp [13]. All of the four baseline compressors are error-bounded and prediction-based. For CPU SZ, we employ its latest version SZ3 [26], which strategically fine-tunes prediction methods. In the meantime, cuSZ is a GPU compressor that is based on prediction and Huffman encoding [42]. cuSZp is another GPU compressor with a kernel fusion design, whereas SZp has a similar compression algorithm and is paralleled by OpenMP on CPU. For CPU compressors, we evaluate them on one AMD EPYC 7742 Processor (2.25GHz, 64C 128T). For GPU compressors, we compile them by CUDA 11.2.0 and test them on one NVIDIA A100 GPU (108 SMs, 40 GB). These devices are provided by Swing cluster from Argonne National Laboratory. All baseline compressors and CereSZ are assessed using a value-range-based relative (REL) error bound, to accommodate the varying values across different HPC datasets. Assuming the value range of a dataset is r, REL λ indicates the difference between original and reconstructed data for all data points should be controlled within λr.

5.1.4 Evaluation Metrics. In addition to throughput, which is the primary focus of this study, both the compression ratio and data quality are crucial for assessing a lossy compressor, as previously mentioned in Section 2.2. As such, we formalize all related metrics as follows.

- **Throughput**: Throughput, measured in gigabytes per second (GB/s), is defined as the amount of data that a compressor per unit of time can process. It can be computed by $Size_{\text{ori}}/T$, where $Size_{\text{ori}}$ denotes the original data size and T denotes execution time (either for compression or decompression). This is the most important metric for compressors that are executed on heterogeneous processors [13, 27, 41, 42, 50].
- **Compression Ratio**: The compression ratio is determined by the formula $Size_{ori}/Size_{cmp}$, where $Size_{\text{cmp}}$ is the size of the compressed data. A higher ratio implies that the compressor is more effective in condensing information from the original dataset. This is a general metric for data reduction works [6, 8, 23, 25, 28].
- **Visualization**: We visualize the HPC dataset for both original and reconstructed ones to present the impacts of errors introduced by CereSZ.
- **PSNR** (Peak Signal-to-Noise Ratio) (in dB) is a log scale quantitative metric used to measure the noise and loss of reconstructed data after compression [11, 40]. A higher PSNR value typically indicates better reconstruction quality.
- **SSIM** (Structural Similarity Index Measure) is another quantitative metric for data quality [44]. It compares the structural similarity between the original and reconstructed data while considering luminance, contrast, and structural information. SSIM values range within $[0, 1]$, and the closer to 1 the better.

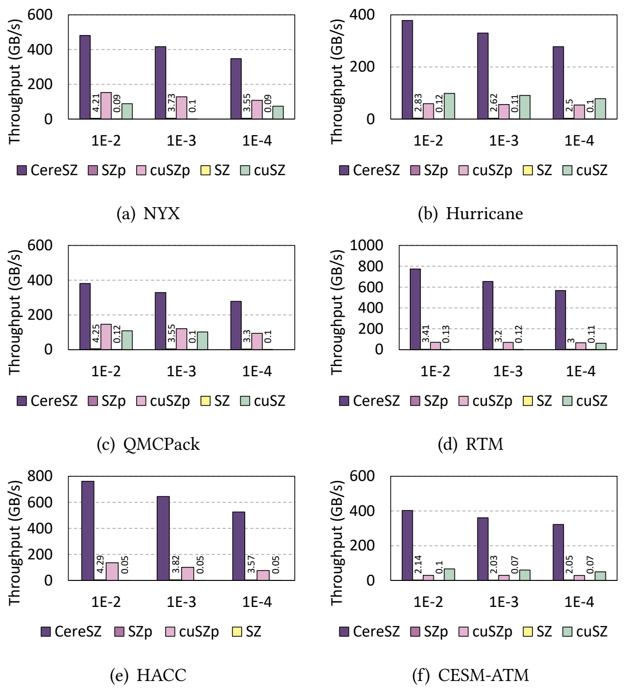
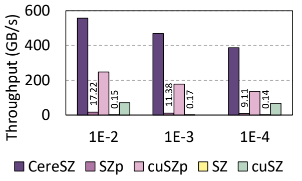


Figure 11: Compression Throughput (GB/s).

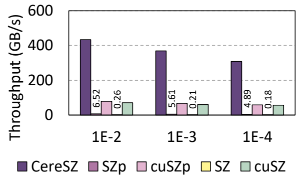
5.2 Throughput

Overall Throughput. We evaluate the throughput, which is the main target in this work, of CereSZ in this section. For each dataset, we report the average throughput across all fields. And for each compressor, we use REL 1E-2, 1E-3, and 1E-4 for a comprehensive evaluation. Note that we use a pipeline length of 1 and run CereSZ on 512×512 PEs.

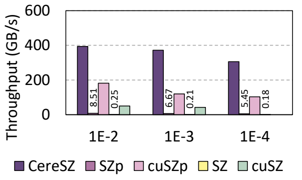
Fig. 11 and Fig. 12 present the throughput for compression and decompression, respectively. We can observe that CereSZ significantly outperforms other compressors among all datasets and settings, due to the high utilization of dataflow architecture and parallelism design. On average, CereSZ can achieve 457.35 GB/s and 581.31 GB/s for compression and decompression throughput, which is 4.9 and 4.8 times faster compared with cuSZp. Specifically, compression throughput in CereSZ is dataset-specific, ranging from 378.21 GB/s in Hurricane to 773.8 GB/s in RTM with error bound REL 1E-2. While setting the error bound as REL 1E-3, this range is from 328.9 GB/s in Hurricane to 654.63 GB/s in RTM. In REL 1E-4, where data quality is higher preserved, CereSZ still achieves at least 277.93 GB/s



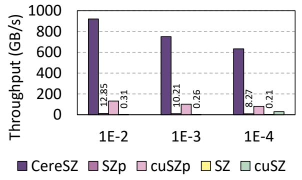
(a) NYX



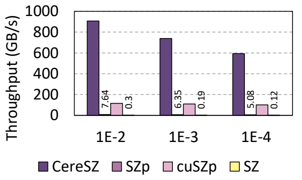
(b) Hurricane



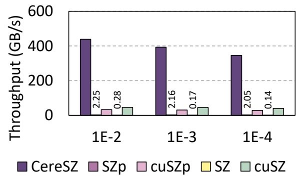
(c) QMCPack



(d) RTM



(e) HACC



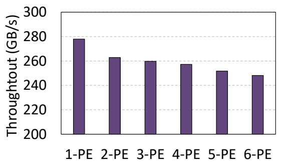
(f) CESM-ATM

Figure 12: Decompression Throughput (GB/s).

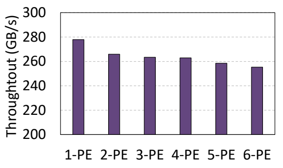
compression throughput (on QMCPack). For decompression, our proposed CereSZ can even achieve up to 920.67 GB/s throughput on RTM. Beyond there are fewer computations in decompression stages (as stated in Section 3), the key reason is that CereSZ scales the computation by data parallelism for different data blocks, pipeline parallelism for computation operations, and data parallelism for pipelines, which better utilizes the PEs on the WSE.

Another important observation is that compression and decompression throughput slightly reduce as the error bound decreases (e.g. from REL 1E-2 to REL 1E-3). For example, in NYX, the compression throughput exhibits 481.37 GB/s, 416.45 GB/s, and 347.51 GB/s in REL 1E-2, REL 1E-3, and REL 1E-4. The reason can be explained below. CereSZ processes original datasets in block granularity, and a larger error-bound can create more zero blocks (i.e. a data block with all zero elements) in the reconstructed data. While processing zero blocks, CereSZ only needs to store a byte flag, avoiding computations including fixed-length encoding and bit-shuffle, hence resulting in higher runtime throughput. This also explains similar trends in cuSZp and SZp, due to the same lossless encoding method.

Throughput of Pipeline with Different Lengths. To understand the impact of pipeline length selection, we run different pipeline implementations on QMCPack and Hurricane as an example. Fig. 13 shows the compression throughput of pipelines with different lengths, where n-PE denotes the pipeline with length n. We can observe that conducting the complete compression process exclusively on a single PE results in the highest throughput. This outcome suggests that utilizing a solitary PE represents the most advantageous selection for maximizing efficiency. Conversely, employing a more extended pipeline results in a reduced throughput. This discrepancy



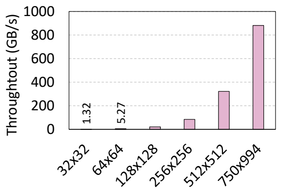
(a) Throughput on QMCPack with error bound 1E-4.



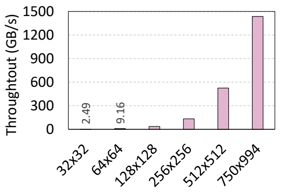
(b) Throughput on Hurricane with error bound 1E-4.

Figure 13: Compression throughput for different pipelines.

can be attributed to the initial estimates of the execution time for each segment, which were approximate and did not represent a perfectly uniform decomposition of the original algorithm. Note that the results are consistent with our analysis in Section 4.4. Such phenomenon can also be observed in decompression and other datasets as well.



(a) Throughput on CESM-ATM with error bound 1E-4.



(b) Throughput on HACC with error bound 1E-4.

Figure 14: Compression throughput with different WSE size.

The impact of WSE size. We also examine the impact of the WSE size, which denotes the width and height of the 2D mesh of PEs, on the throughput. We conduct experiments on two whole HPC datasets (CESM-ATM and HACC) with WSE sizes of 32x32, 64x64, 128x128, 256x256, 512x512, and 750x994, where 750x994 indicates the greatest number of PEs we can use for computation, while the rest PEs are used for routing data on and off the WSE. The result for compression is shown in Fig. 14. As seen, using more PEs can lead to linear speedups. For example, the throughput of using a 32x32 WSE is about 4 times of that using a 16x16 WSE on CESM-ATM and HACC. Specifically, such linear speedups are contributed by multiple rows, as demonstrated by profiling in Section 4.1. Based on this, we can also achieve linear speedups using more columns because the overhead of propagating data through the PEs to multiple pipelines is negligible as shown in (4). Therefore, using more PEs both in height and weight can also achieve linear speedups.

Observation 1: On average, CereSZ can achieve 457.35 GB/s and 581.31 GB/s throughput for compression and decompression, which is 4.9 and 4.8 times faster compared with the existing GPU compressor cuSZp.

5.3 Compression Ratio

In addition to the main focus – throughput, we also evaluate compression ratios of CereSZ along with baseline compressors, of which

		CESM-ATM		HACC		Hurricane		NYX		QMCPack		RTM	
	REL	range	avg	range	avg	range	avg	range	avg	range	avg	range	avg
CERESZ	1E-2	2.67~21.60	8.73	4.66~9.18	6.82	5.21~28.82	17.10	7.83~31.98	20.22	9.59~19.67	14.63	10.52~31.99	23.46
	1E-3	2.13~16.10	6.49	3.18~4.91	4.05	3.41~24.37	12.57	4.54~31.84	14.05	5.31~9.02	7.16	5.94~31.98	17.73
	1E-4	1.68~13.42	5.11	2.38~3.20	2.83	2.53~19.71	9.64	3.10~29.74	9.61	3.48~4.97	4.23	3.79~31.96	12.87
SZp	1E-2	9.91~70.48	23.72	10.16~13.62	11.56	10.80~88.94	40.26	12.21~127.80	67.58	12.44~22.45	17.45	14.22~127.94	67.51
	1E-3	6.70~69.15	20.14	3.82~9.63	5.39	7.44~57.42	23.92	8.62~125.55	40.16	6.08~11.60	8.84	7.91~127.79	43.40
	1E-4	4.22~67.65	17.03	3.49~5.51	3.57	4.49~37.08	15.29	4.91~98.23	23.41	3.79~6.57	5.18	4.73~127.51	28.19
cuSZp	1E-2	2.84~43.75	12.56	5.24~10.08	7.63	5.94~88.88	38.70	9.60~127.80	66.73	12.44~22.21	17.33	13.97~127.95	66.97
	1E-3	2.25~25.86	8.46	3.43~5.20	4.31	3.71~56.88	22.31	5.09~125.55	38.44	6.08~10.08	8.08	6.90~127.80	42.29
	1E-4	1.75~19.59	6.24	2.53~3.39	2.96	2.70~36.66	14.36	3.35~98.23	22.14	3.79~5.56	4.68	4.17~127.52	27.43
SZ	1E-2	26.13~4.0E+4	2.2E+3	16.58~931.76	217.94	23.76~404.71	110.33	1.3E+3~1.2E+5	2.3E+4	17.10~727.13	372.11	23.57~1.3E+5	4.4E+3
	1E-3	9.30~2.9E+4	941.39	6.11~30.97	15.57	8.81~105.49	35.67	84.55~1.8E+4	3.2E+3	6.37~221.11	113.74	9.27~2.3E+4	894.69
	1E-4	5.04~2.9E+4	825.49	3.74~8.92	5.75	4.63~48.46	18.72	14.38~2.6E+3	471.61	3.88~66.09	34.99	5.30~1.6E+4	548.91
cuSZ	1E-2	19.18~25.33	22.89	N/A	N/A	15.35~28.62	22.53	28.71~31.57	30.22	7.50~21.55	14.53	N/A	N/A
	1E-3	11.34~25.16	18.48	N/A	N/A	8.91~23.61	15.97	N/A	N/A	4.26~17.70	10.98	N/A	N/A
	1E-4	5.38~24.43	12.47	N/A	N/A	3.37~17.25	8.36	10.75~31.28	16.22	N/A	N/A	3.67~30.84	11.63

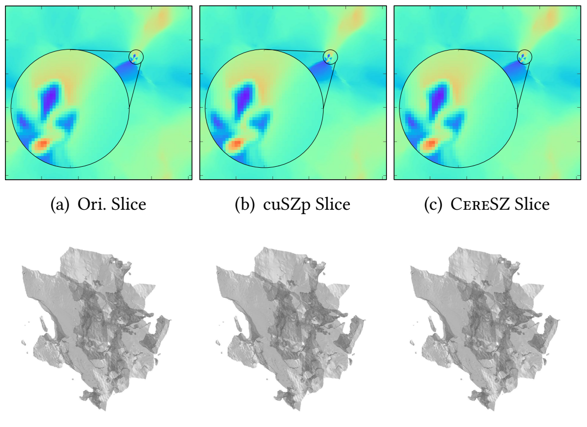
Table 5: Compression ratio results between CERESZ and four baseline error-bounded CPU or GPU compressors.

results are shown in Table 5. Same as Section 5.2, we select 3 error bounds, including REL 1E-2, REL 1E-3, and REL 1E-4, for a comprehensive evaluation. For each dataset, we measure compression ratios for all fields and record their range and the average value. Note that the "N/A" indicates we met bugs in cuSZ compression while storing Huffman codebooks, as confirmed with developers.

As we can see, SZ exhibits the highest compression ratio for all six datasets. The reason is that SZ (specifically SZ3) fine-tunes the prediction strategy between higher dimensional Lorenzo and Spline-Interpolation to aggregate spatial information and efficiently utilizes Huffman encoding along with best-fit lossless compression [29] to further reduce the overhead for storing quantization bins. However, one drawback of such designs in SZ is the limited throughput (routinely less than 1 GB/s), as demonstrated in Section 5.2. This drawback hinders its practical usage in inline compression tasks. As for SZp and cuSZp, the CPU version has higher compression ratios on CESM-ATM and HACC datasets due to its simpler implementations while recording the compressed block offsets.

For CERESZ, it has similar compression ratios compared with cuSZ. We also notice that, although with a similar compression algorithm, CERESZ leads to lower compression ratios compared with SZp and cuSZp in some cases such as NYX dataset REL 1E-2, where cuSZp and SZp have around 3x compression ratios of CERESZ. This is because CERESZ adheres to the 32-bit message passing requirement of the Cerebras WSE to maximize transaction bandwidth. As a result, it allocates 32 bits (or 4 bytes) to record the "fixed-length" at the start of each compressed block. In contrast, this block information requires only 1 byte in SZp and cuSZp, increasing the theoretical compression ratio upper bound by 4 times for sparse datasets (see RTM with REL 1E-2 in Table 6). However, we argue that this penalty in CERESZ can be relieved when we reduce the error bound, indicating CERESZ is more suitable for compression tasks under a strict error bound, such as DNN training, where the weight values are routinely small and irregularly distributed.

Observation 2: With higher throughput, CERESZ has similar compression ratios compared with cuSZ, and slightly lower compression ratios compared with SZp and cuSZp due to the message passing restriction.


(a) Ori. Slice (b) cuSZp Slice (c) CERESZ Slice

(d) Ori. Isosurface (e) cuSZp Isosurface (f) CERESZ Isosurface

Figure 15: Slice (3-th dim and 200-th panel) and isosurface visualization analysis for *velocity_x.f32* field in NYX dataset, where the error bound is REL 1E-4. In this case, cuSZP and CERESZ achieve 3.35 and 3.10 compression ratios. Both of these two compressors have SSIM 0.9996 and PSNR 84.77 dB.

5.4 Data Quality

As mentioned in Section 3, Pre-Quantization (❶) converts original floating points to quantization integers, which is the only lossy step in CERESZ. This design also appears in other several state-of-the-art GPU lossy compressors, such as cuSZ, FZ-GPU, and cuSZp – they only differ in lossless encoding and concatenation strategies while compressing the generated integers. As a result, for all compressors with pre-quantization, the loss of reconstructed data is only determined by the user-defined error bound. We demonstrate this in Fig. 15. Here we reconstruct the *velocity_x.f32* field by cuSZp and CereSZ under REL 1E-4 error bound and visualize them in both slice and isosurface. We can see that CereSZ and cuSZp share the same visualization quality, SSIM (0.9996), and PSNR (84.77 dB), once the error bound is determined. The only difference between cuSZp and CereSZ is the compromised compression ratio (3.35 vs 3.10), which will lead to a more conservative rate-distortion curve. In all, for further details on data quality, we recommend that readers refer to the works that feature pre-quantization design [13, 42, 50].

Beyond visualization quality, existing works [13, 50] also evaluate the data quality of a lossy compression from the perspective of rate-distortion curves. The rate-distortion curve measures the quantitative metrics (e.g. PSNR and SSIM) for different compressors under the same bit rate. Bit rate denotes how many bits are required to preserve each floating point data, which is usually the reciprocal of compression ratio. The higher PSNR/SSIM and lower bit rate represent a higher data quality that a lossy compressor preserves. As mentioned above, compressors including CereSZ, cuSZ, FZ-GPU, and cuSZp share the same pre-quantization design, resulting in the same reconstructed data (also PSNR and SSIM) under the same error bound. In this case, a compressor with higher compression ratios can deliver better rate-distortion curves. Since CereSZ achieves lower but close compression ratios compared with cuSZp, the rate-distortion curves are also slightly compromised.

Observation 3: CereSZ shares the identical visualization quality, SSIM, and PSNR with cuSZp under the same error bounds. The rate-distortion curves of CereSZ are slightly compromised compared with cuSZp.

6 RELATED WORK

6.1 Error-bounded Lossy Compression

Error-bounded lossy compression has been extensively studied in the past decade since it can reduce data size drastically while quantitatively controlling the numerical losses for post-hoc analysis [3, 8, 13, 22, 25, 28, 39, 50]. For the CPU platform, SZ [8, 39], ZFP [28], and MGARD [3, 25] are three leading lossy compressor and utilizes either prediction- or transformation-based designs. They mainly focus on attaining high compression ratios or high reconstructed data quality, instead of ultra-fast throughput.

In the meanwhile, some researchers also implemented lossy compression frameworks on heterogeneous processors (e.g. GPU and FPGA) with improved runtime throughput, to benefit inline compression tasks. Tian et al. proposed cuSZ [42] and waveSZ [41], which are the first prediction-based error-bounded lossy compression frameworks that utilize the hardware characteristics on GPU and FPGA. Yu et al. [48] introduced cuSZx, achieving high compression throughput by a constant block design and fast bit-level operations. Lindstrom et al. [27] implemented the fixed-ratio mode of ZFP, named cuZFP, encompassing the transform and bit truncation stages, to a single-kernel implementation on GPU, resulting in exceptionally fast compression and decompression speeds. Zhang et al. [50] proposed FZ-GPU, drastically improving the end-to-end throughput with warp-level optimizations and a partially fused compression/decompression kernel. Huang et al. [13] fuse entire compression/decompression stages (including quantization, prediction, lossless encoding, parallel scan, and block concatenation) into a single GPU kernel, achieving ultra-fast end-to-end throughput.

Although existing solutions achieve promising results in terms of throughput, compression ratio, and data quality on different platforms, this work has a different focus – CereSZ is the first high-throughput compression framework that exploits the data-flow architecture (e.g. Cerebras AI chips).

6.2 Cerebras-Accelerated Algorithms

In the realm of high-performance computing [15, 30, 35, 37, 38, 45, 47], Cerebras Systems has emerged as a pivotal and influential role. Algorithms designed on Cerebras achieve extraordinary performance, particularly excelling in machine learning and scientific computing workloads, due to their advanced architectural efficiencies and processing capabilities. Zvyagin et al. [55] successfully developed and implemented genome-scale language models (GenSLMs), specifically designed to comprehend and learn the complex evolutionary landscape of SARS-CoV-2 genomes. Remarkably, these models achieved the desired levels of accuracy and perplexity in their results within a timeframe of less than one day, demonstrating their efficiency and effectiveness in rapid computational biology. Ltaief et al. [30] expertly utilize the substantial memory bandwidth of the AI-specialized Cerebras CS-2 systems to efficiently process seismic data. This impressive performance is achieved through the strategic application of low-rank matrix approximation techniques, which are ingeniously designed to accommodate memory-intensive seismic applications within the inherently more limited SRAM capacity of the innovative wafer-scale hardware. This approach effectively tackles a common issue encountered in numerous wave-equation-based algorithms, particularly those dependent on Multi-Dimensional Convolution (MDC) operators. Orenes-Vera et al. [32] implemented fast Fourier transforms for one, two, and three-dimensional arrays on the Cerebras CS-2, of which 3D fast Fourier transforms of a 512^3 complex input array on the Cerebras CS-2 and achieves 959 microseconds, which is the first implementation that breaks the millisecond barrier. Different from the above works, CereSZ is the first touch to introduce data reduction to Cerebras architecture, which can complement and mutually reinforce other techniques in this field.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose an efficient error-bounded lossy compressor CereSZ on Cerebras CS-2 system for scientific data. We carefully select a compression algorithm that processes at a block granularity, thoroughly explore the parallelization of the compression/decompression algorithm, and propose innovative strategies that include both data parallelism and pipeline parallelism to effectively scale out the computation. We demonstrate the effectiveness of our pipeline parallelization approach, achieving linear speedups across the 2D mesh of computation units of Cererbras. Finally, we conducted a comprehensive evaluation of our proposed CereSZ on a variety of large-scale scientific datasets. CereSZ can deliver a throughput of

227.93 GB/s to 773.8 GB/s, 2.43x to 10.98x faster than existing GPU compressor cuSZp with only slightly compromised compression ratios. This result successfully showcases the potential of Cerebras data-flow architectures in data reduction compared with lossy compressors executed on existing platforms including CPU and GPU.

In the future, we plan to explore this work in two directions. First, we aim to further improve the computation balance and bandwidth utilization of PEs in CereSZ. Second, we plan to implement and optimize more compression/decompression algorithms and make them suitable for the dataflow architecture of Cerebras.

ACKNOWLEDGMENT

This research 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, OAC-2311875. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] [n.d.]. *Hurricane ISABEL simulation dataset in IEEE Visualization 2004 Test.* http://vis.computer.org/vis2004contest/data.html

[2] [n.d.]. *NYX simulation.* https://amrex-astro.github.io/Nyx/

[3] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2019. Multilevel techniques for compression and reduction of scientific data—the multivariate case. *SIAM Journal on Scientific Computing* 41, 2 (2019), A1278–A1303.

[4] Ann S Almgren, John B Bell, Mike J Lijewski, Zarija Lukić, and Ethan Van Andel. 2013. Nyx: A massively parallel amr code for computational cosmology. *The Astrophysical Journal* 765, 1 (2013), 39.

[5] Edip Baysal, Dan D Kosloff, and John WC Sherwood. 1983. Reverse time migration. *Geophysics* 48, 11 (1983), 1514–1524.

[6] Hervé Brönnimann, Bin Chen, Manoranjan Dash, Peter Haas, and Peter Scheuermann. 2003. Efficient data reduction with EASE. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 59–68.

[7] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1201–1220.

[8] 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.

[9] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. 2013. HACC: Extreme scaling and performance across diverse architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 1–10.

[10] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, et al. 2016. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy* 42 (2016), 49–65.

[11] Alain Hore and Djemel Ziou. 2010. Image quality metrics: PSNR vs. SSIM. In *2010 20th International Conference on Pattern Recognition.* IEEE, 2366–2369.

[12] Zhenbo Hu, Xiangyu Zou, Wen Xia, Yuhong Zhao, Weizhe Zhang, and Donglei Wu. 2021. Smart-DNN: Efficiently Reducing the Memory Requirements of Running Deep Neural Networks on Resource-constrained Platforms. In *2021 IEEE 39th International Conference on Computer Design (ICCD).* IEEE, 533–541.

[13] 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.* 1–13.

[14] Yafan Huang, Kai Zhao, Sheng Di, Guanpeng Li, Maxim Dmitriev, Thierry-Laurent D Tonellot, and Franck Cappello. 2023. Towards Improving Reverse Time Migration Performance by High-speed Lossy Compression. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid).* IEEE, 651–661.

[15] Peng Jiang, Yihua Wei, Jiya Su, Rujia Wang, and Bo Wu. 2022. SampleMine: A Framework for Applying Random Sampling to Subgraph Pattern Mining through Loop Perforation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.* 185–197.

[16] Pu Jiao, Sheng Di, Hanqi Guo, Kai Zhao, Jiannan Tian, Dingwen Tao, Xin Liang, and Franck Cappello. 2022. Toward Quantity-of-Interest Preserving Lossy Compression for Scientific Data. *Proceedings of the VLDB Endowment* 16, 4 (2022), 697–710.

[17] Sian Jin, Jesus Pulido, Pascal Grosset, Jiannan Tian, Dingwen Tao, and James Ahrens. 2021. Adaptive configuration of in situ lossy compression for cosmology simulations via fine-grained rate-quality modeling. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing.* 45–56.

[18] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2021. Comet: a novel memory-efficient deep learning training framework by using error-bounded lossy compression. *arXiv preprint arXiv:2111.09562* (2021).

[19] Jennifer E Kay, Clara Deser, A Phillips, A Mai, Cecile Hannay, Gary Strand, Julie Michelle Arblaster, SC Bates, Gokhan Danabasoglu, James Edwards, et al. 2015. The Community Earth System Model (CESM) large ensemble project: A community resource for studying climate change in the presence of internal climate variability. *Bulletin of the American Meteorological Society* 96, 8 (2015), 1333–1349.

[20] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, et al. 2018. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter* 30, 19 (2018), 195901.

[21] Argonne National Laboratory. [n.d.]. SZp: An OpenMP-accelerated error-bounded lossy compressor on CPU. https://github.com/szcompressor/SZp-OMP

[22] Shaomeng Li, Peter Lindstrom, and John Clyne. 2023. Lossy scientific data compression with SPERR. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 1007–1017.

[23] Shaomeng Li, Nicole Marsaglia, Christoph Garth, Jonathan Woodring, John Clyne, and Hank Childs. 2018. Data reduction techniques for simulation, visualization and data analysis. In *Computer graphics forum*, Vol. 37. Wiley Online Library, 422–447.

[24] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data).* IEEE, 438–447.

[25] Xin Liang, Ben Whitney, Jieyang Chen, Lipeng Wan, Qing Liu, Dingwen Tao, James Kress, David Pugmire, Matthew Wolf, Norbert Podhorszki, et al. 2021. MGARD+: Optimizing multilevel methods for error-bounded scientific data reduction. *IEEE Trans. Comput.* 71, 7 (2021), 1522–1536.

[26] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. 2022. Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data* 9, 2 (2022), 485–498.

[27] Peter Lindstrom. [n.d.]. cuZFP. https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp.

[28] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[29] Jinyang Liu, Sihuan Li, Sheng Di, Xin Liang, Kai Zhao, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2021. Improving lossy compression for sz by exploring the best-fit lossless compression techniques. In *2021 IEEE International Conference on Big Data (Big Data).* IEEE, 2986–2991.

[30] Hatem Ltaief, Yuxi Hong, Leighton Wilson, Mathias Jacquelin, Matteo Ravasi, and David Elliot Keyes. 2023. Scaling the "memory wall" for multi-dimensional seismic processing with algebraic compression on Cerebras CS-2 systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–12.

[31] Gabriel Marcus, Y Ding, P Emma, Z Huang, J Qiang, T Raubenheimer, M Venturini, and L Wang. 2015. High fidelity start-to-end numerical particle simulations and performance studies for LCLS-II. In *Proceedings, 37th International Free Electron Laser Conference (FEL 2015): Daejeon, Korea, August 23-28.*

[32] Marcelo Orenes-Vera, Ilya Sharapov, Robert Schreiber, Mathias Jacquelin, Philippe Vandermersch, and Sharan Chetlur. 2023. Wafer-Scale Fast Fourier Transforms. In *Proceedings of the 37th International Conference on Supercomputing.* 180–191.

[33] Etienne Robein. November 15, 2016. EAGE E-Lecture: Reverse Time Migration: How Does It Work, When To Use It.

[34] Benjamin Schlueter, Jon Calhoun, and Alexandra Poulos. 2023. Evaluating the Resiliency of Posits for Scientific Computing. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis.* 477–487.

[35] David Schneider. 2022. The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE spectrum* 59, 1 (2022), 34–35.

[36] Milan Shah, Xiaodong Yu, Sheng Di, Michela Becchi, and Franck Cappello. 2023. Lightweight Huffman Coding for Efficient GPU Compression. In *Proceedings of the 37th International Conference on Supercomputing*. 99–110.

[37] Shihui Song and Peng Jiang. 2022. Rethinking graph data placement for graph neural network training on multiple GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–10.

[38] Rick Stevens, Jini Ramprakash, Paul Messina, Michael Papka, and Katherine Riley. 2019. *Aurora: Argonne's next-generation exascale supercomputer.* Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[39] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1129–1139.

[40] Dingwen Tao, Sheng Di, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2019. Z-checker: A framework for assessing lossy compression of scientific data. *The International Journal of High Performance Computing Applications* 33, 2 (2019), 285–303.

[41] Jiannan Tian, Sheng Di, Chengming Zhang, Xin Liang, Sian Jin, Dazhao Cheng, Dingwen Tao, and Franck Cappello. 2020. Wavesz: A hardware-algorithm co-design of efficient lossy compression for scientific data. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 74–88.

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

[43] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. 2021. Revisiting huffman coding: Toward extreme performance on modern gpu architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 881–891.

[44] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[45] Yihua Wei and Peng Jiang. 2022. STMatch: accelerating graph pattern matching on GPU with stack-based loop optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[46] André Weißenberger and Bertil Schmidt. 2018. Massively parallel Huffman decoding on GPUs. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[47] Junqi Yin, Shubhankar Gahlot, Nouamane Laanait, Ketan Maheshwari, Jack Morrison, Sajal Dash, and Mallikarjun Shankar. 2019. Strategies to deploy and scale deep learning on the summit supercomputer. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 84–94.

[48] Xiaodong Yu, Sheng Di, Kai Zhao, Jiannan Tian, Dingwen Tao, Xin Liang, and Franck Cappello. 2022. Ultrafast error-bounded lossy compression for scientific datasets. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 159–171.

[49] Boyuan Zhang, Bo Fang, Qiang Guan, Ang Li, and Dingwen Tao. 2023. HQ-Sim: High-performance State Vector Simulation of Quantum Circuits on Heterogeneous HPC Systems. In *Proceedings of the 2023 International Workshop on Quantum Classical Cooperative*. 1–4.

[50] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. 2023. FZ-GPU: A Fast and High-Ratio Lossy Compressor for Scientific Computing Applications on GPUs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 129–142.

[51] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Martin Swany, Dingwen Tao, and Franck Cappello. 2023. Gpulz: Optimizing lzss lossless compression for multi-byte data on modern gpus. In *Proceedings of the 37th International Conference on Supercomputing*. 348–359.

[52] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1643–1654.

[53] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific data reduction benchmark for lossy compressors. In *2020 IEEE international conference on big data (Big Data)*. IEEE, 2716–2724.

[54] 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.

[55] Maxim Zvyagin, Alexander Brace, Kyle Hippe, Yuntian Deng, Bin Zhang, Cindy Orozco Bohorquez, Austin Clyde, Bharat Kale, Danilo Perez-Rivera, Heng Ma, et al. 2023. GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics. *The International Journal of High Performance Computing Applications* 37, 6 (2023), 683–705.