Ultrafast Error-Bounded Lossy Compression for Scientific Datasets

Xiaodong Yu
Argonne National Laboratory
Lemont, IL, USA
xyu@anl.gov

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

Kai Zhao
University of California
Riverside, CA, USA
kzhao016@ucr.edu

Jiannan Tian
Washington State University
Pullman, WA, USA
jiannan.tian@wsu.edu

Dingwen Tao
Washington State University
Pullman, WA, USA
dingwen.tao@wsu.edu

Xin Liang
Missouri University of Science and Technology
Rolla, MO, USA
xliang@mst.edu

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

ABSTRACT

Today's scientific high-performance computing applications and advanced instruments are producing vast volumes of data across a wide range of domains, which impose a serious burden on data transfer and storage. Error-bounded lossy compression has been developed and widely used in the scientific community because it not only can significantly reduce the data volumes but also can strictly control the data distortion based on the user-specified error bound. Existing lossy compressors, however, cannot offer ultrafast compression speed, which is highly demanded by numerous applications or use cases (such as in-memory compression and online instrument data compression). In this paper we propose a novel ultrafast error-bounded lossy compressor that can obtain fairly high compression performance on both CPUs and GPUs and with reasonably high compression ratios. The key contributions are threefold. (1) We propose a generic error-bounded lossy compression framework—called SZx—that achieves ultrafast performance through its novel design comprising only lightweight operations such as bitwise and addition/subtraction operations, while still keeping a high compression ratio. (2) We implement SZx on both CPUs and GPUs and optimize the performance according to their architectures. (3) We perform a comprehensive evaluation with six real-world production-level scientific datasets on both CPUs and GPUs. Experiments show that SZx is 2~16× faster than the second-fastest existing error-bounded lossy compressor (either SZ or ZFP) on CPUs and GPUs, with respect to both compression and decompression.

*Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA
2022. ACM ISBN 978-1-4503-9199-3/22/06.
https://doi.org/10.1145/3502181.3531473

CCS CONCEPTS

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

KEYWORDS

High-speed Compressor, Error-bounded Lossy Compression, GPU, Scientific Data

ACM Reference Format:
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 (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531473

1 INTRODUCTION

Background. With the ever-increasing complexity of modern scientific research, today's high-performance computing applications and advanced instruments are producing extremely large volumes of data in their simulations or experiments. The Hardware/Hybrid Accelerated Cosmology Code (HACC) [15], for example, can produce 20 TB of simulation data in only one run with hundreds of simulation iterations and trillions of particles involved.

Limitation of state-of-art approaches. During the past five years, several excellent error-bounded lossy compressors have been developed to resolve the big data issue. Nevertheless, the compression/decompression throughput is still far lower than the target performance demanded by many use cases, such as instrument data compression and in-memory compression. The Linear Coherent Light Source (LCLS-II) [27] could generate instrument data at a rate of 250 GB/s [8], and these data need to be stored and transferred to a parallel file system (PFS) in a timely manner for post hoc analysis. By comparison, the single-core CPU performance of existing lossy compressors is generally only 200~400 MB/s [21, 29], and the GPU performance is only 10~66 GB/s [10, 31], which has also been

verified in our experiments. Another typical example is exascale parallel quantum computing (QC) simulation, which requires a fairly large memory capacity (e.g., 2^{58} ≈256 PB when simulating 50 qubits each with double precision) for each run in practice [35]. To reduce memory requirement significantly, QC simulation researchers [35] have developed a method to store the lossy-compressed data in memory and decompress the data whenever needed in the course of the simulation. The method suffers, however, from considerable overhead in simulation time (up to ~20× in the worst case), which is undesired by users.

Research motivation and challenges. In this paper we focus on how to significantly accelerate both compression and decompression performance for error-bounded lossy compression while keeping a high compression ratio. This work involves addressing two grand challenges. (1) To pursue ultrahigh lossy compression/decompression performance, we have to restrict the whole design to use only fairly lightweight operations including addition/subtraction and bitwise operations. But doing so raises a serious challenge to maintaining a good compression ratio. Specifically, the relatively expensive operations such as multiplication and division should be suppressed because of their significantly higher cost. All of the existing efficient error-bounded lossy compressors, however, depend on such expensive operations. For instance, SZ 2.1 [21] relies on linear regression prediction, which involves masses of multiplications to compute the coefficients. Moreover, SZ 2.1 relies on a linear-scale quantization to control the user-specified error bound, which involves a division operation ($quantization\_bin=[\frac{prediction\_error}{2 \cdot error\_bound} + \frac{1}{2}]$ [13]) on each data point. ZFP [22] is another state-of-the-art error-bounded lossy compressor, which is designed based on the data transform; it also involves masses of matrix-multiplication operations. (2) Parallelizing the whole design on parallel architectures, especially the massively parallel GPU devices, is challenging. The dependencies are exposed during the parallelization, and some of these are extremely difficult to break. Therefore, a sophisticated design and optimization are desired in order to enable the parallelization and achieve optimal performance.

Key contributions. We propose a novel, ultrafast, error-bounded lossy compression framework—*SZx*—for both CPUs and GPUs. The key contributions are summarized as follows.

- We develop SZx, which composes only lightweight operations such as bitwise operations, additions, and subtractions. SZx also supports strict control of the compression errors within user-specified error bounds, thanks to our careful design of the error-control mechanism.
- We optimize the SZx algorithm using inexpensive bitwise right shifting to improve the performance; we also investigate the compression quality improvement by exploring the best block size.
- We implement SZx on both CPUs and GPUs with sophisticated and novel designs and optimize the performance with respect to their architectures.
- We comprehensively evaluate SZx by running it with six real-world scientific datasets on heterogeneous compute nodes offered by different supercomputers, including Summit at Oak Ridge National Laboratory (ORNL) and ThetaGPU at Argonne National Laboratory (ANL). We rigorously compare SZx with two state-of-the-art lossy compressors, SZ and ZFP, as well as their GPU versions cuSZ and cuZFP, respectively.

Experimental results and artifact availability. Experiments show that SZx is 2~7× faster than the second-best existing error-bounded lossy compressor on CPUs and 2~16× faster than the second-best compressor on GPUs, with respect to both compression and decompression. At such high performance, SZx can still get a good compression ratio—3~12 for the overall compression ratio of each application and up to 124 for the compression ratio of the specific fields—with good reconstructed data quality.

Paper structure. The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 4 we present a design overview of our ultrafast error-bounded lossy compression framework. In Section 5 we propose algorithmic optimizations for improving both the performance and the compression quality. In Section 6 we describe the SZx implementations on both CPUs and GPUs. In Section 7 we present and discuss the performance evaluation results. In Section 8 we conclude the paper with a vision of future work.

2 RELATED WORK

High-speed scientific data compression can be split into two basic categories—lossless compression and lossy compression. Each of these will be discussed in the following text, especially with regard to performance/speed.

High-speed lossless compressors have been developed because of the strong demand on compression performance in many use cases. Facebook Zstd [9], for example, was developed for the sake of high performance, with very similar compression ratios compared with other state-of-the-art lossless compressors such as Zlib [44] and Gzip [11]. In general, Zstd can be 5~6× faster than Zlib, as shown in [9], and hence it has been widely integrated and used in 80+ production-level software codes, libraries, and platforms. Unfortunately, Zstd supports only lossless compression, which would mean very low compression ratios (1.2~2 in most cases) when compressing scientific datasets that are composed mainly of floating-point values (to be shown later).

High-speed lossy compression has also gained significant attention by compressor developers and scientific applications researchers. SZ [12, 21, 29] is a fast error-bounded lossy compressor, which can reach 200~300 MB/s in compression and decompression speed [12, 21, 29]. However, it is still not as fast as expected by quantum computing simulations [35], so a faster lossy compression method called *QCZ* was customized with comparable compression ratios (especially for high-precision compression with a relative error bound of 1E-4 or 1E-5). ZFP [22] is another fast error-bounded lossy compressor, which is well known for its relatively high compression ratios and fairly high compression speed on both CPUs and GPUs. Based on our experiments (to be shown later), ZFP and QCZ have comparable compression speed, and they are generally 1.5~2× as fast as SZ. We emphasize that, in fact, SZ already has higher performance than many other compressors, as demonstrated in literature: it has a comparable performance with FPZIP [23] and SZauto [43] and about one to two orders of magnitude higher performance than ISABELA [19], MGARD [6], and TTHRESH [7].

Over the past decade, GPUs have become prevalent because of their massive parallelism and computational power [25]. Various applications have been successfully accelerated on GPU-based platforms [36, 37, 39–42]. Because of the high demand for ultrafast error-bounded lossy compressors, a few specific error-controlled lossy compression algorithms have been developed for GPU accelerators; cuSZ [31] and cuZFP [10] are two leading ones. The cuSZ algorithm is the only GPU-based lossy compressor supporting absolute error bounds for scientific data compression. It was designed based on the classic prediction-based compression model SZ and optimized for GPU performance by leveraging a dual-quantization strategy [31] to deal with the Lorenzo prediction. The cuZFP compressor, on the other hand, leverages the high-performance CUDA library to reach a very high throughput; it can do so because ZFP's core stage is performing a customized orthogonal data transform that can be executed in the form of matrix-multiplication. CuZFP, however, does not support error-bounded compression but only fixed-rate compression, which suffers from very low compression ratios, as verified in [33].

In comparison with all these related works, our proposed SZx is about 2~7× as fast as the second-fastest lossy compressor ZFP on CPUs and 2~16× as fast as the second-fastest (cuSZ) on GPUs, also with relatively high compression ratios (3~12 depending on the user's error bound).

3 PROBLEM FORMULATION

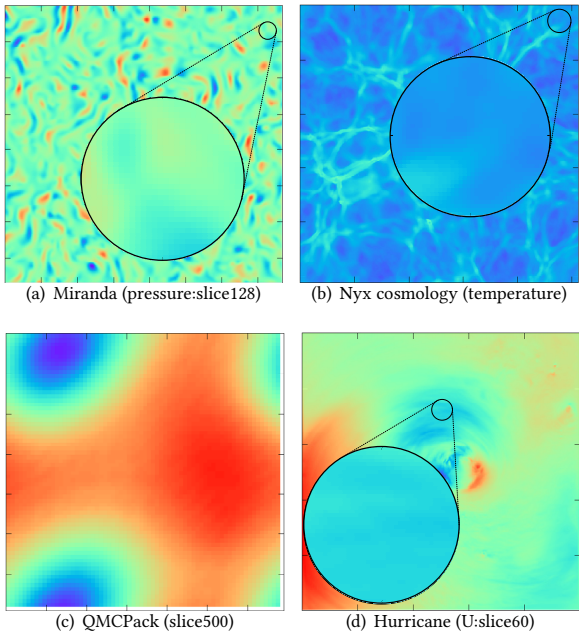
In this section we formulate the research problem we focus on in this paper: optimization of the error-bounded lossy compression/decompression performance with compression ratios as high as possible. Specifically, given a scientific dataset (denoted by D) composed of N data values each denoted by d_i, where i=1,2,3,$\cdots$,N, the objective of our work is to develop an error-bounded lossy compressor with a fairly high performance in both compression and decompression for both CPUs and GPUs, while also strictly respecting the user-required error bound, which can be represented as the following formula:

$$\begin{aligned} &\max(CT) \ \textit{and} \ \max(DT) \\ &s.t. \ |d_i - d_i'| \le e \\ &\quad \textit{CR is relatively high}, \end{aligned} \tag{1}$$

where CT and DT represent the compression throughput and decompression throughput, respectively; d_i and d_i' denote the original data value and decompressed data value in the dataset, respectively; e is the user-specified absolute error bound; and CR is the compression ratio, which is defined as the ratio of the original data size to the lossy compressed data size. In order to obtain as high performance as possible, CR would definitely be not optimal. However, we still hope to get a relatively high CR (expected to be over 5 or 10). Here, "relatively high" refers to much higher CR than that of lossless compressors (typically 1.2~2 for scientific data), as we prove by experiment data in Section 7. This objective is meaningful to those users who can tolerate errors but cannot afford high compression overhead and high decompression overhead.

The compression throughput and decompression throughput are defined in Formula (2) and Formula (3), respectively:

$$CT = (N \cdot b)/T \tag{2}$$



(a) Miranda (pressure:slice128) (b) Nyx cosmology (temperature)

(c) QMCPack (slice500) (d) Hurricane (U:slice60)

Figure 1: Demonstrating High Smoothness of Scientific Datasets

$$DT = (N \cdot b)/T', \tag{3}$$

where N is the number of data points in the dataset D; b represents the number of bytes per value in D (e.g., b = 4 when the original data precision is single-precision floating point); and T and T' denote the time cost when compressing the dataset D and the time cost when decompressing the corresponding compressed data, respectively.

In addition to the maximum compression error (i.e., error bound as shown in Formula (1)), we evaluate the reconstructed data quality by popular data distortion metrics such as peak signal to noise ratio (PSNR) [30] and structural similarity index measure (SSIM) [34], which have been commonly used by the lossy compression and visualization community. In general, the higher the PSNR or the higher the SSIM, the better the reconstructed data quality.

4 ULTRAFAST ERROR-BOUNDED LOSSY COMPRESSION FRAMEWORK – SZX

In this section we present the design overview of our ultrafast error-bounded lossy compression framework SZx. Detailed performance optimization strategies for CPUs and GPUs will be discussed in the next section.

Our design is based on the fact that most of the scientific datasets are fairly smooth in space, such that all the values in a small block (e.g., 16 or 32 consecutive data points) are likely to be close to one another. Thus the mean of the minimal value and maximal value in the block can be used to represent the whole block based on a certain error bound. Figure 1 shows a visualization of four typical fields from four different real-world simulation datasets—Miranda large-eddy simulation [3], Nyx cosmology simulation [4], QMCPack quantum chemistry [18], and a hurricane climate simulation [1])—clearly demonstrating the high smoothness of the data in local

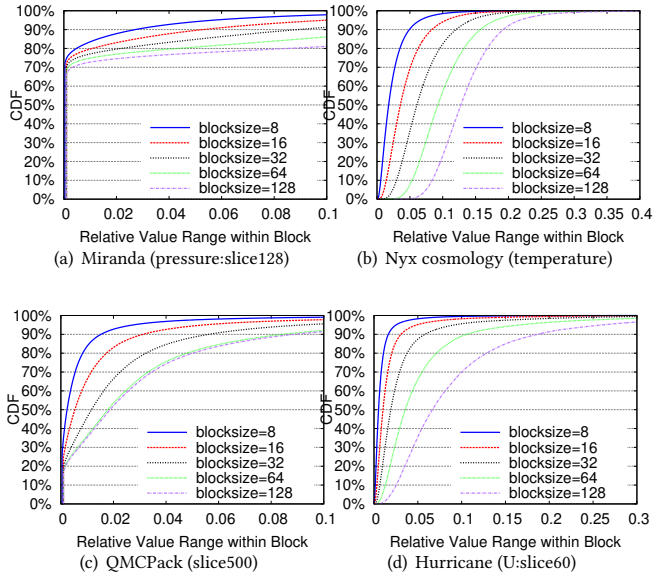


Figure 2: Cumulative Distribution Function (CDF) of Block's Value Range

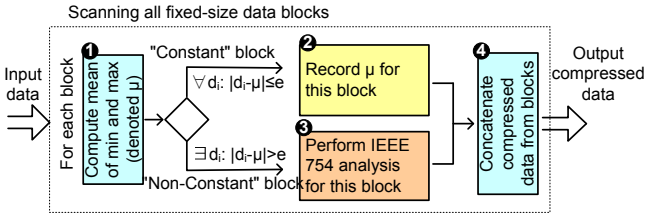


Figure 3: Design architecture/workflow of SZx

spatial regions. Furthermore, Figure 2 shows the cumulative distribution function of the block's relative value range.[1] It verifies that the four scientific datasets all exhibit fairly high smoothness of the local data without loss of generality. Specifically, for the Miranda dataset and QMCPack dataset, 80+% of the blocks have very small relative value ranges (≤0.01), when the block size is 8.

We design our compressor SZx in terms of the local smoothing feature, as illustrated in Figure 3. The fundamental idea is organizing the whole dataset as many small 1D blocks (or segments) and checking whether the mean of the min and max (denoted by μ) in each block can be used to represent all values in this block with deviations respecting the user-specified error bound. If yes, we call this block a "constant" block, and we just need to store μ for this block of data; otherwise, we compress all the data points in this block by analyzing their IEEE 754 representations in terms of the user-required error bound.

We present the pseudocode of the skeleton design in Algorithm 1 to further describe details. Table 1 summarizes the key notation to assist in understanding the algorithm.

[1] A block's relative value range is defined as the ratio of the block's value range to the dataset's global value range. The reason we check the block's relative value range is that the error-bounded lossy compression is often performed via a value-range based relative error bound [30], where the absolute error bound is calculated based on the dataset's global value range.

Table 1: Key Notations Involved in The SZx Algorithm

Notation	Description
D	dataset given for compression
e	user-specified error bound
d_i	data points in the original raw dataset D
B_k	kth block in the dataset
μ_k	mean of min and max in Block k
r_k	variation radius of Block k
R_k	required bits calculated via e and μ_k for B_k
v_i	normalized values based on μ_k in each block B_k
L_i	identical leading bits of v_i compared with v_{i-1}

We describe Algorithm 1 as follows. As mentioned previously, the whole dataset is split into many small fixed-size 1D blocks, and the compression will be executed block by block (line 2). Because of the high smoothness of data in locality, quite a few data blocks may have values that already respect the error bound based on the mean of the min and max (denoted by μ) (lines 4~6); these "constant" blocks will be compressed by simply storing the corresponding μ value. The types of blocks need to be kept in a separate array called *type_array*, which will be used to decide block type during the decompression stage.

Algorithm 1 Skeleton Design of SZx

Input: dataset D, user-specified error bound e, block size (denoted b)
Output: compressed data stream in form of bytes

```
1:  i ← 0, k ← 0;/*Set 0 to all counters*/
2:  for each block B_k with block size b do
3:      Compute μ_k for B_k; /*Compute mean of min and max*/
4:      if (∀d_i ∈ B_k: |d_i − μ_k| ≤ e) then
5:          type_array←0;/*0 indicates 'constant block'*/
6:          μ_array ← μ_k; /*Collect μ for 'constant' blocks*/
7:      else
8:          type_array←1;/*1 indicates 'nonconstant block'*/
9:          Compute required number of bits (denoted as R_k);
10:         for each normalized value v_i in B_k do
11:             Compute identical_leading_bytes for v_i and v_{i−1};
12:             Encode identical_leading_bytes into xor_leadingzero_array;
13:             mb_array ← R_k − L_i; /*Commit required bits excluding L*/
14:         end for
15:     end if
16:     Aggregate output: μ_array, xor_leadingzero_array, mb_array;
17: end for
```

For each of the nonconstant blocks, we first normalize the data by subtracting the mean of min and max in the block (i.e., μ) and then compress each such normalized value by IEEE 754 binary representation analysis according to the following three steps.

- **Line 9**: We compute the required number of significant bits (denoted as R_k) based on user-specified error bound, by the following formula:

$$R_k = \begin{cases} 0, & p(r_k) - p(e) \le 0 \\ fullbits(type), & p(r_k) - p(e) > fullbits(type) \\ p(r_k) - p(e), & otherwise, \end{cases} \quad (4)$$

  where $p(x)$ denotes getting the exponent of the number x , r_k denotes the variation radius of data in the block k, and fullbits(type) refers to the data type's size (e.g., 32 bits for single-precision floating-point type). The idea is to normalize the data values by subtracting the mean of the min and max

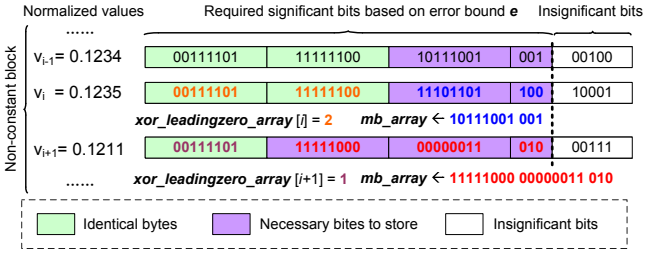


Figure 4: Compressing nonconstant float32 block by binary representation analysis: suppose three adjacent normalized values in a nonconstant block are 0.1234, 0.1235, and 0.1211, respectively. (The required significant bits are determined by Formula (4).)

such that the maximum exponent of each normalized value is foreseeable and thus the required bits are estimable by combining the exponent of the error bound e.

- **Line 11**: We compute identical leading bytes by an XOR operation between the normalized data value v_i and its preceding data value v_{i-1}. The number of leading zeros after the XOR operation indicates the number of identical leading bytes between the two data points.
- **Line 12**: We encode the number of identical leading bytes for each data point by a 2-bit code: 00, 01, 10, and 11 correspond to 0, 1, 2, and 3 identical leading bytes, respectively. We use a 2-bit-per-value array (called *xor_leadingzero_array*) to carry these 2-bit codes, as illustrated in Figure 4.
- **Line 13**: We commit the necessary significant bits, that is, the required bits (denoted as R_k) excluding identical leading bytes (denoted by L_i), to a particular mid-bits array (denoted as *mb_array*), as shown in Figure 4.

5 ALGORITHMIC OPTIMIZATIONS

In this section we describe our specific optimization strategies at the algorithm level. These optimizations aim to improve both the performance and the compression ratio.

5.1 Performance Optimization by Bitwise Right Shifting

Here we describe how to accelerate SZx by an efficient bitwise right-shifting operation, which mainly involves lines 9~14 in Algorithm 1. This is a fundamental optimization strategy that can also be applied in other devices/accelerators such as GPUs. In what follows, we first describe a potential performance issue in the SZx design, followed by our optimization solution.

As illustrated in Figure 5, the mantissa bits that need to be stored for the normalized value v_i should exclude the identical bytes L_i and the insignificant bits that are calculated based on the user-specified error bound and variation radius of the corresponding block. The number of such necessary mantissa bits is generally not a multiple of 8 (to be verified later), so that committing/storing these bits in the compressed data requires specific bitwise operation strategies.

Storing a short bit-array with an arbitrary number of bits is a common operation in lossy compression. The most straightforward solution (Solution A as shown in Figure 5) is treating the given bit-array as a particular integer and populating the target bit-stream pool (i.e., mb-array in the figure) by applying a couple of bitwise

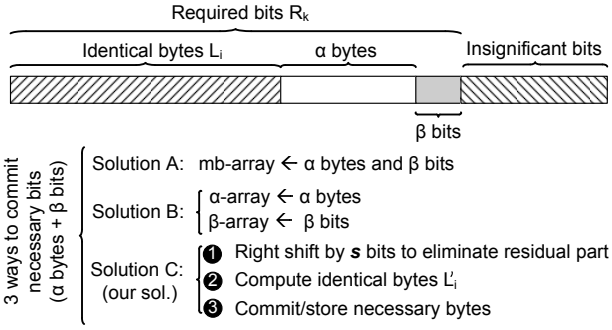


Figure 5: Three ways to store necessary mantissa bits (Solution C is our performance optimization strategy)

operations (such as bit shift, bit and, and bit or) on the integer number. Many lossy compressors, such as Pastri [14], store the arbitrary bits in this way. An alternative solution (Solution B as shown in Figure 5) is splitting the necessary bits into two parts—a number of necessary bytes (α bytes) plus a few residual bits (β bits); this approach was adopted by SZ [12, 13]. In this solution the residual bits with varied number of bits still need to be gathered in a target array by a set of bitwise operations.

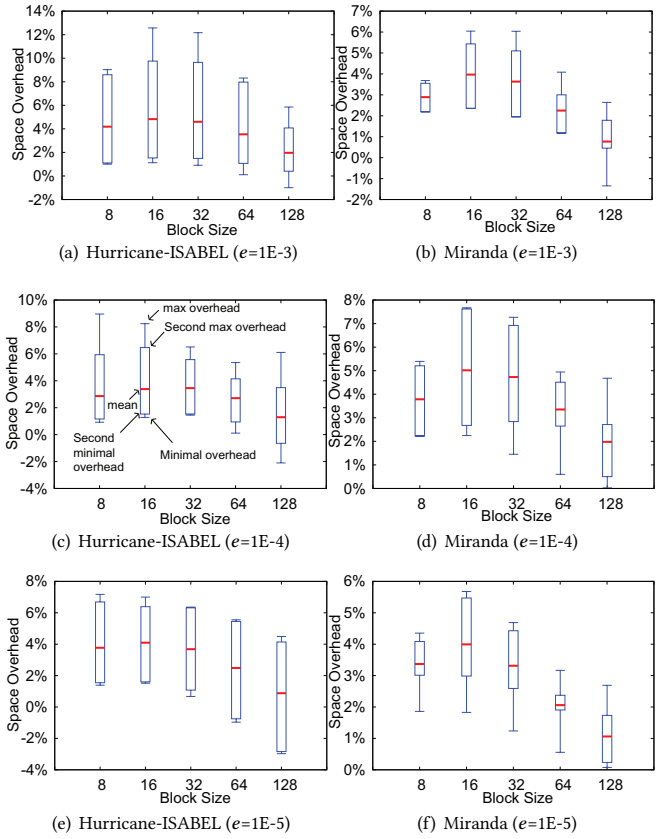
By comparison, we develop an ultrafast method (Solution C as shown in Figure 5) to deal with the necessary bits efficiently. The basic idea is bitwise right shifting the normalized value by s bits, where s is given in Formula (5), such that the number of the necessary bits to be stored is always a multiple of 8. The necessary mantissa bits then can be represented by an integer number of bytes, with eliminated residual bits. In this situation we just need to use a memory copy operation to commit the necessary bits to one byte array, which would be fairly fast:

$$s = \begin{cases} 0, & R_k\%8 = 0 \\ 8 - R_k\%8, & R_k\%8 \neq 0 \end{cases} \tag{5}$$

5.2 Investigation of Space Overhead for Bitwise Right Shifting

The bitwise right-shifting operation may increase the total number of required bits to store, thus reducing the compression ratios in turn. In the following text, we will show that the increased number of bits per value because of the bitwise right-shifting operation is very limited compared with the compressed data size, thanks to the design of identical leading bytes. Such a space overhead is negligible in most cases. In fact, although the bitwise right-shifting operation may increase the required number of bits, this operation may also potentially increase the number of identical leading bytes, such that some necessary bits could be "recorded" by the identical leading array instead. In other words, after the bitwise right-shifting operation, the necessary bits tend to increase on the right end but tend to decrease on its left end, thus forming a counteraction to a certain extent.

We use Figure 6 (based on two real-world simulation datasets with different value-range-based error bounds [30]) to show the specific space overhead of our solution designed with the bitwise right-shifting operation, as compared with the compressed data size. The space overhead is defined as the ratio of the increased storage space introduced by the bitwise right-shifting method to



(a) Hurricane-ISABEL (e=1E-3) (b) Miranda (e=1E-3)

(c) Hurricane-ISABEL (e=1E-4) (d) Miranda (e=1E-4)

(e) Hurricane-ISABEL (e=1E-5) (f) Miranda (e=1E-5)

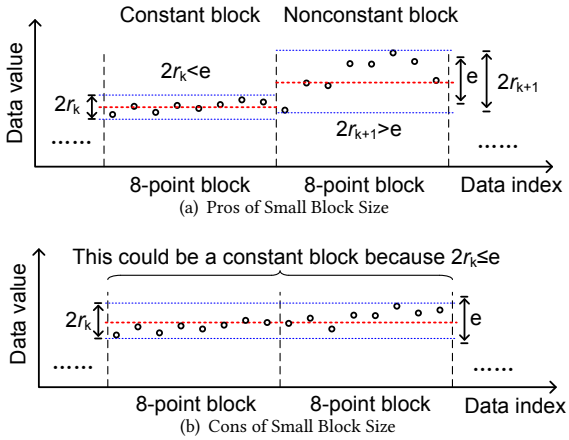
Figure 6: Space overhead of bitwise right shifting used in SZx. The figure shows the min, 2nd-min, avg, 2nd-max, and max overhead for two application datasets each with multiple fields.

the compressed data size, as presented in Formula (6):

$$Overhead = \frac{\sum_{v_i \in D} (R_k + s - L_i') - \sum_{v_i \in D} (R_k - L_i)}{D_{size}/CR}, \quad (6)$$

where CR is the compression ratio, D_{size} refers to the original data size (thus D_{size}/CR means compressed data size), $\sum_{v_i \in D} (R_k + s - L_i')$ refers to the total amount of necessary bytes to store under the Solution C (our solution), and $\sum_{v_i \in D} (R_k - L_i)$ refers to the total amount of necessary bytes to store by Solution A or B.

In Figure 6, which involves a total of about 100 different fields across these two applications, one can clearly observe that the space overhead is always lower than 12% for all the fields and that the average overhead for each case (with a specific block size) is always around or lower than 5% compared with the compressed data size. We give an example to further explain how small the overhead is. Specifically, for the field "density" in the Miranda simulation dataset, the original data size is 256×384×384×bytes = 144 MB, and the compression ratio of SZx is 9.923, so the compressed data size is about 15.2 MB. Our characterization shows that Solution B and Solution C lead to 81,340,334 necessary bits (i.e., 10,167,542 bytes) and 83,054,120 necessary bits (i.e., 10,381,765 bytes), respectively, which means the overhead is only $\frac{10,381,765-10,167,542}{15.2MB}$=1.4% for this field.



(a) Pros of Small Block Size

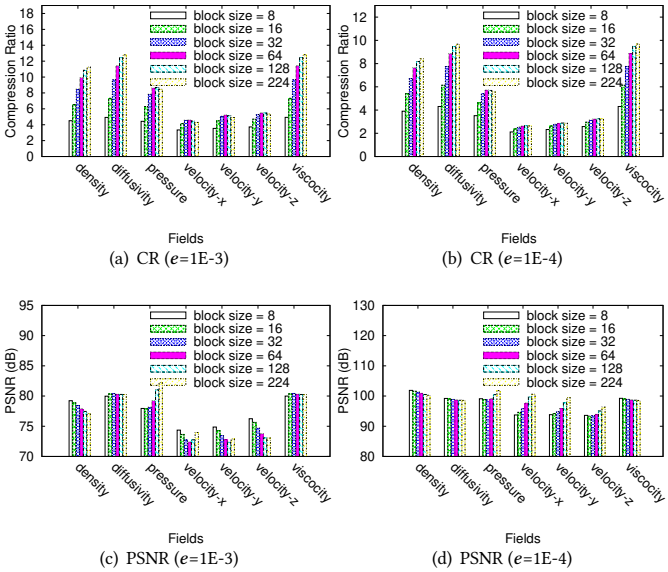
(b) Cons of Small Block Size

Figure 7: Constant block's pros and cons when block size is small

5.3 Optimization of Compression Quality by Exploring Best Block Size

Different block sizes may affect the compressed data sizes (i.e., compression ratios) significantly. Thus we must investigate the most appropriate setting of the block size for SZx. As described previously, the design comprises two types of blocks, called "constant" blocks (lines 4~5 in Algorithm 1) and "nonconstant" blocks (lines 6~12 in Algorithm 1), respectively. Before exploring the optimal block size, we need to understand how the two types of blocks contribute to the compressed data size (or compression ratios). To this end, we analyze three *impact factors*.

- *Analysis of constant blocks* Constant blocks refer to blocks each of which can be approximated by using one data value μ_k (i.e., mean of min and max): the smaller the block size, the more data points to be included in the constant blocks, because of the finer-grained blockwise processing, as illustrated in Figure 7 (a). As shown in the figure, the first set of 8 data points can form a constant block because of the relatively small block size. In this sense, the compression ratio tends to increase with decreasing block size because all the values within the constant block can be approximated by one value (i.e., μ_k), which is called **impact factor** Ⓐ in the following text. However, since each constant block needs to store a constant value μ_k in the compressed data, the smaller the block size, the larger number of μ_k need to be stored, which may also decrease the compression ratio in turn, as illustrated in Figure 7 (b). We call this phenomenon **impact factor** Ⓑ. Specifically, for the relatively smooth regions in the dataset, the algorithm still needs to store multiple μ_ks even though a large number of adjacent data points could be approximated by only one uniform value instead. This may introduce significant overhead because of extra unnecessary μ_k to store, thus leading to lower compression ratios.
- *Analysis of nonconstant blocks* On the one hand, the impact factor Ⓑ also applies on nonconstant blocks since they also need to store μ for data denormalization during the decompression. On the other hand, a smaller block size may tend to get a higher compression ratio because of the following reason: the smaller the block size, the smaller the variation



(a) CR (e=1E-3) (b) CR (e=1E-4)

(c) PSNR (e=1E-3) (d) PSNR (e=1E-4)

Figure 8: Compression quality of Miranda data with various block sizes

in the block (i.e., smaller μ_k), and thus the fewer necessary bits to store. We call this **impact factor** ❸. Specifically, as shown in Figure 7 (a), the first 8-point block has much smaller data variation than does the other one, so that the corresponding required exponent would be smaller, leading to fewer required mantissa bits (according to Formula (4)).

Based on this analysis, different block sizes may have distinct pros and cons with regard to compression quality of the two types of blocks. It is not obvious what block size can get the best compression quality. In what follows, we explore the best block size setting by characterizing the compression ratios and PSNR with different block sizes, as presented in Figure 8. PSNR is a critical lossy compression data quality assessment metric and has been widely used in the lossy compression and visualization community [12, 22, 29, 30, 38, 43]. PSNR is defined in Formula (7):

$$psnr = 20\log_{10}\frac{(d_{\max} - d_{\min})}{\sqrt{MSE}}, \tag{7}$$

where d_{min} and d_{max} are the min value and max value in the dataset D and MSE refers to the mean squared error between the original dataset D and reconstructed dataset D'. The higher the PSNR, the higher the precision of the reconstructed data.

In the exploration we checked many different error bounds from 1E-3 through 1E-6. Because of space limits, we present in Figure 8 only the results about the value-range-based error bound of 1E-3 and 1E-4, which compress seven fields of the Miranda simulation dataset by SZx. Other error bounds and datasets exhibit similar results.

From Figure 8 we can observe that the compression ratio increases with block size in most cases, while the PSNR always stays at a similar level across different block size settings. We empirically find that the best block size is 128. Figures 8 (a) and (b) show that the CR will converge after the block size becomes larger than 128, while block size 128 and 224 exhibit more or less the same PSNR

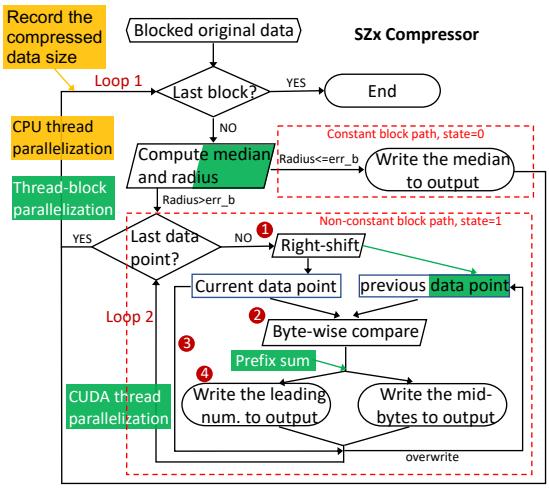


Figure 9: Flowchart of SZx's parallel compression

in Figures 8 (c) and (d). With the same accuracy, smaller block size can lead to better GPU performance. This characterization also indicates that **impact factor** ❷ dominates the overall compression ratios, because this is the only factor that may enhance the compression ratio with increasing block size.

6 IMPLEMENTATIONS ON CPUS AND GPUS

In this section we elaborate on the parallel design details of the SZx compressor and decompressor on both CPUs and GPUs. For the GPU implementation we describe the data dependencies and our efficient designs that can overcome them.

6.1 CPU-Based Design and Parallelization

Compressor: Figure 9 shows the workflow of SZx's parallel compression. In this figure the flows consist of white boxes and black arrows, describing the baseline sequential implementation. The yellow box indicates CPU parallelization (discussed later in this subsection), while the green boxes indicate the designs for GPU parallelization (detailed in next subsection). The original data is divided into equal-sized data blocks as the input for the compressor. With the baseline implementation, the data blocks are processed iteratively (*Loop 1*). For each block, the compressor first computes the μ (i.e., mean of min and max) and radius. if the radius is smaller than the error bound, then the *constant block path* is taken, and μ will be recorded. Otherwise, the data block will go through the *non-constant block path*. In this path, all data points of the block are compressed in sequence (*Loop 2*). The compression of a data point consists of four steps. (1) Right shift the current data point as described in Section 5.1. (2) Compare the current and previous data point in bytewise fashion. (3) Store the current data point as the previous data point for the next iteration. (4) Count the leading identical bytes as the leading number, and record the remainders as mid-bytes.

Decompressor: Figure 10 depicts the SZx's parallel decompression workflow. Similar to Figure 9, the white boxes indicate the baseline implementation. The input data is decompressed block by block (*Loop 1*). The constant blocks are retrieved by using the

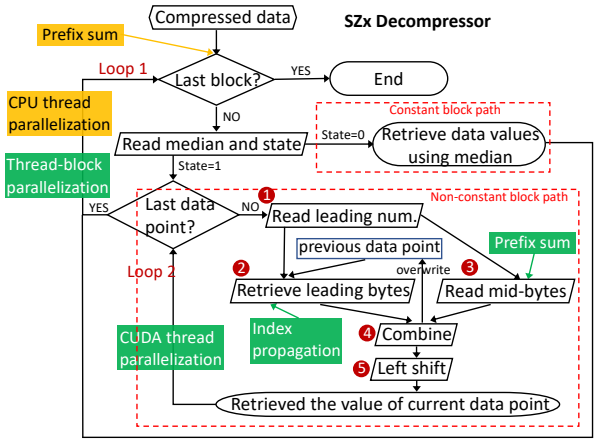


Figure 10: Flowchart of SZx's parallel decompression

corresponding μ value, while the nonconstant blocks would be reconstructed point by point (*Loop 2*). The reconstruction of each data point in a nonconstant block has five steps. (1) Read the leading number n of the current data point. (2) Retrieve the leading bytes by copying the first n bytes of the previous data point. (3) Retrieve the remaining bytes by reading the corresponding mid-bytes. (4) Combine the retrieved bytes to form the current data point, and use it to overwrite the previous data point for the next iteration. (5) Left shift the current data point to get the decompressed value.

Parallelization: We use OpenMP to parallelize SZx on a multicore CPU. Our blockwise design simplifies the parallelization since each data block is compressed and decompressed independently. Accordingly, we break the *Loop 1*s in Figures 9 and 10 and assign the processing of different data blocks to different CPU threads, as indicated by the yellow boxes. The only issue is that, during the decompression, the offsets of the data blocks in the compressed bytes are unknown to the CPU threads, because the data blocks have distinct compressed data lengths. This dependency makes the CPU threads unable to read the compressed blocks from the correct addresses. To solve this problem, we use a 16-bit integer array (called *zsize_array*) to record the compressed data size for each block. Specifically, each element (with *unsigned char* data type) in the array is used to record the number of bytes after compression per data block. This *zsize_array* needs to be stored together with the compressed data because it is needed to determine the starting location in the compressed data stream for different threads during the decompression. To implement this design, the decompressor uses a prefix-sum step to compute the starting location of each block for all threads and then performs the decompression of all the blocks in parallel.

6.2 Design and Optimization for GPUs

In this subsection we describe our design and implementation for cuSZx—the CUDA GPU version of SZx. The green boxes in Figures 9 and 10 illustrate the modifications we made on the CPU implementation workflow to accommodate GPU architecture.

6.2.1 Design Overview and General Optimization. Our cuSZx design assigns different data blocks to different CUDA thread-blocks (i.e., unrolling *Loop 1*) and uses different CUDA threads to process different data points in a data block (i.e., unrolling *Loop 2*). Thanks to the independence between data blocks, this design completely avoids expensive grid-level synchronization and communication. In order to optimize the performance, the data block size is chosen as a multiple of the GPU warp size, and the threads in each thread-block are organized in two dimensions with the x-dimension length equal to the warp size.

During the compression, each thread block performs parallel *min* and *max* with the help of CUDA warp-level operations to compute the μ and radius of the assigned data block. If the assigned data block is identified as a constant block, the thread block will record the μ and immediately go forward to process the next data block. Since the number of data blocks is considerably larger than the number of thread blocks, this scheduling fashion can significantly mitigate the workload imbalance. The nonconstant block will be further processed in the thread block with the approach that one CUDA thread compresses one data point following the four steps depicted in Figure 9. This design, however, exposes some data dependencies and hence requires algorithmic adjustments to make the thread-level parallelization smooth. They will be discussed in Section 6.2.2.

In the decompression, since the states of data blocks are known and the constant block retrievals just need to get μ, we decompress only the nonconstant blocks in the GPU. Similar to the compression parallelization, each thread block processes one data block at a time, and each of its CUDA threads decompresses one data point following Steps 1–5 in Figure 10. The thread-level parallel decompression also exposes dependencies, and we will discuss the solutions in Section 6.2.2 as well.

We design several general optimizations to benefit the overall performance of both the compression and decompression. (1) Instead of letting each thread read one *char* data from a *char* array (e.g., mid-byte array), we use one warp to read the array as *char4* data type and then share the read data with other warps through shared memory. With this design we can maximize the bandwidth usage and reduce the global memory accesses. (2) We store the current data block into shared memory after the first time it is read by its thread block, to accelerate its reuse. This design also enables the optimal bytewise manipulation of floating-point values on the GPU since the GPU registers do not support the *union* data structure.

6.2.2 Challenges and Solutions. The thread-level parallelization of point-by-point compression and decompression for nonconstant blocks exposes two types of dependencies: address dependency and data value dependency. The two raise separate challenges and need different strategies in order to be overcome.

Challenge 1: In Figure 9 the number of mid-bytes of every data point is unknown before Step 2. Therefore, an iterator is required in the sequential implementation to indicate the offset in *mb_array* serving as the starting address for writing the current data point's mid-bytes. In GPUs, however, since all data points in the data block are processed simultaneously, the starting address for every data point (except the first one) will remain undetermined at Step 4 until the threads communicate their number of mid-bytes. The same dependency exists at Step 3 in Figure 10, when the threads in the decompressor want to read the mid-bytes.

Solution 1: We add a prefix-scan step before Step 4 in Figure 9 and Step 3 in Figure 10. The prefix-scan can be parallelized on GPUs

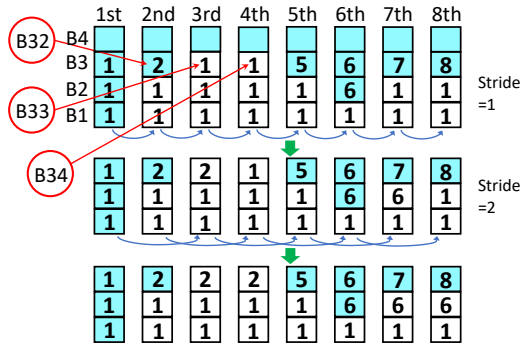


Figure 11: Schematic graph showing the index propagation for parallel leading-byte retrievals

by using two-level in-warp shuffles [24]. With this step, all threads can sync and communicate to efficiently find their own starting address.

Challenge 2: Step 2 in both Figures 9 and 10 depend on the value of the previous data point. The two data dependencies have different depths. In compression, the previous data point's value is known to the current thread since it can be directly read from the input original data. Hence the depth is 1. In decompression, however, the previous data point's value remains unknown to the current thread since it is simultaneously being retrieved by the neighbor thread. A direct read of the previous data value by the current thread will cause the read-after-write (RAW) hazard. Accordingly, the data dependencies in the decompression yield dependence chains. A chain starts at the mid-byte right before a leading byte and keeps extending until reaching the next mid-byte.

We provide a leading byte retrieval example as shown in the first row of Figure 11 to further explain the RAW hazard and dependence chain. In this example, the data block contains eight floating-point data points. The mid-bytes of each data point are highlighted in blue. Since the fourth byte (B4) of every data point is mid-byte, it will not affect the data dependency. We consider the retrievals of the third byte of the second (B32), third (B33), and fourth (B34) data points. In sequential implementation, B33 can be retrieved by reading the mid-byte B32, and B34 then can be retrieved by reading B33. Consequently, we get the correct values B34=B33=B32.In the parallel context, however, retrieving B34 by reading B33 will get an undefined value since B33 is also a leading byte and will cause the RAW hazard. B32, B33, and B34 form a dependence chain, since B34 also needs to get the correct value from B32. The same issue will occur when simultaneously retrieving B27 and B28.

Solution 2: Simply letting each CUDA thread read both the current data point and the adjacent preceding data point from the input original data can break the data dependency during the compression, since the dependency depth is only 1.

The essential factor in breaking the data dependency in the decompression is to identify the dependence chains. Then each leading byte knows where to read for retrieving the value. For example, B34 and B33 know they should retrieve their value from B32 after the B32-B33-B34 chain is identified. To efficiently identify the chains in parallel fashion, we propose an *index-propagation* approach. It assigns each byte an initial index as shown in Figure 11 (note that they are not byte values). All leading bytes will get an initial index 1, while the mid-bytes will get their actual index (e.g., 2 for B32). We ignore the B4s since they are all mid-bytes. Then a parallel propagation is performed in recursive doubling style to propagate the indices. In the running example, during the first round, each thread propagates its bytes' indices to its adjacent thread (stride=1) using warp-level shuffles. For each byte in a thread, its own index will be overwritten if the corresponding received index is greater. For instance, B33's initial index is 1, and it will be overwritten by B32's index, which is 2 after the first round propagation. In the second round, the indices are propagated with stride=2 and follow the same overwriting rule. After the third round with stride=4, the index propagation of the running example finishes. Notice that we do not display the last round in Figure 11 because it does not change the final indices.

In the final indices result, the bytes of consecutive data points that have the same indices indicate a dependence chain. For example, in the last row of Figure 11, B32, B33, and B34 are the bytes of three consecutive data points, and they all have an index 2, so they form a dependence chain. We can observe that our *index-propagation* successfully identifies all dependence chains in the running example. Then each leading byte can retrieve its value according to its final value (e.g., 2 in B34 indicates its value should be read from B32).

The *index-propagation* approach is inexpensive. It requires only the lightweight shuffle operations. Furthermore, with the recursive doubling, the parallel propagation complexity is reduced from O(n) to O($\log n$) (e.g., three propagations can propagate eight data points in the running example).

7 PERFORMANCE EVALUATION

In this section we analyze the evaluation results, which are performed by using six real-world scientific datasets on heterogeneous devices on two different supercomputers.

7.1 Experimental Setup

Table 2 describes all the application datasets used in our experiments. The datasets are downloaded from the well-known Scientific Data Reduction Benchmark website [5].

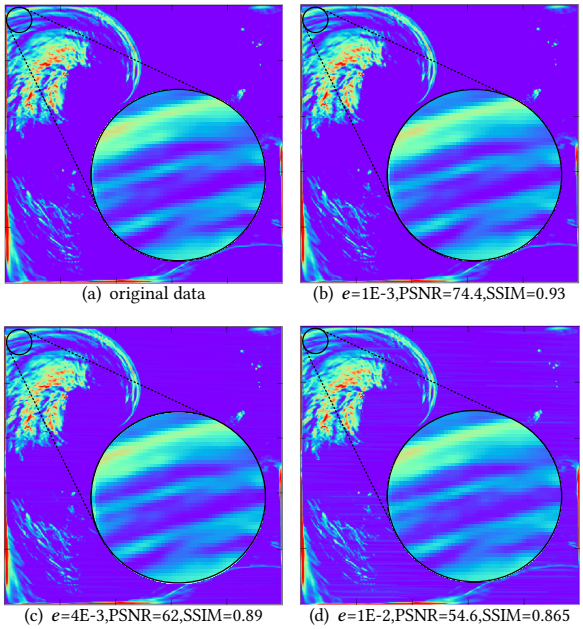
We perform our GPU experiments on both an A100 GPU (offered by ANL ThetaGPU [20]) and V100 GPU (offered by ORNL Summit [28]). NVIDIA's Ampere microarchitecture is the successor of the Volta microarchitecture. V100 has 80 streaming multiprocessors (SMs) with 64 CUDA cores per SM (total of 5,120 cores device-wide), while A100 has 108 SMs and 6912 CUDA cores in total. We compare our developed ultrafast compressor SZx with two lossy compressors—SZ [12, 29] and ZFP [22]; these are arguably the fastest existing error-bounded compressors based on prior studies [12, 43], and they both have GPU versions that can be compared with our solution SZx in the experiments.

7.2 Evaluation Results

First, we check the data reconstruction quality under our SZx for all the simulation datasets involved in our experiments. We conclude that the overall visual quality looks great when the value-range-based error bound (denoted by *REL*) is set to 1E-2~1E-4 for SZx. Because of space limits, we demonstrate the visual quality, PSNR,

Table 2: Applications (all datasets here are originally stored in single-precision floating point)

Application	# of fields	Size per field	Description
CESM-ATM (CE.) [17]	77	1800×3600	Atmosphere simulation of Community Earth System Model
Hurricane (Hu.) [1]	13	100×500×500	simulation of Hurricane ISABEL
Miranda (Mi.) [3]	7	256×384×384	large-eddy simulation of multi-component flows with turbulent mixing
Nyx (Ny.) [4]	6	512×512×512	adaptive mesh, massively parallel cosmological simulation
QMCPack (QM.) [18]	2	288/816×115×69×69	simulation for electronic structure of atoms, molecules and solids
SCALE-LetKF (SL.) [2]	12	98×1200×1200	SCALE-RM weather simulation based on LETKF filter



(a) original data (b) e=1E-3,PSNR=74.4,SSIM=0.93

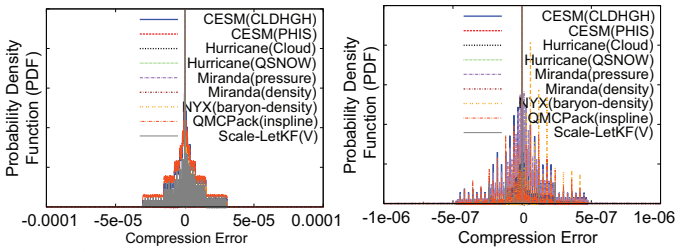
(c) e=4E-3,PSNR=62,SSIM=0.89 (d) e=1E-2,PSNR=54.6,SSIM=0.865

Figure 12: Visual quality of SZx on Hurricane-ISABEL simulation (the compression ratios are 14.6, 18, 20.64, respectively)

and SSIM using only the Hurricane-ISABEL simulation dataset (CLOUDf48), as shown in Figure 12 (compression ratios are 14.6, 18, and 20.64, respectively). We can observe that the reconstructed data's visual quality is high, even zooming in the top-left corner by 50×, although a few artifacts can be seen in the dark blue area of Figure 12 (d). How to further mitigate or remove artifacts will be our future work.

Figure 13 presents the distribution of compression errors for different application datasets with the absolute error bounds of 1E-4 and 1E-6. Because of space limits, we present only a couple of fields for each application. All other fields of the same application exhibit similar results, based on our observation. We validate that SZx can always respect user-specified error bounds for all the data fields across different applications in our experiments, even with a very small error bound (e.g., 1E-6), as shown in Figure 13 (b).

We present compression ratios of our SZx as well as those of SZ and ZFP in Table 3, by showing the minimum, overall (i.e., harmonic mean), and maximum CR, respectively, for all the fields in each application. The table shows that SZx can get very high compression ratios (e.g., 124 for CESM) when REL=1E-2. Its overall compression ratio is 3~12 in all cases, which is 0.5~3× lower than that of ZFP and 3~30× lower than that of SZ. These results are reasonable because SZ and ZFP adopt advanced multidimensional data analysis and sophisticated encoding methods, which can get



(a) Error bound = 1E-4 (b) Error bound = 1E-6

Figure 13: Distribution of compression errors under SZx

fairly high compression ratios; however, the compressors may suffer lower execution performance on both CPUs and GPUs in turn (to be shown later). As a comparison, the overall compression ratio of the lossless compressor Zstd is only 1.12~1.49, which is lower than that of SZx by about 200~400%. Therefore, SZx could be the favorite of end users who desire higher CR than the lossless compression provides but are sensitive to the compression time.

To demonstrate the algorithmic efficiency of SZx, we first present the single-core CPU-based compression and decompression throughputs of the three compressors in Table 4 and Table 5, respectively. The numbers shown in the tables are the overall performance considering all the fields for each application. Through the tables, we can observe that our SZx significantly outperforms the other two error-bounded lossy compressors in terms of both compression and decompression speeds. Specifically, for compression, SZx is ~2.5~5× faster than ZFP and ~5~7× faster than SZ, while for decompression, SZx is about 2~4× as fast as both ZFP and SZ. As we elaborated in the algorithm design and optimization sections, SZx achieves such a high performance thanks to the lightweight skeleton design described in Algorithm 1 and the bitwise right-shifting strategy proposed in Section 5.1.

We next compare the performance of the OpenMP-based multicore CPU implementation for SZx, ZFP, and SZ. Tables 6 and 7 display the compression and decompression throughput, respectively. We set the number of threads to 64 since we empirically find it is the optimal setting for all three compressors. Some data are not available because (1) the OpenMP version of SZ (omp-SZ) does not support 2D data (i.e., CESM) and (2) the OpenMP version of ZFP (omp-ZFP) has no decompressor implementation. We observe from the tables that, although the speedup varies according to dataset, our OpenMP version of SZx (omp-SZx) always shows the best performance. Specifically, our omp-SZx compressor can achieve 3.4~6.8× and 2.4~4.8× speedup compared with that of the omp-ZFP and omp-SZ counterparts, and our omp-SZx decompressor achieves 2.3~4.6× speedup compared with that of the omp-SZ decompressor.

Table 3: Compression Ratios (Original Data Size / Compressed Data Size)

		CESM			Hurricane			Miranda			Nyx			QMCPack			Scale-LetKF		
	REL	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
SZx	1E-2	4	9.1	124	4	6.6	21.1	8.2	11.8	16.2	4.8	11.34	124	9.2	9.4	9.7	7.6	10.6	25.2
	1E-3	2.84	4.61	19.3	2.9	4	17.6	4.5	7.2	12.5	3.2	5.9	119	4.3	4.4	4.4	3.65	4.7	7.8
	1E-4	2.14	3.3	17	2.1	3	16.2	2.7	4.5	9.5	2.4	3.7	75	2.9	2.9	2.9	2.7	3.14	5.6
ZFP	1E-2	8	13.6	46	6.4	11.3	25.8	30.5	46.6	74.6	22.5	38.8	1.1k	39.1	39.2	39.4	9.4	14.5	23.8
	1E-3	4.3	7.9	30	4.3	6.7	13.2	20.6	25.6	38.5	8.2	13.1	150	21	21.1	21.2	6.4	7.8	13.4
	1E-4	3	5.1	18.8	2.9	4.32	10.4	11	14.5	22.9	4.1	6.2	74	10.3	10.3	10.4	3.9	4.6	7.7
SZ	1E-2	34.4	151	3k	20.4	49.8	339	92.8	126	234	263	507	21k	201	213	227	26.3	84	746
	1E-3	15.6	151	840	9.24	17.5	58.8	49.6	59.5	75.2	36.7	79	3.6k	52	54.3	56.8	18.9	26.5	140
	1E-4	6.4	38.3	104	5.6	9.8	31	25.1	29.6	35	10.3	18.2	621	18.9	19.2	19.6	10	13.9	23.1
zstd	-	1.03	1.44	17.1	1.08	1.49	19.56	1.6	1.21	4.86	1.08	1.12	1.14	1.18	1.19	1.2	1.08	1.37	2.95

Table 4: Compression Throughput on Single-Core CPU (MB/s)

	REL	CE.	Hu.	Mi.	Ny.	QM.	SL.
SZx	1E-2	1034	796	959	1087	969	1032
	1E-3	822	750	833	877	902	703
	1E-4	752	662	807	722	813	663
ZFP	1E-2	392	256	249	418	323	258
	1E-3	288	213	211	284	275	208
	1E-4	234	181	280	226	208	174
SZ	1E-2	236	193	186	258	205	217
	1E-3	170	153	161	229	216	156
	1E-4	143	130	139	164	147	124

Table 5: Decompression Throughput on Single-Core CPU (MB/s)

	REL	CE.	Hu.	Mi.	Ny.	QM.	SL.
SZx	1E-2	1221	1085	1950	1450	1292	1408
	1E-3	1022	1006	1546	1218	1083	975
	1E-4	925	864	1319	956	928	886
ZFP	1E-2	485	476	498	732	685	360
	1E-3	327	371	401	455	524	395
	1E-4	246	297	327	333	376	299
SZ	1E-2	559	451	549	635	588	519
	1E-3	381	291	444	534	462	334
	1E-4	269	229	392	359	282	236

Table 6: Compression Throughput on a Multicore CPU (GB/s)

	REL	CE.	Hu.	Mi.	Ny.	QM.	SL.
SZx	1E-2	4.38	6.89	9.13	7.25	9.69	8.57
	1E-3	3.77	6.32	8.53	6.55	8.11	7.51
	1E-4	3.74	6.06	8.54	6.68	7.77	7.34
ZFP	1E-2	0.74	1.52	2.70	1.49	2.23	1.83
	1E-3	0.61	1.31	2.42	1.36	1.87	1.58
	1E-4	0.55	1.09	1.87	1.23	1.46	1.32
SZ	1E-2	n/a	1.80	1.99	2.12	3.60	2.90
	1E-3	n/a	1.49	1.82	2.12	3.49	2.85
	1E-4	n/a	1.67	1.78	1.95	3.23	2.54

Subsequently, we evaluated the GPU performances of cuSZx, cuZFP, and cuSZ on two cutting-edge supercomputers—ANL ThetaGPU (A100) and ORNL Summit (V100), respectively. We note that both cuZFP and cuSZ have also been deeply optimized with respect to the GPU architecture by their developers [10, 31, 32]. The compression and decompression performance results regarding all the

Table 7: Decompression Throughput on a Multicore CPU (GB/s) (ZFP's results are all n/a because it does not support multithread decompression)

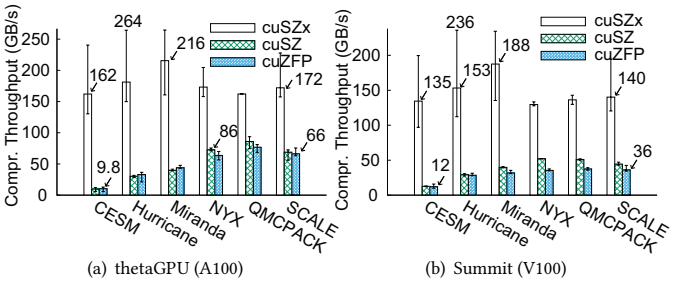
	REL	CE.	Hu.	Mi.	Ny.	QM.	SL.
SZx	1E-2	1.93	3.89	5.32	11.34	16.01	11.36
	1E-3	1.51	3.98	5.91	11.84	15.08	10.78
	1E-4	1.31	3.91	5.41	11.52	14.53	11.39
ZFP	1E-2	n/a	n/a	n/a	n/a	n/a	n/a
	1E-3	n/a	n/a	n/a	n/a	n/a	n/a
	1E-4	n/a	n/a	n/a	n/a	n/a	n/a
SZ	1E-2	n/a	1.73	2.11	2.84	4.90	3.01
	1E-3	n/a	1.67	2.09	2.87	4.32	2.92
	1E-4	n/a	1.48	1.98	3.08	3.92	2.48

fields of each application are presented in Figure 14 and Figure 15, respectively.

According to Figure 14, the peak compression performance of SZx can reach up to 264 GB/s (see Hurricane ISABEL's result in Figure 14 (a)). The overall compression performance of SZx is 150~216 GB/s on ThetaGPU and 140~188 GB/s on Summit. As a comparison, both cuSZ and cuZFP suffer from very low GPU performance (9.8~86GB/s on ThetaGPU and 12~52GB/s on Summit). Moreover, based on Figure 15, we can observe that the peak decompression performance of SZx can reach up to 446 GB/s (see Miranda's result in Figure 15 (a)). The overall decompression performance of SZx is 150~291 GB/s on ThetaGPU and 120~243 GB/s on Summit. As a comparison, both cuSZ and cuZFP suffer from much lower decompression performance (9.7~67GB/s on ThetaGPU and 13.7~48 GB/s on Summit).

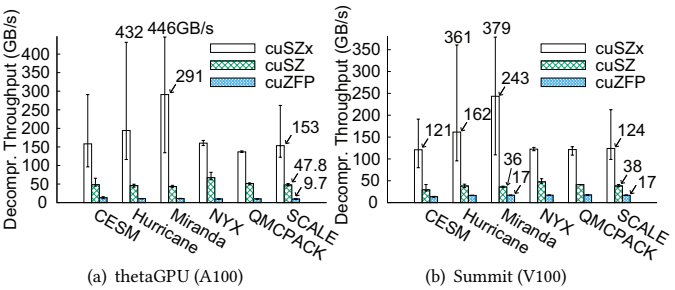
We can observe that our cuSZx achieves higher speedups than does the single-core CPU SZx (16× vs. 7×), compared with their respective counterparts. The reason is that some complex steps in SZ and ZFP, for example, Huffman decoding [26], are extremely irregular and unfriendly to the GPU execution model. On the other hand, the simplified design of our SZx significantly reduces the parallelization complexity and is friendlier to GPUs. Consequently, with the optimizations described in Section 6.2, our cuSZx significantly outperforms the highly optimized cuSZ and cuZFP. We also emphasize that our cuSZx preserves the same compression ratio as SZx does, since it makes no change to Algorithm 1.

We now evaluate the overall data dumping/loading performance on ANL ThetaGPU nodes with different execution scales. Specifically, for the data dumping experiment, we use an MPI code to



(a) thetaGPU (A100) (b) Summit (V100)

Figure 14: Overall compression throughput per GPU (GB/s)



(a) thetaGPU (A100) (b) Summit (V100)

Figure 15: Overall decompression throughput per GPU (GB/s)

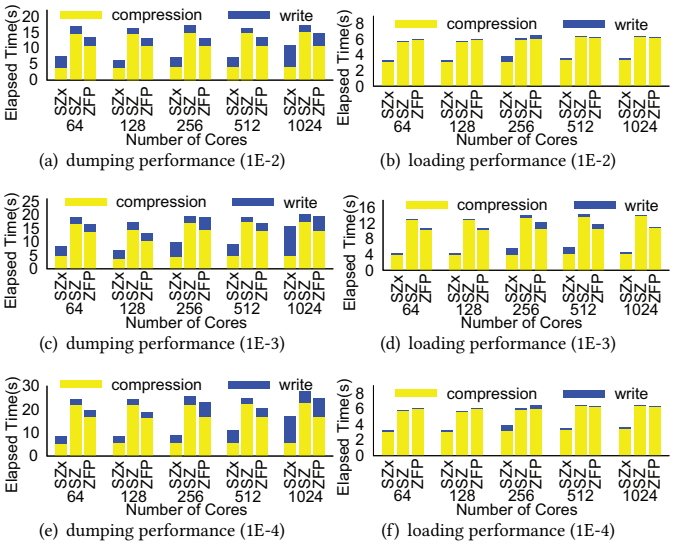
launch 64~1024 ranks/cores, each performing lossy compression using the Nyx dataset and writing compressed data onto the parallel file system. For the data-loading experiment, each MPI rank reads the compressed data from PFS and then performs decompression. We present the performance breakdown in Figure 16 in terms of different value-range-based error bounds.

In the figure, we can clearly observe that SZx obtains the highest overall performance in both data dumping and data loading on ThetaGPU. In particular, the solution with SZx takes only $\frac{1}{3}$~$\frac{1}{2}$ time to dump or load data compared with other solutions in most cases. That is, the I/O performance is improved by 100%~200% under SZx. The key reason is that ThetaGPU has a relatively high I/O bandwidth, so that the overhead at the compression/decompression stage turns out to be the key bottleneck at the execution scales of our experiments.

8 CONCLUSION AND FUTURE WORK

In this paper we propose an ultrafast error-bounded lossy compression framework SZx. It is designed for scenarios where the long compression time of current lossy compressors is not acceptable and a higher compression ratio than provided by the lossless compressors is still demanded. We rigorously confine the design of SZx to use only super-lightweight calculations such as addition, subtraction, and bitwise operations. We perform comprehensive evaluations of both the CPU- and GPU-based implementations using six real-world datasets and two cutting-edge supercomputers. The key insights are summarized as follows.

- With the same error bound, SZx has reasonably lower compression ratios than ZFP and SZ do (0.3~3× lower than ZFP and 3~30× lower than SZ) because it has no sophisticated data prediction/transform step and no expensive encoding algorithms such as Huffman encoding.



(a) dumping performance (1E-2) (b) loading performance (1E-2)

(c) dumping performance (1E-3) (d) loading performance (1E-3)

(e) dumping performance (1E-4) (f) loading performance (1E-4)

Figure 16: Data dumping/loading performance on ThetaGPU (Nyx dataset)

- On a single-core CPU with the same error bound, SZx is 2.5~5× faster than ZFP and 5~7× faster than SZ in compression; SZx is 2~4× as fast as both SZ and ZFP in decompression.
- On a multicore CPU with the same error bound, SZx is 3.4~6.8× and 2.4~4.8× faster than ZFP and SZ in compression, while it can achieve 2.3~4.6× speedup compared with SZ in decompression.
- On a GPU with the same error bound, SZx's peak performance in compression and decompression on single GPU can reach up to 264 GB/s and 446 GB/s, respectively. These results are 2~16× as fast as SZ and ZFP on GPUs.
- When compressing&writing compressed data to PFS or reading&decompressing compressed data from PFS on ANL ThetaGPU with 64~1024 cores, the overall data dumping/loading performance under SZx is higher than that with SZ or ZFP by 100%~200%, because of SZx's fairly high performance.

In future work, we plan to explore how to further improve compression ratios of SZx. We also plan to quantitatively characterize the trade-off between the compression ratio and the performance [16] of our SZx.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. This research was also supported by ARAMCO. The material was supported by the U.S. Department of Energy, Office of Science and Office of Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357. This research was also supported by the U.S. National Science Foundation under Grants OAC-2042084, OAC-2003709, OAC-2104023, and OAC-2104024.

REFERENCES

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

[2] [n. d.]. The Local Ensemble Transform Kalman Filter (LETKF) data assimilation package for the SCALE-RM weather model. https://github.com/gylien/scale-letkf. Online.

[3] [n. d.]. Miranda turbulence simulation. https://wci.llnl.gov/simulation/computer-codes/miranda. Online.

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

[5] [n. d.]. Scientific Data Reduction Benchmark. https://sdrbench.github.io/. Online.

[6] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2018. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Computing and Visualization in Science* 19, 5 (01 Dec 2018), 65–76.

[7] Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. 2018. TTHRESH: Tensor Compression for Multidimensional Visual Data. *CoRR* abs/1806.05952 (2018). arXiv:1806.05952 http://arxiv.org/abs/1806.05952

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

[9] Yann Collet. 2015. Zstandard – Real-time data compression algorithm. *http://facebook.github.io/zstd/* (2015).

[10] cuZFP. 2020. https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp. Online.

[11] L Peter Deutsch. 1996. GZIP file format specification version 4.3.

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

[13] Sheng Di, Dingwen Tao, Xin Liang, and Franck Cappello. 2019. Efficient Lossy Compression for Scientific Data Based on Pointwise Relative Error Bound. *IEEE Transactions on Parallel and Distributed Systems* 30, 2 (2019), 331–345. https://doi.org/10.1109/TPDS.2018.2859932

[14] Ali Murat Gok, Sheng Di, Yuri Alexeev, Dingwen Tao, Vladimir Mironov, Xin Liang, and Franck Cappello. 2018. PaSTRI: Error-Bounded Lossy Compression for Two-Electron Integrals in Quantum Chemistry. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11. https://doi.org/10.1109/CLUSTER.2018.00013

[15] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Venkatram Vishwanath, Tom Peterka, Joe Insley, et al. 2016. HACC: Extreme scaling and performance across diverse architectures. *Commun. ACM* 60, 1 (2016), 97–104.

[16] Dewan Ibtesham, Dorian Arnold, Patrick G Bridges, Kurt B Ferreira, and Ron Brightwell. 2012. On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. In *2012 41st international conference on parallel processing*. IEEE, 148–157.

[17] JE Kay, C Deser, A Phillips, A Mai, C Hannay, G Strand, JM Arblaster, SC Bates, G Danabasoglu, J 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.

[18] Jeongnim Kim and et al. 2018. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. 30, 19 (apr 2018), 195901.

[19] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. 2011. Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 366–379.

[20] LCRC. 2021. ThetaGPU Machine Overview. https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview. Online.

[21] 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*. IEEE.

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

[23] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 1245–1250.

[24] Mark Harris, Shubhabrata Sengupta and John D. Owens. [n. d.]. Parallel Prefix Sum (Scan) with CUDA.

[25] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP?. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[26] Cody Rivera, Sheng Di, Jiannan Tian, Xiaodong Yu, Dingwen Tao, and Franck Cappello. 2022. Optimizing Huffman Decoding for Error-Bounded Lossy Compression on GPUs. *arXiv preprint arXiv:2201.09118* (2022).

[27] SLAC National Accelerator Laboratory. 2017. Linac Coherent Light Source (LCLS-II). https://lcls.slac.stanford.edu/. Online.

[28] Summit. [n. d.]. https://www.olcf.ornl.gov/summit/.

[29] 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*. IEEE, 1129–1139.

[30] 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. https://doi.org/10.1177/1094342017737147

[31] Jiannan Tian 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 (PACT '20)*. 3–15.

[32] Jiannan Tian, Sheng Di, Xiaodong Yu, Cody Rivera, Kai Zhao, Sian Jin, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data on GPUs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 283–293.

[33] Robert Underwood, Sheng Di, Jon C. Calhoun, and Franck Cappello. 2020. FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data. https://arxiv.org/abs/2001.06139. Online.

[34] Zang Wang, Alan C. Bovick, Hamid R. Sheikh, and Eero P. Simoncelli. [n. d.]. The SSIM Index for Image Quality Assessment. https://www.cns.nyu.edu/~lcv/ssim/

[35] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T. Chong. 2019. Full-State Quantum Circuit Simulation by Using Data Compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. Association for Computing Machinery, New York, USA, Article 80, 24 pages.

[36] Xiaodong Yu and Michela Becchi. 2013. Exploring different automata representations for efficient regular expression matching on GPUs. *ACM SIGPLAN Notices* 48, 8 (2013), 287–288.

[37] Xiaodong Yu, Tekin Bicer, Rajkumar Kettimuthu, and Ian Foster. 2021. Topology-aware optimizations for multi-GPU ptychographic image reconstruction. In *Proceedings of the ACM International Conference on Supercomputing*. 354–366.

[38] Xiaodong Yu, Sheng Di, Ali Murat Gok, Dingwen Tao, and Franck Cappello. 2021. cuZ-checker: A GPU-Based Ultra-Fast Assessment System for Lossy Compressions. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 307–319.

[39] Xiaodong Yu, Viktor Nikitin, Daniel J Ching, Selin Aslan, Doğa Gürsoy, and Tekin Biçer. 2022. Scalable and accurate multi-GPU-based image reconstruction of large-scale ptychography data. *Scientific Reports* 12, 1 (2022), 1–16.

[40] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. 2017. An enhanced image reconstruction tool for computed tomography on GPUs. In *Proceedings of the Computing Frontiers Conference*. 97–106.

[41] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. 2019. GPU-based iterative medical CT image reconstructions. *Journal of Signal Processing Systems* 91, 3 (2019), 321–338.

[42] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng (Daphne) Yao. 2020. GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting. In *The 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[43] 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 (HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 89–100.

[44] Zlib. [n. d.]. https://www.zlib.net/. Online.