SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5554-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SCW63240.2024.00042

SZOps: Scalar Operations for Error-bounded Lossy Compressor for Scientific Data

Tripti Agarwal*, Sheng Di†, Jiajun Huang‡, Yafan Huang §, Ganesh Gopalakrishnan*,
Robert Underwood†, Kai Zhao¶, Xin Liang‖, Guanpeng Li†, Franck Cappello†
*University of Utah, Salt Lake City, UT, USA
†Argonne National Laboratory, Lemont, IL, USA
‡University of California, Riverside, CA, USA
§University of Iowa, Iowa City, IA, USA
¶Florida State University, Tallahassee, FL, USA
‖University of Kentucky, Lexington, KY, USA
tripti.agarwal@utah.edu, sdi1@anl.gov, jhuan380@ucr.edu, yafan-huang@uiowa.edu, ganesh@cs.utah.edu,
runderwood@anl.gov, kzhao@cs.fsu.edu, xliang@uky.edu, guanpeng-li@uiowa.edu, cappello@mcs.anl.gov

***Abstract*—Error-bounded lossy compression has been a critical technique to significantly reduce the sheer amounts of simulation datasets for high-performance computing (HPC) scientific applications while effectively controlling the data distortion based on user-specified error bound. In many real-world use cases, users must perform computational operations on the compressed data. However, none of the existing error-bounded lossy compressors support operations, inevitably resulting in undesired decompression costs. In this paper, we propose a novel error-bounded lossy compressor (called SZOps), which supports not only error-bounding features but efficient computations (including negation, scalar addition, scalar multiplication, mean, variance, etc.) on the compressed data without the complete decompression step, which is the first attempt to the best of our knowledge. We develop several optimization strategies to maximize the overall compression ratio and execution performance. We evaluate SZOps compared to other state-of-the-art lossy compressors based on multiple real-world scientific application datasets.**

***Index Terms*—Error-bounded Lossy Compression, Scientific Application**

I. Introduction

Today's scientific applications tend to be running on extremely large execution scales, which may easily produce sheer amounts of simulation datasets that need to be kept in memory or stored in disks with limited storage capacity. Climate simulations, for example, may produce 200+ TB of data within 16 seconds [1], and Fusion simulations can generate over 200 PB of data in a single run [2]. Such a large volume of simulation datasets may cause serious issues in data storage and transfer because of the limited storage space and data movement bandwidth (such as network, I/O, and memory).

Error-bounded lossy compression [3]–[10] has been proposed for years to resolve the above issues, especially because it can get fairly high compression ratios while strictly controlling the data distortion based on user-required error bound. For example, SZ and ZFP have been effective in significantly improving the I/O data writing performance, as shown in [11]. MDZ [12] can be used to substantially reduce the storage size for Molecular Dynamics simulations while preserving the radio distribution function (RDF) very well. Wu et al. [13] developed an efficient lossy compression algorithm that can effectively compress the memory footprint for quantum computing simulations at runtime, which can significantly lower the requirement of memory capacity. Error-bounded lossy compression (e.g., Ocelot [14]) has also been used to improve the data transfer on a wide area network (WAN). Some general-purpose lossy compressors [8], [15] can significantly reduce the scientific data size, though they may suffer relatively low compression speed. FAZ, for example, can compress large turbulence simulation data (Miranda [16]) and seismic data (RTM [17]) by 93.6× and 514×, respectively, at the relative error bound of 10^{-4} (a.k.a., 1E-4). Such compressors are very helpful in the use-case with very limited storage capacity or data transfer bandwidth.

In addition to the above use cases which mainly make use of lossy compression to reduce storage size or mitigate data transfer cost, quite a few emerging use cases require performing certain operations on top of the compressed data. The existing compression methods, however, do not support performing various operations on the compressed data, so the users have to decompress the full dataset before executing the operations, inevitably introducing a high execution cost. For example, quantum circuit simulation [13]) may produce an extremely large amount of data to keep in memory, so it needs to compress the data to control the memory footprint. The data stored in the compressed format may need to be decompressed upon the need of simulation at runtime, which requires extra decompression steps inevitably for the traditional compressors. If a compressor supports performing operations on the compressed data, the extra decompression cost can be saved or minimized, which can thus improve the overall performance in turn.

In this paper, we develop a compression mechanism allowing to perform various arithmetic operations (such as

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

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00042

negation, scalar addition, and scalar multiplication) and reductions (such as mean, variance and standard-deviation) on the error-bounded compressed datasets without expensive full decompression.

As mentioned previously, operations on error-bounded lossy compression is very helpful in many emerging use cases such as reducing memory footprint and avoiding expensive decompression costs because of avoiding the full decompression step. For example, using lossy compression to reduce the communication cost for accelerating the overall MPI collective operation performance [18]. In the existing solutions, each process participating in the collaborative operation needs to fully decompress the compressed data received from another process, execute an arithmetic aggregation/reduction operation (such as addition), and then perform another compression on the aggregated data. This results in higher decompression time and more memory footprint, which can be reduced by applying operations directly compressed data.

Developing an efficient operation mechanism on error-bounded lossy compression is very challenging because of diverse compressor designs. In general, each existing error-bounded lossy compressor involves multiple steps from the data decorrelation to lossless encoding. For example, ZFP decorrelates the data by a blockwise near-orthogonal transform and SZ leverages various data prediction methods to do it. In order to reach high compression ratios, the lossy compressors often depend on sophisticated lossless encoders. For instance, ZFP adopts an embedded encoding and SZ chooses to use Huffman encoding + Zstd [19]. Supporting the arithmetic operations in the presence of encoding techniques is quite non-trivial.

Our proposed novel error-bounded lossy compression mechanism supports scalar operations, which is the first attempt in the error-bounded lossy compression community to the best of our knowledge. The fundamental idea is designing an efficient, lightweight compression pipeline that takes into account the execution of potential operations on the compressed data, and also minimizes the required steps and cost in the decompression in terms of performing the user-specified operations. We summarize the key contributions as follows:

- We develop an efficient error-bounded lossy compression method, which supports scalar operations on compressed data.
- We carefully optimize the pipeline to support various arithmetic operations (negation, scalar addition, scalar subtraction, scalar multiplication, mean, variance and standard deviation) on compressed datasets without the full decompression.
- We perform a comprehensive evaluation based on multiple real-world scientific datasets for our compressor to show that our compressor can improve the execution performance ranging from 2.17× to more than 206× when compared with traditional workflow for various scalar operations across different datasets.

The remainder of the paper is organized as follows. Section II discusses the related works. We formulate the research problem in Section III. We present the compression pipeline in Section IV. We describe the scalar operations integrated with the compression as well as the optimizations in Section V. In Section VI, we provide the evaluation results as well as the analysis. In the end, we conclude the paper with a vision of the future work in Section VII.

II. Related Work

Error-bounded lossy compressors, such as ZFP [6] and SZ [3], [4], are often used for scientific data compression and can achieve a high compression ratio such as 50 or more, while strictly controlling the data distortion. None of the lossy compression techniques, however, were built with the goal of operations on compressed data. That is, if the users want to operate on data that has been compressed, they have to first fully decompress the data and then perform the operation. This will inevitably introduce undesired execution costs and memory overhead.

There exist some compression techniques that support operations like Blaz [20] and PyBlaz [21]. Blaz is a simple compressor that can only support 2-D arrays and can perform simple operations such as scalar addition, matrix addition, and multiplication of scalar. Since Blaz is a single-threaded sequential code, PyBlaz was built to support a more sophisticated compression setting. PyBlaz can support arbitrary dimensional data along with a lot more operations and measures. However, none of the above works provide a guarantee of compression error boundness for both the compression pipeline and operation built on top of them.

In our work, we mainly work with error-bounded lossy compressors, hence we provide a detailed literature survey of such compressors. These state-of-the-art error-bounded lossy compressors can be split into three models.

Prediction-based lossy compression model. This compression model generally involves three key stages: data prediction, quantization, and lossless compression. The typical examples include SZ1 [3], SZ2 [4], SZ3 [22], [23], and FPZIP [24]. SZ2, for example, adopts a hybrid prediction method combining Lorenzo predictor [25] and linear regression. The biggest advantage of the prediction-based lossy compression model is that it can be easily/efficiently customized by changing the specific solutions for each stage to adapt to different application datasets or use-cases [8], [12], [26]–[28]. MDZ [12], for example, is a error-bounded lossy compressor tailored based on Molecular Dynamics (MD) datasets under the SZ prediction-based framework. CliZ [27] is an efficient error-bounded prediction-based lossy compressor optimized for climate datasets based on the climate data properties/features.

Transform-based lossy compression model. The key idea of this compression model is performing data transform (such as wavelet transform) to convert the raw dataset to another coefficient dataset. ZFP [6] and SPERR [15] are two examples that uses this technique. This step can effectively decorrelate the raw dataset, such that a large majority of the data values are very close to 0. Then, an encoding method would be applied to significantly reduce the data size. ZFP, for example, applies

a so-called big-plane-based embedded encoding method [6] which stores only necessary bits with respect to user-required error bounds; SPERR performs a set partitioning embedded block coding algorithm (called SPECK [29]) to shrink the coefficient data size.

Higher-order singular value decomposition (HOSVD) based compression model. HOSVD [30], [31] decomposes the data (i.e., a tensor) to a set of matrices and a small core tensor, with well-preserved L_2 normal error. By combining HOSVD and Tucker decomposition with other techniques such as bit-plane, run-length, and/or Core tensor arithmetic coding, the data size could be significantly reduced.

To the best of our knowledge, none of the existing error-bounded lossy compressors support direct operations. In this paper, we fill this gap by proposing a novel error-bounded lossy compressor namely, SZOps that can perform scalar operations and scalar reductions in a compressed data domain. Throughout the rest of the text, the term scalar operations is employed when referring to both scalar operations and scalar reductions collectively.. Nonetheless, when explicitly referring to reduction operations, the distinct term scalar reductions will be utilized to ensure clarity and precision.

III. Problem Formulation

We formulate the research problem as follows. Given a raw dataset (denoted by D_r), we denote the corresponding compressed data as c and the decompressed dataset as $\hat{D}_c$. For a error-bounded lossy compressor with operations (such as *SZOps*), it would perform the user-required scalar operation on the compressed data, leading to a new compressed data stream (denoted by z), whose corresponding decompressed dataset is denoted as $\hat{D}_z$. Basically, we focus on two types of operations: univariate operation (denoted $f(\cdot)$, such as negation or adding a constant) and univariate reductions (denoted $r(\cdot)$ such as mean, standard deviation of input compressed data).

Figure 1 illustrates the entire workflow for the two types of operations. In the traditional workflow (see Figure 1 (a)), the user needs to get the decompressed dataset ($\hat{D}_c$) based on the compressed data stream (c) before operating $f(\cdot)$ or $r(\cdot)$. In comparison, the compression (i.e., marked as the *new workflows* in the figure) allows users to execute this operation on top of the compressed data format (c) without fully decompressing c.

Based on the two types of operations: univariate and multivariate, the output can also be split into two types as follows.

- *Compression-as-output*: the output is another compressed data stream (z) with the specific operations already applied on top of the decompressed data – illustrated as the blue workflow in Figure 1.
- *Computation-as-output*: the output is a result based on a computation applied on the dataset (such as the mean, standard deviation or maximum value) – illustrated as the magenta workflow in Figure 1.

The key objective of the research is to develop an efficient error-bounded compressor, which can avoid the expensive full data decompression when performing various operations on top of the compressed data. It is worth noting that completely avoiding decoding during the operation or reduction is impossible. Instead, our design motivation/objective is to minimize the decoding work as much as possible by keeping only necessary steps concerning scalar operations.

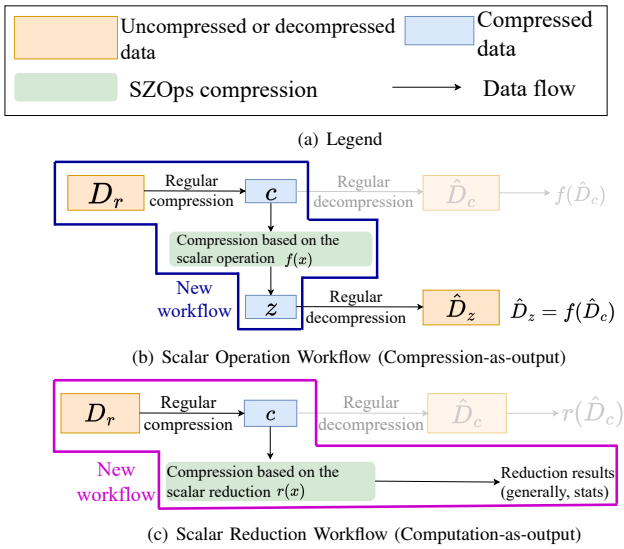


Fig. 1. Workflow For SZOps Compression

TABLE I
NOTATIONS USED IN SZOPS PIPELINE AND ITS OPERATIONS.

Notation	Description
D_r	Raw Data
c	Compressed Data
$\hat{D}_c$	Decompressed Data
z	Operated Compressed Data
$\hat{D}_z$	Operated Decompressed Data
$f(\cdot)$	univariate operation
$r(\cdot)$	univariate reductions
ϵ	user-defined error bound
$\mathbf{A}$	block 1 of some D_r (block_size = m'×n')
$\mathcal{O}_\mathbf{A}$	Outlier of block $\mathbf{A}$
$\varsigma_\mathbf{A}$	Sign array of block $\mathbf{A}$
$\varrho_\mathbf{A}$	Quantized array for block $\mathbf{A}$
$\mathcal{P}_\mathbf{A}$	Predicted array of block $\mathbf{A}$
$\mathcal{C}_\mathbf{A}$	Bits for compressed block $\mathbf{A}$

Table I summarizes all notations used in the paper, which helps discuss the design of our designed SZOps and scalar operations and reductions explained in Section IV and Section V.

IV. Design Overview

The SZOps mainly aims to support operations while still respecting error-bound features and expecting to reach relatively high compression ratios and high compression/decompression performance. Towards this end, we substantially improve the cuSZp compression pipeline [32] (which was initially designed for only GPU) by developing a new multi-threaded CPU version (we call it *SZp*) and enabling it to support

operations. Compared with classic SZ's original design [3], [4], our compression pipeline features higher compression and decompression speed, also being much more suitable for operations on compressed data, although with gracefully degraded compression ratios.

A. Compression Pipeline

SZOps is a floating-point data compressor consisting of three main steps: Quantization (QZ), Decorrelation (LZ), and Blockwise Fixed length byte Encoding (BF), as shown in Figure 2.

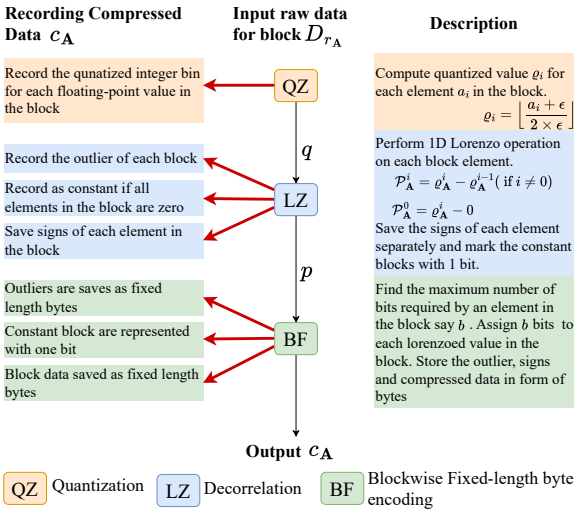


Fig. 2. SZOps compression pipeline (workflow). Decompression of SZOps is the inverse of all the steps.

In what follows, we describe the SZOps pipeline using 2D arrays, D_r, which can be extended to other dimensions (e.g., 1D and 3D) easily. Without loss of generality, the 2D array consists of $m \times n$ elements/data points. We split the dataset into blocks, and the block size is set to $m' \times n'$, which results in a total of $\frac{m}{m'} \times \frac{n}{n'}$ blocks of D_r. These blocks are denoted as $D_r^1, D_r^2, \ldots, D_r^{\frac{m}{m'} \times \frac{n}{n'}}$, for array D_r which can be compressed independently. For simplicity, without loss of generality, we describe our the compression pipeline using one specific block (D_r^1) from the dataset, respectively (as shown in Figure 2). We denote the elements in D_r^1 as $\mathbf{A} = a^i$, where $i = m' \times n'$ is the number of elements in the block and set the user-defined absolute error bound to ϵ.

We now explain each step of SZOps using block $\mathbf{A}$, with a few more new defined notations wherever necessary.

1) **Quantization (QZ):** This step converts the entire dataset into integers based on the user-defined error bound (ϵ). The quantized value (a.k.a., quantization bin number) is given by the Formula (1).

$$\varrho^i_{\mathbf{A}} = \left\lfloor \frac{a^i + \epsilon}{2 \times \epsilon} \right\rfloor \tag{1}$$

, where a^i is the i-th floating-point value in block $\mathbf{A}$ and $\lfloor \rfloor$ is a floor function. This step converts all the floating-point data into integer numbers (i.e., quantization bin numbers) because the transformed data are easier to process by lossless encoders such as Huffman encoding. Note that the data reproduced based on the quantization bins during the inversion of this step are not the same as the original data, and the data loss is limited within the error bound.

2) **Decorrelation (LZ):** In this step, we exploit that most scientific data are spatially adjacent-correlated (meaning that the data close to each other or within a region have similar value ranges). Hence, we apply a 1-D Lorenzo operator [33] on each block given in Formula (2). This helps in further decorrelating the integer values to reduce the necessary bits to store. We further store the *outlier* (**first value of each block**) separately, represented as $\mathcal{O}_{\mathbf{A}}$.

$$\begin{aligned} \mathcal{P}^i_{\mathbf{A}} &= \varrho^i_{\mathbf{A}} - \varrho^{i-1}_{\mathbf{A}} \qquad \forall i \neq 0 \\ \mathcal{P}^i_{\mathbf{A}} &= \varrho^i_{\mathbf{A}} - 0 \qquad \text{if } = 0 \end{aligned} \tag{2}$$

We use an example to explain decorrelation further. Suppose the block size for a 2-D input (D_r) is 2×2 and we have the following data values in the block $\mathbf{A}$ = {-0.025, -0.025, -0.051, -0.052}. Then, the quantized integers for each value are $\varrho_{\mathbf{A}} = \{-1, -1, -3, -3\}$. A 1D Lorenzo predictor (subtracting each value from its corresponding previous neighbor) is applied on the on $\varrho_{\mathbf{A}}$, resulting in predicted values $\mathcal{P}_{\mathbf{A}} = \{0, 0, -2, 0\}$ and the outlier $\mathcal{O}_{\mathbf{A}} = -1$.

We store the sign of each element separately, which removes the ambiguity that can occur during the succeeding lossless encoding step, i.e., fixed-length encoding. Positive numbers are represented with a bit value of 0, and negative are represented with a bit value of 1. Hence, $\mathcal{P}_{\mathbf{A}}$=$\{0, 0, 2, 0\}$, $\mathcal{O}_{\mathbf{A}}$=-1, and $\varsigma_{\mathbf{A}} = \{0, 0, 1, 0\}$, where $\varsigma_{\mathbf{A}}$ is the sign array for block $\mathbf{A}$.

3) **Blockwise Fixed-length byte encoding (BF):** The data obtained from the prediction $\mathcal{P}_{\mathbf{A}}$ is converted into bits using a fixed-length encoding technique. In this method, the maximum number of bits required for an integer in a given block $\mathbf{A}$ is calculated, and then all the numbers are stored with the same number of bits, represented by $\mathcal{C}_{\mathbf{A}}$. This reduces the number of bits (converted to bytes) required to store all the elements.

In the above example, for predicted array $\mathcal{P}_{\mathbf{A}} = \{0, 0, 2, 0\}$, the maximum number of bits taken by the block is 2 bits by integer 2, hence the entire block can be represented with 8 bits, i.e $\mathcal{C}_{\mathbf{A}} = (00001000)_2 = (8)_{16}$. Note that if all the elements in a $\mathcal{P}_{\mathbf{A}}$ (except for the outlier $\mathcal{O}_{\mathbf{A}}$) have integer value 0 for any block $\mathbf{A_i}$, we indicate such a block as a constant block and represent it with bit 0.

Finally, the compressed data is stored as follows: Fixed-length for each block followed by outlier for each block $\mathcal{O}_{\mathbf{A_j}}$, followed by sign bits for each element of all blocks $\varsigma^i_{\mathbf{A_j}}$ and then the compressed bits of each block $\mathcal{C}_{\mathbf{A_j}}$,

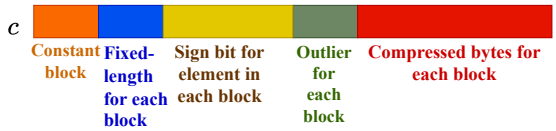
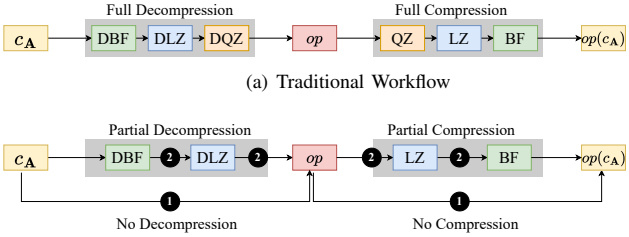


Fig. 3. Representation of compressed data

where is $j = \{1, 2, \ldots, \{\frac{m}{m'} \times \frac{n}{n'}\}\}$. Figure 3 shows a simple compressed data representation.

B. Workflow for Operations on Compressed data

The compression and decompression pipelines of SZOps are shown in Figure 4. The **traditional workflow operation** performs the full decompression (i.e., decompress the fixed-length encoded bytes for each block, then the inverse of Lorenzo operation, and finally the inverse of the quantization step). The desired operation is then applied to the decompressed data, and full compression, including quantization, Lorenzo, and block-wise fixed-length encoding, is again applied to the operated data to obtain the compressed format. In the **SZOps operation workflow**, the main idea is avoiding the full decompression and full compression on the operated data as discussed above in the traditional workflow. This involves skipping the steps of decompression and corresponding compression as required so that the operated compressed data is same as the operated compressed data obtained using traditional workflow.



(a) Traditional Workflow

(b) Scalar Operation/Reduction Workflow

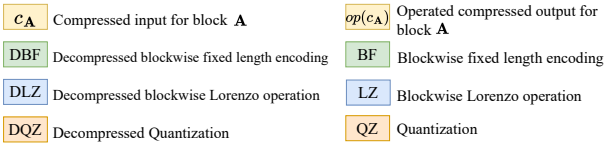


Fig. 4. Illustration of tradition workflow vs. SZOpz workflow for different scalar operations/reductions.

Depending on the datasets and operations, there are two different ways of performing the operations.

❶ *Directly performing operations on the input compressed data.* Operations like negation and addition of scalar on compressed data can be performed in a fully compressed space because the compressed data consists of signs and outliers saved separately, which can be used to calculate these operations.

❷ *Performing operations by first decompressing data using an inverse of blockwise fixed length encoding and inverting the Lorenzo operation.* Operations such as multiplication of scalar, mean, variance, standard deviation, use this workflow, and the operated data obtained is again compressed back by applying the Lorenzo operator and then performing blockwise fixed-length encoding. Details of how each of the operations is performed are explained in Section V.

TABLE II
LIST OF OPERATIONS IN SZOPS, ALONG WITH THE TYPE OF OPERATION AND THE RESULT TYPE OBTAINED AFTER THE OPERATION IS APPLIED. NOTE THAT, ALL THE OPERATIONS FOLLOW THE ERROR-BOUNDNESS. THIS IS BECAUSE NONE OF THE OPERATIONS APPLY INVERSE QUANTIZATION ON THE INPUT COMPRESSED DATA.

Type	Operation	Result Type
Univariate Operation	Negation	Compression-as-output
Univariate Operation	Scalar addition	Compression-as-output
Univariate Operation	Scalar subtraction	Compression-as-output
Univariate Operation	Scalar multiplication	Compression-as-output
Univariate Reduction	Mean	Computation-as-output
Univariate Reduction	Variance	Computation-as-output
Univariate Reduction	Standard Deviation	Computation-as-output

V. OPERATIONS AND REDUCTIONS FOR SZOPS

In this section, we discuss the different operations we developed in SZOps. In the following, we still mainly describe our design based on the blocks ($\mathbf{A}$) from the compressed dataset without loss of generality. After applying quantization and prediction, we obtain specific metadata: the outlier for block $\mathbf{A}$, denoted as $\mathcal{O}_{\mathbf{A}}$, the predicted value that are represented as array $\mathcal{P}_{\mathbf{A}}$, and the sign elements represented as array $\varsigma_{\mathbf{A}}$, respectively. We also use intermediate quantized values for some of the operations and denote the quantized value as $\varrho_{\mathbf{A}}$. Note that these quantized and predicted values are integers, which are subsequently stored as bytes using a fixed-length byte encoding scheme.

Using the above notations (also summarized in Table I), we explain different scalar operations supported by SZOps (listed in Table II) along with examples wherever necessary. Some operations are derivable from other operations; hence, we discuss those operations briefly.

A. Scalar Operations

Scalar Operations are point-wise operations that are performed on each element of the compressed dataset (or matrix). We describe each scalar operation available in SZOps here in detail.

1) **Negation:** Negation operation [34] is a unary operation and is solely dependent on reversing the signs of the data (saved explicitly in our compressed data), making the operation in fully compressed space[1]. Consider a single block array $\mathbf{A}$, then the negation operation is performed as follows: Invert the signs of each element in the array $\varsigma_{\mathbf{A}}$ to obtain the inverted signs, $\neg\varsigma_{\mathbf{A}}$. This is done by applying a logical NOT operation ($\neg$) element-wise: $\neg\varsigma_{\mathbf{A}} = \{\neg\varsigma_{a_0}, \neg\varsigma_{a_1}, ..., \neg\varsigma_{a_{\{m' \times n'\}-1}}\}$.
2) **Scalar Addition:** The scalar addition [34] involves adding a constant scalar value to an input array. In our

[1]Fully compressed space means that the compressed bits saved in our compressed data are not even partially decompressed

compressor, this is done by calculating the quantized bin index of the scalar s based on the user-defined error (let the quantized bin index be ϱ_s) and then by adding the scalar value to the outliers $\mathcal{O}$ of each block. Since we save the $\mathcal{O}$ separately, this operation is also performed in a fully-compressed space.
Suppose we want to add a value say 0.67 to $\mathbf{A}$. The quantized bin value for $s = 0.67$ will be $\varrho_s = 33$. Hence adding ϱ_s to the outlier of $\mathbf{A}$ i.e. $\mathcal{O}_\mathbf{A} + \varrho_s = -1 + 33 = 32$. Finally, the metadata for the scalar addition will result in $\mathcal{O}_\mathbf{A} = 32$, $\mathcal{P}_\mathbf{A} = \{0, 0, 1, 0\}$ and $\varsigma_\mathbf{A} = \{1, 1, 0, 1\}$.

3) **Scalar Subtraction:** Scalar subtraction [34] involves subtracting a scalar value (s) from the matrix. This is similar to scalar addition, but here the scalar quantized value (ϱ_s) is deducted from the outliers $\mathcal{O}$ of each block. This operation is also performed in full compressed space.
4) **Scalar Multiplication:** Scalar multiplication [34] involves multiplying an element in a matrix. Since the values in the matrix obtained are predicted values and the prediction is made based on addition operations, it is impossible to perform multiplication without exact quantized values for each block. Hence, we revert the matrix for multiplication to obtain the corresponding quantized values denoted as $\varrho_\mathbf{A}$. We then get the quantized value of the scalar s as ϱ_s and multiply it with $\varrho_\mathbf{A}$. These are then reversed into compressed form to obtain a compressed scalar multiplied matrix. As we do have to decompress the data for scalar multiplication partially, this operation is performed in partially decompressed space[1].
Suppose we want to multiply a scalar value $s = 3.14$ by A. The quantized bin value for s will be $\varrho_s = 157$. Multiplying ϱ_s to the quantized values for the block $\varrho_\mathbf{A} = \{-1, -1, -3, -3\}$ results in $\varrho_\mathbf{A} = \{-157, -157, -471, -471\}$ which is then divided by error produced by quantization of scalar value ($2 \times \epsilon$). The metadata for this scalar multiplication will be $\varrho_\mathbf{A} = \{-3, -3, -9, -9\}$. Finally, the compressed data will have $\mathcal{O}_\mathbf{A} = -3$, $\mathcal{P}_\mathbf{A} = \{0, 0, 6, 0\}$ and $\varsigma_\mathbf{A} = \{0, 0, 1, 0\}$.

B. Univariate Reductions

Univariate reductions are the operations performed on one compressed data (or matrix), which results in a single floating-point value. We describe each reduction available in SZOps here in detail.

1) **Mean:** Mean [35] is calculated as the sum of all the elements in the matrix divided by the total number of elements. The quantized values of the block ($\varrho_{\mathbf{A_1}}$) are summed together to get block-wise addition. These block-wise additions are then divided by the total number of elements ($m \times n$) to obtain the mean of the entire matrix. This process produces a final decompressed mean value instead of compressed data. Note that the same kernel can be used to calculate block-wise means by adding the quantized elements $\varrho_{\mathbf{A_i}}$ where $i = \frac{m}{m'} \times \frac{n}{n'}$ of each block and dividing each block by the number of elements in the block ($m' \times n'$).
Suppose, we want to find the mean of $\mathbf{A_1}$ where the quantized values $\varrho_{\mathbf{A_1}} = \{-1, -1, -3, -3\}$. These values are then added together, resulting in -8, which is then divided by the number of elements (here $m' \times n' = 4$) and then finally multiplied by $2 \times \epsilon$ to get the final mean value of -0.04.
2) **Variance:** Variance [36] is similar to the mean operation. Still, each quantized value is first subtracted from the mean of the matrix, and then the obtained value is squared and added to get block-wise additions. The block-wise additions are then summed together and divided by the total number of elements ($m \times n$) to obtain the variance of the entire matrix.
3) **Standard Deviation:** Standard deviation [37] operation is similar to variance operation, which is calculated by taking the square root of the variance of the compressed data.

[1]Partially decompressed space is defined as space, where the entire decompression pipeline is not performed instead some steps of decompression, are performed to obtain the desired results.

VI. Performance Evaluation and Analysis

In this section, we evaluate the performance of SZOps for different operations using 4 scientific applications across different domains (Section VI-A). We evaluate time performance breakdown (Section VI-B1), throughput (Section VI-B2), and compression ratio (Section VI-B3) based on these datasets with error bound of 1E-4. We show the significant performance improvement of SZOps operations over the traditional compression + operation workflow operated based on SZp, which is an outstanding ultra-fast error-bounded lossy compressor [32].

A. Experimental Setup

1) Platforms: All the experiments are performed on a machine with the following specifications: Each node has one AMD Ryzen 5 3600 processor with 6 cores and 12 threads, and 128 GB of DRAM. Each node on the system consists of 12 logical CPUs, and our multi-threaded code uses all 12 logical CPUs per node.

2) Datasets: The datasets used for experiments are four varied types of floating-point scientific data, as listed in Table III. These datasets are taken from Scientific Data Reduction Benchmarks [38] from various domains, i.e., weather simulation (Hurricane ISABEL [39]), climate simulation (CESM-ATM) [40], climate simulation (SCALE-LETKF) [41], and turbulence simulation Data (Miranda [16]). They are commonly used to evaluate different lossy compressors available in various works of literature [5], [22], [32].

3) Evaluation Metrics: For evaluating the scalar operations provided in SZOps, we perform time cost analysis, throughput analysis, and calculate the compression ratios. Below are the details of these evaluation metrics.

- **Time Cost (in seconds)** helps determine the runtime a compressor takes to perform a compression or decompression in a compressor. In SZOps, we measure the time

TABLE III
SCIENTIFIC SIMULATION REAL-WORLD DATA USED IN THE EVALUATION.

Datasets	# of fields	Dimension	Data size
Hurricane	7	$500 \times 500 \times 100$	1.25GB
CESM-ATM	5	3600×1800	1.47GB
SCALE-LETKF	12	$98 \times 1200 \times 1200$	4.9GB
Miranda	7	$256 \times 384 \times 384$	1.87GB

cost by one or more kernels for its execution; hence, the total time is the sum of the time cost by each kernel execution. For a traditional workflow of SZp, the time cost to operate is the sum of decompression time, the time taken to operate, and the compression time to compress the operated data.

- **Throughput (GB/s)** helps determine the gigabytes of data that a compressor can process in the entire process: total data divided by the time cost to process that data.
- **Compression Ratio** is the ratio of original data size to the compressed data size. The compression ratio indicates how powerful the compressor is, shrinking the original data without losing the relevant information. In our compressor, this pertinent information is the user's error bound (ϵ). We will show that our SZOps has even higher compression ratios than SZp. This is because there is no extra storage overhead in our operations of compression design, and our design can compress the blocks with outliers more effectively.

B. Performance Evaluation

First, we evaluate the overall performance of the traditional compression+operation+decompression workflow based on multiple state-of-the-art error-bounded lossy compressors (including SZp, SZ2, SZ3, SZx and ZFP). As shown in Table IV, SZp significantly outperforms all other compressors (about 1.5× speedups over the second-best one – SZx). The key reason is that SZp has the highest throughput in both compression and decompression from among all the compressors here, and the compression/decompression cost is the major bottleneck of the whole workflow, despite lower compression ratios compared with other compressors (as shown later VII). Since SZp is the best compressor for the traditional workflow, we mainly compare our SZOps with SZp in the following text, without loss of generality.

TABLE IV
THROUGHPUT (MB/SEC) FOR DIFFERENT OPERATIONS ON HURRICANE DATASET USING MULTIPLE COMPRESSORS WITH ϵ=1E-4. THIS EXPERIMENT IS PERFORMED BY FIRST PERFORMING COMPRESSION ON THE DATA, THEN DECOMPRESSING THE DATA, AND FINALLY APPLYING DIFFERENT OPERATIONS ON THE DECOMPRESSED DATA.

Operations	SZp	SZ2	SZ3	SZx	ZFP
Negation	384	100	81	264	108
Scalar addition	358	99	80	251	105
Scalar subtraction	369	99	81	257	106
Scalar multiplication	366	99	81	255	106
Mean	381	100	81	262	107
Variance	287	92	76	214	98
Standard Deviation	294	93	77	218	99

1) Time Cost: We evaluate the runtime for each operation in SZOps and compare the results with SZp. For SZOps, the time taken by each operation is calculated for the four datasets (Table III). For SZp, we perform the following tests:

- For **scalar operation**: decompression of compressed data + operations + compression.
- For **scalar reduction**: decompression of compressed data + operation.

The time taken by each step in SZp is added to obtain the end-to-end time each operation takes. We measure the time for each field of the four datasets using the absolute error bounds of 1E-4.

We observe in Figure 5 that the time cost by SZOps reductions (blue color) is significantly lower (except for *computation-as-output* for specific examples) than the time taken by operations performed using SZp, i.e., the total time cost on decompression, operation, and compression steps (shown with orange, green, and red colors). The time required for scalar operations *(compression-as-output)* is less as compared to the traditional SZp pipeline. This efficiency is attributed to the utilization of kernel operations from SZOps, which involve either partial decompression or no decompression for certain operations such as negation, scalar addition, and scalar subtraction (see table V). As a result, the overhead of decompression and subsequent compression time are substnatially reduced or even completely eliminated.

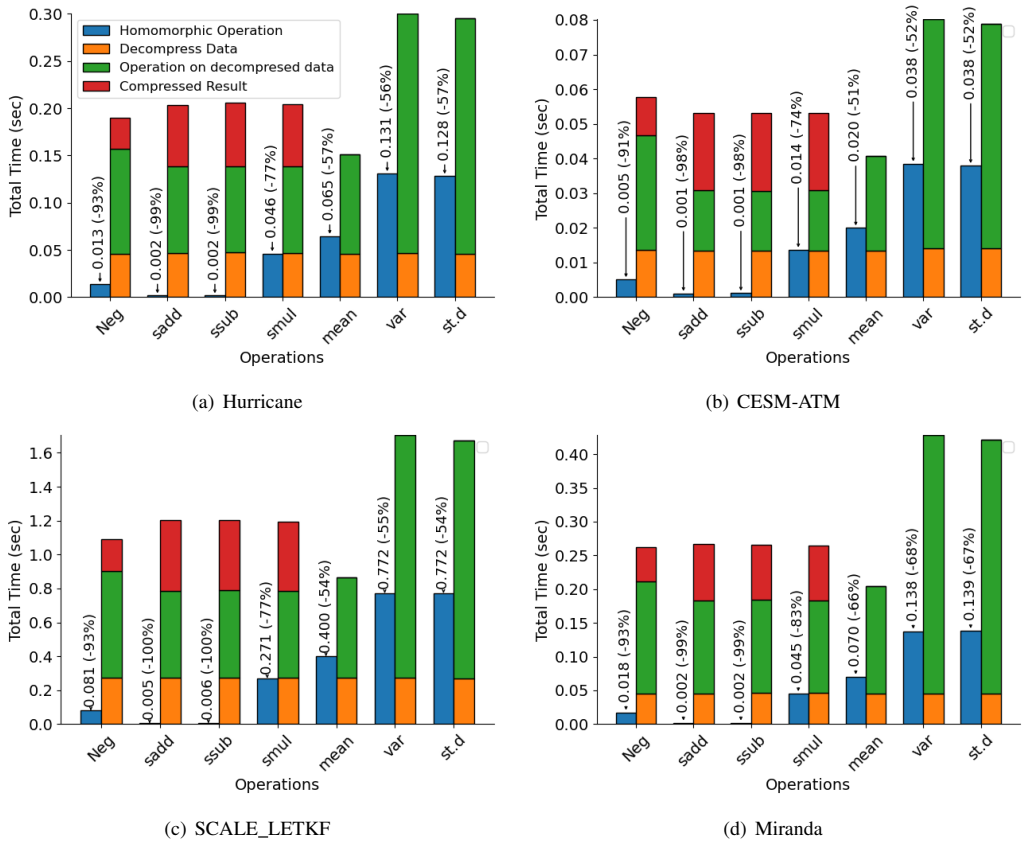
Note performing SZOps reduction operations might not always be faster than the traditional method. However, it still saves memory as we do not have to perform complete decompression and store all the data in the memory to perform reduction operations.

2) Throughput Analysis: We evaluate the throughput of SZOps and compare it with SZp for different operations. We measure the end-to-end throughput of SZp for each operation using the absolute error-bound 1E-4 for each field of the four datasets. We also evaluate the kernel throughput of SZOps using the same absolute error bound and compare it with the end-to-end throughput of SZp. We observe in Figure 6 that

TABLE V
REASONS TO PERFORMANCE IMPROVEMENT FOR DIFFERENT OPERATIONS

Operations	Reason
Scalar operations	No decompression (partial decompression + constant blocks only for scalar multiplication)
Scalar Reductions	constant blocks + integer data operations

the throughput of SZOps (shown with navy blue color) is higher than the end-to-end throughput of SZp (shown with yellow color). This is because we execute all operations within fully or partially compressed spaces while also excluding constant block computations. As a result, time is saved during the data decompression. Hence, more data can be processed per unit of time, increasing the throughput of SZOps. The throughput of reduction operations (*computation-as-output*) is lowest amongst other scalar operations. This difference arises from the prominent dependence of reduction operations on



(a) Hurricane

(b) CESM-ATM

(c) SCALE_LETKF

(d) Miranda

Fig. 5. The time cost of various operations, including Decompression (orange), Operation (green), and Compression (red) times for SZp, as well as the total time (blue) for SZOps, is compared using absolute error bounds (ϵ) of 1E-4. The total time of SZOps encompasses the kernel time taken by different operations, including partial decompression and partial compression time taken by certain operations, as detailed in Section IV-B. Each bar is color-coded to represent the time taken for a specific operation, as demonstrated in (a). <**time taken**> **(- value %)** on each blue bar represents the time taken by SZOps operation and the percentage decrease in SZOps's operation time in comparison to the corresponding SZp's operation time, respectively, for different datasets.

the number of constant blocks (see Table VI). These constant blocks contain zero values, enabling their exclusion during computation.

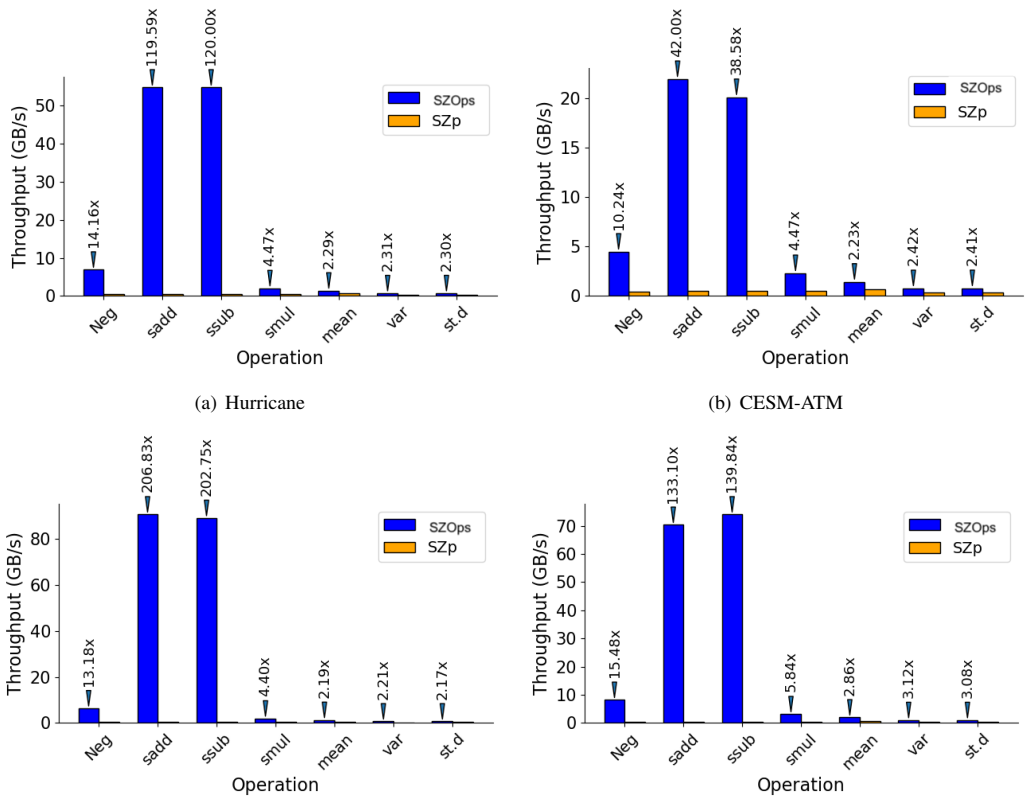
TABLE VI
TOTAL BLOCKS AND CONSTANT BLOCKS IN EACH DATASET OVER ALL THE FIELDS FOR ERROR BOUND (ϵ) 1E-2.

Datasets	Const. blocks	Total blocks	% (Const./Total)
Hurricane	360827	2734375	13%
CESM-ATM	7817	506250	1.5%
SCALE-LETKF	1071863	26460000	4%
Miranda	593722	4128768	14%

3) Compression Ratio: In Table VII, we evaluate the compression ratios for different compressors using an absolute error bound of 1E-4. SZOps outperforms SZp in terms of compression ratio but falls behind SZ, SZ3, and ZFP. The higher compression ratios of SZ, SZ3, and ZFP can be attributed to their advanced data decorrelation techniques, such as dynamic interpolation and orthogonal transform, and their effective lossless-encoding methods, such as Huffman/Zstd [19] and embedded coding [6]. The SZOps may have a higher compression ratio than SZp does, mainly because SZOps reorganizes the outliers in the pipeline (see Figure 2), making the linear recurrence decoding steps for combining the compressed data for each block easier. This improvement eliminates the need to store compressed byte length limits per block, a significant limitation in SZp's compression efficiency [42]. It is worth noting that although SZOps has lower compression ratios than other modern compressors such as SZ, SZ3 and ZFP, it exhibits substantially higher throughputs on various operations (see Figure 6 and Table IV): 2×-200×, which is critical to the online execution performance of large-scale scientific applications.

TABLE VII
AVERAGE COMPRESSION RATIOS FOR DIFFERENT SCIENTIFIC SIMULATION DATA USING DIFFERENT COMPRESSORS.

Datasets	SZOps	SZp	SZ	SZ3	SZx	ZFP
Hurricane	2.78	1.59	8.83	10	3.6	4.4
CESM-ATM	2.68	2.33	6.48	5.0	2.17	3.01
SCALE-LETKF	17.02	15.21	360.65	205.74	37.13	69.48
Miranda	6.19	4.97	24.64	27.70	5.11	8.78



(a) Hurricane (b) CESM-ATM

(c) SCALE_LETKF (d) Miranda

Fig. 6. Kernel throughput for SZOps and end-to-end throughput for SZp using absolute error bound (ϵ) 1E-4. The performance throughput ratio of each SZOps operation with respect to SZp is shown above each blue bar.

VII. Conclusion and Future Work

We propose SZOps, an error-bounded lossy compressor that can perform scalar operations in compressed space. SZOps consists of a seven lightweight scalar operation and reductions. To the best of our knowledge, this is the first attempt to develop an efficient error-bounded lossy compressor that support compression in terms of diverse types of scalar operations. We perform experiments with real-world datasets, and show that SZOps can achieve higher throughput and better performance than a SZp while providing reasonable compression ratios.

In the future, we will add more operations, like multivariate operations, distance measures, similarity measures, and compositions to make the tool more powerful.

Acknowledgments

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 OAC Grants 2003709, 2104023, 2311875, 2330367, 2311756, and 2313122, and CCF under Grants 2403379, 2346394, 2426055, 2217154, 2124100, and 1956106. The experimental resource for this paper was provided by the Laboratory Computing Resource Center on the Bebop cluster at Argonne National Laboratory.

References

[1] J. E. Kay and et al., "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*, vol. 96, no. 8, pp. 1333 – 1349, 2015.

[2] "Team at princeton plasma physics laboratory employs doe supercomputers to understand heat-load width requirements of future-iter device," 2021.

[3] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *IEEE International Parallel and Distributed Processing Symposium*, 2016, pp. 730–739.

[4] D. Tao, S. Di, Z. Chen, and F. Cappello, "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, 2017, pp. 1129–1139.

[5] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, Z. Chen, and F. Cappello, "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*, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 89–100.

[6] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[7] D. Tao, S. Di, Z. Chen, and F. Cappello, "In-depth exploration of single-snapshot lossy compression techniques for n-body simulations," in *2017*

IEEE International Conference on Big Data (Big Data), 2017, pp. 486–493.

[8] J. Liu, S. Di, K. Zhao, X. Liang, Z. Chen, and F. Cappello, "Faz: A flexible auto-tuned modular error-bounded compression framework for scientific data," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–13.

[9] X. Yu, S. Di, K. Zhao, J. Tian, D. Tao, X. Liang, and F. Cappello, "Ultrafast error-bounded lossy compression for scientific datasets," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 159–171. [Online]. Available: https://doi.org/10.1145/3502181.3531473

[10] J. Huang, J. Liu, S. Di, Y. Zhai, Z. Jian, S. Wu, K. Zhao, Z. Chen, Y. Guo, and F. Cappello, "Exploring wavelet transform usages for error-bounded scientific data compression," in *2023 IEEE International Conference on Big Data (BigData)*, 2023, pp. 4233–4239.

[11] X. Liang, S. Di, D. Tao, S. Li, B. Nicolae, Z. Chen, and F. Cappello, "Improving performance of data dumping with lossy compression for scientific simulation," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–11.

[12] K. Zhao, S. Di, D. Perez, X. Liang, Z. Chen, and F. Cappello, "Mdz: An efficient error-bounded lossy compressor for molecular dynamics," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 27–40.

[13] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356155

[14] Y. Liu, S. Di, K. Chard, I. Foster, and F. Cappello, "Optimizing scientific data transfer on globus with error-bounded lossy compression," in *IEEE International Conference on Distributed Computing Systems (IEEE ICDCS2023)*, 2023.

[15] S. Li, P. Lindstrom, and J. Clyne, "Lossy scientific data compression with sperr," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 1007–1017.

[16] B. J. Olson and J. Greenough, "Large eddy simulation requirements for the Richtmyer-Meshkov instability," *Physics of Fluids*, vol. 26, no. 4, p. 044103, 04 2014. [Online]. Available: https://doi.org/10.1063/1.4871396

[17] E. Robein, "Eage e-lecture: Reverse time migration: How does it work, when to use it," https://youtu.be/ywdML8ndYeQ, November 15, 2016.

[18] J. Huang, S. Di, X. Yu, Y. Zhai, Z. Zhang, J. Liu, X. Lu, K. Raffenetti, H. Zhou, K. Zhao, Z. Chen, F. Cappello, Y. Guo, and R. Thakur, "An optimized error-controlled mpi collective framework integrated with lossy compression," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 752–764.

[19] Y. Collet, "Zstandard – real-time data compression algorithm," *http://facebook.github.io/zstd/*, 2015.

[20] M. Martel, "Compressed matrix computations," 2022.

[21] T. Agarwal, H. Dam, P. Sadayappan, G. Gopalakrishnan, D. B. Khalifa, and M. Martel, "What operations can be performed directly on compressed arrays, and with what error?" in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 254–262.

[22] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1643–1654.

[23] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao *et al.*, "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors," *IEEE Transactions on Big Data*, vol. 9, no. 2, pp. 485–498, 2022.

[24] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[25] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-core compression and decompression of large n-dimensional scalar fields," in *Computer Graphics Forum*, vol. 22, no. 3. Wiley Online Library, 2003, pp. 343–348.

[26] J. Liu, S. Di, K. Zhao, X. Liang, S. Jin, Z. Jian, J. Huang, S. Wu, Z. Chen, and F. Cappello, "High-performance effective scientific error-bounded lossy compression with auto-tuned multi-component interpolation," *Proc. ACM Manag. Data*, vol. 2, no. 1, mar 2024. [Online]. Available: https://doi.org/10.1145/3639259

[27] Z. Jian, S. Di, J. Liu, K. Zhao, X. Liang, H. Xu, R. Underwood, S. Wu, J. Huang, Z. Chen, and F. Cappello, "Cliz: Optimizing lossy compression for climate datasets with adaptive fine-tuned data prediction," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 417–429. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS57955.2024.00044

[28] D. Wang, J. Pulido, P. Grosset, J. Tian, S. Jin, H. Tang, J. Sexton, S. Di, K. Zhao, B. Fang, Z. Lukić, F. Cappello, J. Ahrens, and D. Tao, "Amric: A novel in situ lossy compression framework for efficient i/o in adaptive mesh refinement applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3613212

[29] W. A. Pearlman, A. Islam, N. Nagaraj, and A. Said, "Efficient, low-complexity image coding with a set-partitioning embedded block coder," *IEEE transactions on circuits and systems for video technology*, vol. 14, no. 11, pp. 1219–1235, 2004.

[30] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, "TTHRESH: Tensor compression for multidimensional visual data," *IEEE transactions on visualization and computer graphics*, vol. 26, no. 9, pp. 2891–2903, 2019.

[31] G. Ballard, A. Klinvex, and T. G. Kolda, "Tuckermpi: A parallel c++/mpi software package for large-scale data compression via the tucker tensor decomposition," *ACM Trans. Math. Softw.*, vol. 46, no. 2, jun 2020. [Online]. Available: https://doi.org/10.1145/3378445

[32] Y. Huang, S. Di, X. Yu, G. Li, and F. Cappello, "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*, ser. SC '23. NY, USA: Association for Computing Machinery, 2023.

[33] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-core compression and decompression of large n-dimensional scalar fields," *Computer Graphics Forum*, vol. 22, no. 3, pp. 343–348, 2003.

[34] R. Bronson, *Theory and problems of matrix operations*. The McGraw Hill Companies,, 1989.

[35] L. Underhill and D. Bradfield, *Introstat*. Juta and Company Ltd, 1996.

[36] L. Wasserman, *All of statistics: a concise course in statistical inference*. Springer, 2004, vol. 26.

[37] J. M. Bland and D. G. Altman, "Measurement error." *BMJ: British medical journal*, vol. 312, no. 7047, p. 1654, 1996.

[38] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "SDRBench: Scientific data reduction benchmark for lossy compressors," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2716–2724.

[39] "Ieee visualization 2004 contest data set – hurricane isabel," http://vis.computer.org/vis2004contest/data.html, 2004.

[40] J. W. Hurrell, M. M. Holland, P. R. Gent, S. Ghan, J. E. Kay, P. J. Kushner, J.-F. Lamarque, W. G. Large, D. Lawrence, K. Lindsay *et al.*, "The community earth system model: a framework for collaborative research," *Bulletin of the American Meteorological Society*, vol. 94, no. 9, pp. 1339–1360, 2013.

[41] SCALE-LETKF simulation, https://github.com/SCALE-LETKF-RIKEN/scale-letkf/tree/5.4.5-v1, 2023, online.

[42] A. N. Laboratory, "Szp-a lossy error-bounded compression library for compression of floating-point data using openmp acceleration." 2023.