# cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance

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

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

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

Guanpeng Li
University of Iowa
Iowa City, IA, USA
guanpeng-li@uiowa.edu

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

## ABSTRACT

Modern scientific applications and supercomputing systems are generating large amounts of data in various fields, leading to critical challenges in data storage footprints and communication times. To address this issue, error-bounded GPU lossy compression has been widely adopted, since it can reduce the volume of data within a customized threshold on data distortion. In this work, we propose an ultra-fast error-bounded GPU lossy compressor cuSZp. Specifically, cuSZp computes the linear recurrences with hierarchical parallelism to fuse the massive computation into one kernel, drastically improving the end-to-end throughput. In addition, cuSZp adopts a block-wise design along with a lightweight fixed-length encoding and bit-shuffle inside each block such that it achieves high compression ratios and data quality. Our experiments on NVIDIA A100 GPU with 6 representative scientific datasets demonstrate that cuSZp can achieve an ultra-fast end-to-end throughput (95.53x compared with cuSZ) along with a high compression ratio and high reconstructed data quality.

## CCS CONCEPTS

• **Theory of computation** → **Data compression**; • **Computing methodologies** → **Parallel algorithms**; • **Computer systems** → **High-performance computing**.

## KEYWORDS

Error-bounded Lossy Compression, GPU, Parallel Computing, Scientific Simulation, High-speed Compressor, CUDA

**ACM Reference Format:**
Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. 2023. cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607048

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


*SC '23, November 12–17, 2023, Denver, CO, USA*

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607048

## 1 INTRODUCTION

Today's scientific high-performance computing (HPC) applications produce a sheer amount of data during execution for post hoc analysis [6, 12, 18, 23, 26, 27, 36]. For example, reverse time migration (RTM), which is a representative simulation for seismic imaging containing thousands of timesteps, may produce up to 2,800 TB data for a 10x10x8 square kilometers geological structure within only one timestep [13, 25]. Hence, it has become extremely challenging to store and manage such a great amount of data in an efficient manner in the existing storage infrastructure.

Data compression is a widely used approach to address the challenges of managing vast amounts of data in HPC applications [6, 26, 29, 40]. While lossless compression techniques suffer from low compression ratios (up to 2:1) [24, 26], error-bounded lossy compression turns out to be a promising solution [21, 33, 41]. By allowing certain errors introduced by lossy compression, the technique can significantly improve the compression ratios (up to 100+) while also preserving high reconstructed data quality [8, 30, 41].

In general, three goals are often pursued in the design of a lossy compressor: (1) high compression/decompression throughput, (2) high compression ratios, and (3) high reconstructed data quality, which can be analyzed by visualization or statistical metrics. Although existing GPU lossy compression techniques, such as cuSZ [21], cuZFP [33], cuSZx [39], can achieve high kernel throughput and compression ratios, their designs have several key defects. For example, cuSZ relies on CPU computation to build its Huffman tree for the variable-length encoding step, which may significantly affect the overall end-to-end performance. Moreover, cuSZ performs the compression (and decompression) in multiple kernels, which may cause inevitable extra data movement overheads. Another state-of-the-art lossy compressor, cuSZx, improves the throughput by using a constant block design, however, it still requires extra CPU computations to accomplish preprocessing and global synchronization, thereby end-to-end throughput still suffers. Moreover, the constant block design in cuSZx flushes the data points in the relatively smooth regions to a constant value, which may significantly lower the reconstructed data quality unexpectedly. cuZFP has no drawbacks mentioned above because it integrates all steps in one

GPU kernel, but it experiences low quality in reconstructed data because it supports only fixed-rate mode (hence not error-bounded).

In this work, we propose a novel error-bounded GPU lossy compressor that exhibits the best level for all the three design goals mentioned above. There are two main research challenges that we have to address: **(C1) Putting the entire parallel computation in one kernel.** The ideal implementation for an ultra-fast GPU compressor is to put all components into one single kernel, inhibiting unnecessary overheads including kernel launching and data movements. Unlike fixed-rate compressors [21], error-bounded lossy compressors are often designed in a block-wise processing manner [19–21, 28, 39], and each block incurs uncertain length of compressed data, leading to imbalanced workload. This is a serious issue for GPU implementation because there exists a linear recurrence to aggregate all the variable-length data chunks together at the end of compression across different thread blocks. **(C2) Balancing between high throughput and high compression ratio.** To achieve high compression ratios, existing error-bounded compressors often require fine-grained computations such as Huffman encoding [33] and entropy/dictionary encoding [7] to operate for repeated patterns generated from previous components. Such expensive designs unavoidably reduce the throughput, especially in GPU platform that prefers massive parallelism.

Toward this end, we propose cuSZp[1], an ultra-fast error-bounded lossy compression framework for GPU with optimized end-to-end performance. Specifically, cuSZp is a block-wise design that can be roughly categorized into four major steps. **(S1) Quantization and Prediction**: After dividing the whole dataset into blocks each with the same length, cuSZp first performs a pre-quantization and a lightweight Lorenzo prediction inside each block, which is the only "lossy" step in cuSZp. **(S2) Fixed-length Encoding**: Then, still inside each block, cuSZp computes the maximum absolute value of the quantization integer and adopts a fixed-length encoding to compress the data losslessly. If all quantization integers of a block are zero, cuSZp records this block as zero-block and bypasses this step. By doing so, cuSZp can achieve a fairly high compression ratio on smooth and sparse datasets. **(S3) Global Synchronization**: Since different blocks may have various maximum required numbers of bits in the fixed-length encoding, cuSZp performs a global prefix-sum synchronization to compute the compressed data index along with the compressed size of the whole dataset. Our designed global synchronization fully utilizes the hierarchy parallelism and thus can achieve high throughput. **(S4) Block Bit-shuffle**. Finally, cuSZp optimizes the bit-level shifting operation with a bit-shuffle algorithm to store the compressed data back to GPU global memory. *To the best of our knowledge, we are the first work that proposes an error-bounded lossy compressor in a single GPU kernel and achieves an ultra-fast end-to-end performance also with a high compression ratio and high data quality.* We perform a comprehensive evaluation using NVIDIA Ampere A100 GPU provided by Argonne Swing cluster, based on 6 real-world application data across from different domains (including weather simulation, cosmology simulation, quantum Monte Carlo simulation, seismic imagining, and so on), which involves 100+ datasets fields in total. The main evaluation results of cuSZp are summarized as follows:

- cuSZp can achieve 93.63 GB/s and 120.04 GB/s **end-to-end throughput** on average for compression and decompression, respectively, which are 95.53× and 55.18× faster than cuSZ and cuSZx. By end-to-end throughput, we mean the period from the moment when the raw data are populated in the GPU global memory to the moment when the compressed data are put in the GPU global memory by the compressor.
- The **kernel throughput** of cuSZp is also inline/comparable (around 100GB/s or higher in most cases) with state-of-the-art GPU lossy compressors including cuZFP, cuSZx, and cuSZ, where the kernel throughput means the performance of kernel execution excluding kernel launch and data movement.
- Compared with two cutting-edge error-bounded lossy compressors (cuSZ and cuSZx), cuSZp can obtain the highest compression ratios in most cases at the same error bound with far higher end-to-end throughput.
- cuSZp can preserve a good quality for reconstructed data in both statistical and visualization metrics for different scientific datasets. Specifically, compared with cuZFP, not only can cuSZp significantly improve the rate distortion for both PSNR and SSIM, but it can also considerably remove the undesired artifacts in the reconstructed datasets.

The rest of this paper is organized as follows. Section 2 defines the terminologies and states the problem. Section 3 and 4 describes the design of cuSZp. Section 5 evaluates cuSZp with several representative scientific datasets. Section 6 and 7 present several discussions and the related works, respectively. In Section 8, we conclude this work with a vision of future work.

## 2 TERMINOLOGIES AND PROBLEM STATEMENT

In this Section, we first define the terminologies that are used in this paper, then we state the problem based on these terminologies.

### 2.1 Terminologies

- **Error-bounded lossy compression**: In error-bounded lossy compression, the introduced error should be strictly bounded by the user-defined tolerance (i.e. error bound). Given a scientific dataset $D = \{d_1, d_2, ..., d_N\}$, where $d_i$ and $N$ denote the $i$-th data point and the length of $D$, respectively. After the compression and decompression computations, the reconstructed data can be denoted by $D' = \{d'_1, d'_2, ..., d'_N\}$, which should satisfy $\max_{i=1,2,...,n} |d_i - d'_i| \leq eb$, where $eb$ is the user-defined error bound.
- **Error bound**: There are two types of error modes, absolute error bound (ABS) and value-range-based relative error bound (REL), that are commonly used in scientific applications [31]. ABS error bound $\delta$ is set as a constant value, while REL error bound can be denoted as $\lambda r$, where $\lambda \in (0, 1)$ denotes relative ratio and $r$ represents the value range (i.e. max value - min value) in scientific dataset $D$. Thus, these two error bounds can be formulated as

$$eb = \begin{cases} \delta, & \text{ABS error bound } \delta \text{ is used.} \\ \lambda r, & \text{REL error bound } \lambda \text{ is used.} \end{cases} \quad (1)$$

[1] cuSZp code: https://github.com/szcompressor/cuSZp.

For simplicity, we use ABS $\delta$ and REL $\lambda$ (e.g. ABS=1E-4, REL=1E-3) to denote these two error bounds in later text.

- **Throughput**: Compression/decompression throughput represents how much data can be processed in a segment of time (e.g. GB/s). This is also the main benefit to utilize a GPU lossy compressor rather than a CPU one.
- **Compression ratio**: Compression ratio is defined as the ratio of the original data size to the compressed data size.
- **Data quality**: Data quality evaluates the distortion (i.e. errors) in a dataset that is reconstructed by a lossy compressor.
- **PSNR**: Peak signal-to-noise ratio (PSNR) [11], measured by decibels (dB), is a commonly used statistical metric to measure the quality of reconstructed data compared to its original version. PSNR measures the quality from the perspective of pixel-wise differences, and the higher the better.
- **SSIM**: Structural similarity index measure (SSIM) [35] is another important metric to evaluate the reconstructed data quality. SSIM is a perceptual metric that is designed to reflect the perceived quality by calculating luminance, contrast, and structure information. The value range of SSIM is from -1 to 1, and the higher the better (1 indicates perfect similarity).

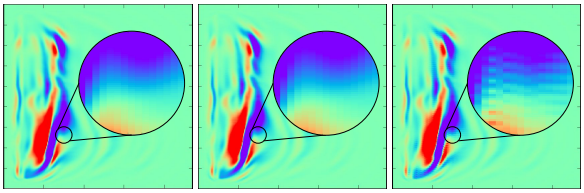
## 2.2 Problem Statement

In this work, our objective is to propose a GPU error-bounded lossy compressor that can achieve (i) end-to-end high throughput, (ii) high compression ratio, and (iii) high data quality at the same time.

**(i) As for the end-to-end throughput**, we notice that the scientific applications running on GPU often load the data stored in GPU global memory and process them at runtime. Regarding the compression operation, they expect to deal with the data compression and decompression fully in GPU kernels with all related data pointed by GPU pointers. Note that a kernel here denotes a function that is fully executed on GPU. In our work, we target the *end-to-end performance*, which refers to the duration from the moment when the original data is generated in GPU memory to the time stamp that the compressed data is stored in GPU memory. In contrast, many of existing GPU compression methods [33, 39] focus only on the kernel throughput, which still suffers from very low end-to-end performance because of the inevitable CPU operations in their methods or expensive data movements among CPU, GPU global memory and shared memory. In our solution, we manage to develop an efficient compression method that includes every step in one GPU kernel, which effectively eliminates the costs mentioned above. Consequently, in single-kernel GPU compressor design, end-to-end throughput is the same as kernel throughput.

**(ii) For the compression ratio**, due to the need for high throughput, it is challenging to develop a lossy compressor that can have a higher compression ratio than cuSZ, which is a cutting-edge GPU error-bounded lossy compressor optimized for high compression ratio in particular [33]. Whereas, we still manage to devise an effective compression method that also features comparable or even higher compression ratios in comparison.

**(iii) For data quality**, we use not only statistical metrics PSNR and SSIM, but also perform visualization assessment, which is also conducted commonly in many applications such as climate simulation [15], reverse time migration (RTM) [43] and cosmology simulations [9]. The importance of visualization assessment is also verified in Figure 1. We use a real-world scientific dataset – RTM simulation [4, 13] as an example, which was reconstructed by two different lossy compressors. Figure 1(a) demonstrates the visualization of the original dataset, and Figure 1(b) and 1(c) correspond to the decompressed datasets by different settings. Although the reconstructed data2 has a higher SSIM (0.9948), it has more obvious distorted patterns compared with reconstructed data1, which is almost identical to the original data. This example reveals the fact that only using PSNR and SSIM may not be adequate to assess the reconstructed data quality: i.e., a high PSNR or SSIM may still have notably distorted visualization, which is not satisfied from the perspective of domain experts. Therefore, we use all of them to assess the data quality.



(a) Original data (b) Reconstructed data1 (SSIM: 0.9871) (c) Reconstructed data2 (SSIM: 0.9948)

**Figure 1: Demonstrating the importance of visualization in the compressed data quality assessment. Scientific Dataset: RTM (dim2, slice100).**

# 3 HIGH-LEVEL DESIGN OF CUSZP

In this section, we provide a high-level overview of the compression and decompression algorithms in cuSZp.

## 3.1 cuSZp Compression

Figure 2 shows a high-level overview of cuSZp compression framework. Given a scientific dataset as input, cuSZp always treats it as a 1D array and divides it into a set of blocks each with the same length. If one block is a "Non-Zero" block (i.e., the block has at least one data point which is not 0), cuSZp will conduct Quantization and Prediction (❶) and Fixed-length Encoding (❷), also generating the block offset (i.e. compressed data size of this block) for this block. Otherwise, the block offsets are recorded as 0, representing that all data points in this block are zero. Then, all computed block offsets are written into a *block offset array*, which is stored as a global variable with a length that equals to the total number of blocks in the input dataset. To generate the locations of the encoded/compressed block data, cuSZp performs Global Synchronization (❸) to parallelize this linear recurrence and generate a synchronized offset array. The synchronized offset array has the same length as the block offset array, recording the encoded data location in the final compressed data. Finally, cuSZp performs a Block Bit-shuffle (❹) and writes the encoded data into final compressed data based on the computed location.

## 3.2 cuSZp Decompression

Figure 3 presents a high-level overview of cuSZp decompression framework. Given the compressed data in bytes, cuSZp first reads the fixed-length for each block and generates *block offset array*

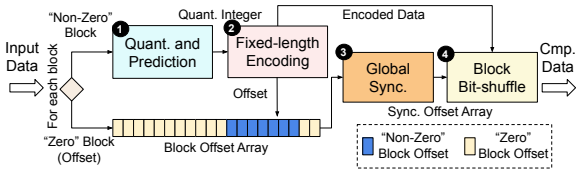



**Figure 2: High-level overview of cuSZp compression.**

(details will be presented in Equation 2). Similar to the compression phase, cuSZp performs Global Synchronization (❶) to generate the locations that store the compressed data for each block. Given these locations, cuSZp performs the reverse operations of Block Bit-shuffle (❷), Fixed-length Encoding (❸), and Quantization and Prediction (❹), in order to obtain the reconstructed data.

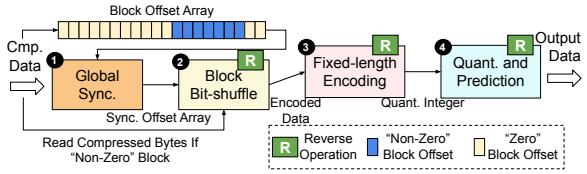



**Figure 3: High-level overview of cuSZp decompression.**

For simplicity, in Section 4, we only introduce the compression phase in detail, whereas some reverse operations, such as how to perform reverse Quantization and Lorenzo based on error bounds $eb$, will only be discussed in context if needed.

## 4 DETAILED DESIGN OF CUSZP

In this Section, we describe each component in cuSZp compression and explain the design motivation. Key notations that are used in this section are summarized in Table 1.

| Notation | Description |
|---|---|
| $D$ | Original scientific dataset. |
| $D'$ | Reconstructed scientific dataset. |
| $eb$ | User-defined error bound. |
| $B_k$ | The $k$-th block in scientific dataset $D$ or $D'$. |
| $L$ | The length of one block. |
| $N$ | The number of blocks in scientific dataset $D$ or $D'$. |
| $d_i$ | The $i$-th original data in one block. |
| $d'_i$ | The $i$-th reconstructed data in one block. |

**Table 1: Notations that are frequently used in Section 4.**

### 4.1 Quantization and Prediction (❶)

The goal of Quantization and Prediction step is to convert floating point data into a set of integers that contain less randomness in their bits, making them easier to be processed in the succeeding encoding steps. Similar to previous works [33], cuSZp first adopts a pre-quantization to convert data types from floating points to integer based on the user-defined error bound. This is the only "lossy" step in the entire compression pipeline. Then, inside each block, cuSZp performs a lightweight Lorenzo prediction to reduce the integer value to a smaller reversible one, removing repeated bit-level patterns and hence increasing encoding efficiency.

Pre-quantization in cuSZp is used to convert the continuous floating point values to a discrete set of integers, where the maximum error in integers is guaranteed to be within the user-defined error bound. Given one block $B = \{d_1, d_2, ..., d_L\}$, $d_i$ denotes $i$-th floating point element in block $B$ and $L$ represents the block length. The pre-quantization step can generate a set of integers $\{r_1, r_2, ..., r_L\}$, where each integer $r_i$ can be computed as a rounding operation $\text{round}(d_i/2eb)$, guaranteeing $|r_i \times 2eb - d_i| \leq eb$. In the decompression, given a quantization integer $r_i$, the output data can be constructed as $d'_i = r_i \times 2eb$. As we can see, the pre-quantization in cuSZp is the only "lossy" step and can be computed independently, indicating it is naturally suitable for parallel implementation on GPU. The quantization step in cuSZp is similar to other state-of-the-art lossy compression works [10, 30, 33].

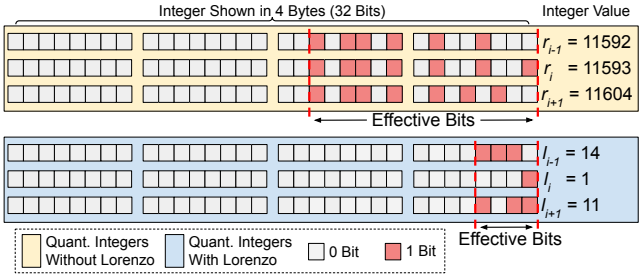



**Figure 4: Processing quantization integer with lightweight Lorenzo prediction.**

cuSZp then performs a lightweight Lorenzo prediction [14], a 1D 1-Layer Lorenzo prediction, to process the quantization integers, making them easier to be encoded in the later steps. In the pre-quantization, a smaller user-defined error bound can lead to a larger quantization integer, causing repeated bit patterns in the stream of data, as shown in Figure 4. To deal with this, cuSZp uses a Lorenzo prediction to eliminate these redundant repeated bits by recording only the differences between adjacent quantization integers inside the block. For each quantization integer $r_i$ in one block, this process can be formulated as $l_i = r_i - r_{i-1}$, where $r_{i-1}$ is set as 0 for the first quantization integer. Our Lorenzo prediction is conducted separately among different blocks, so that the linear recurrence only occurs within each block, making this possible for parallelism. Figure 4 illustrates why Lorenzo prediction can reduce the repeated bit patterns significantly. Here we use only 3 data points from a data block. We can see that the number of effective bits is reduced from 14 to 4, aggregating the data information into a denser format.

Beyond 1D 1-Layer Lorenzo prediction, there are also other Lorenzo predictions with higher dimensions and layers [14, 30]. The reason we choose this lightweight one is discussed as follows. First, for the parallel implementation in GPU, we conduct Lorenzo prediction only inside each block in the scientific dataset, where the values are relatively smooth among adjacent data points (we will demonstrate this in Section 4.2). Such operation reduces the data complexity, generating similar performance between lightweight and more complex Lorenzo predictions (also validated in our experiments). Since the first and foremost target of cuSZp is high throughput, we use a lightweight one with fewer computations.

### 4.2 Fixed-length Encoding (❷)

cuSZp uses a Fixed-length Encoding to reduce the data size by only preserving a fixed number of bits for each data point in one "Non-Zero" block. Given a block that consists of quantization integers (processed after Lorenzo prediction), cuSZp first records their sign

information with bits and combines those bits into a combined sign map, since the C language stores negative integers with an extra complement notation. If this integer is positive, cuSZp will mark it using the bit 0, otherwise bit 1. Therefore, for a block with length $L$, cuSZp requires extra $L/8$ unsigned chars, of which length is 1 byte for each unsigned char, to store the sign map. Then, cuSZp iterates the data points in the block and computes the maximum value. As all the quantization integers are already converted to their corresponding absolute values, cuSZp just needs to record the position of the last non-zero bit of this maximum integer. This position is then regarded as the fixed-length for this block. We use an example to further illustrate this idea. Suppose a non-zero block (after pre-quantization+Lorenzo+absolute) contains 8 data points {1,2,5,11,2,0,0,1,0}, then the maximum value is 11, which indicates we just need to keep 4 bits (the position mentioned above) for each of the data points. cuSZp preserves the same number of bits (i.e. fixed-length) for all absolute integers in each "non-zero" block. If the fixed-length of block $B_k$ is computed as $F_k$, the compressed data length (in bytes) of $B_k$ can be computed as the following equation:

$$CmpL_k = \frac{(F_k + 1) \times L}{8} \tag{2}$$

where $CmpL_k$ denotes the compression size of block $B_k$. $CmpL_k$ is also the block offset to be stored in the global variable for later synchronization.

Figure 5 illustrates the fixed-length encoding step in cuSZp by a block consisting of 8 quantization integers (i.e. $L = 8$). After generating the sign map, cuSZp iterates all absolute integers and localizes the maximum value $l_2 = 134$. We can see that $l_2$ has the leftmost leading non-zero bit (8th bit), which also decides the fixed-length as 8. The decided fixed-length 8 can cover all effective bits (from the first bit to the last non-zero bit) for the rest absolute integers within the same block. The compression size can be computed as $(8 + 1) \times 8/8 = 9$ bytes, based on Equation (2).

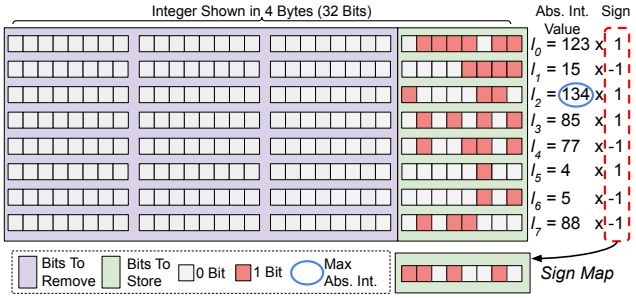
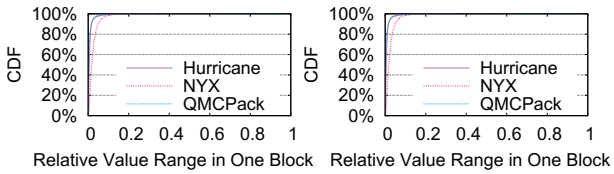


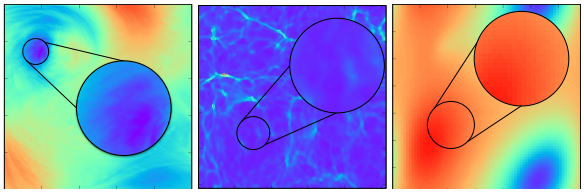

**Figure 5: Compressing a "None-Zero" block ($L = 8$) with Fixed-length Encoding.**

Compared with Fixed-length Encoding in cuSZp, there are some other more fine-grained variable-length encoding strategies, such as Huffman encoding, which can achieve a high compression ratio by storing high-frequency data with fewer bits. However, we still choose Fixed-length Encoding in cuSZp, because of the following reasons. First, cuSZp is a parallel design and splits data into multiple blocks, where data frequency inside one block is not high, limiting the compression ratio for Huffman encoding. Second, building and storing the Huffman tree acquires extra computation and space overhead, especially for a parallel block-wise design, inhibiting a high throughput design for cuSZp. At last, we observe that scientific datasets are routinely very smooth in space, and the value range of one cuSZp block ($L$ consecutive data points) is relatively small. Figure 6 shows a quantitative analysis for 3 widely used scientific datasets, which are Hurricane (Field: U) from climate simulation [15], NYX (Field: temperature) from cosmology simulation [3], and QMCPack from quantum simulation [17]. We calculate the cumulative distribution function (CDF) of the block's value range ($L$=8 and 32) and normalize this range into $[0, 1]$ based on the value range of the whole dataset for the sake of clear observation. As we can see, these datasets exhibit high smoothness within blocks. In absolute terms, in Hurricane, the relative value ranges of more than 80% blocks are smaller than 0.02 when block length $L$ is set as 8. Figure 7 demonstrates such smoothness from the perspective of visualization as well. The same conclusions can be drawn when we set $L$ as 64 and 128. As a result, we reckon that Fixed-length Encoding is an appropriate choice in cuSZp design.




(a) Block length $L = 8$ (b) Block length $L = 32$

**Figure 6: Cumulative Distribution Function (CDF) of block's relative value range in 3 scientific datasets.**



(a) Hurricane (dim3, slice50) (b) NYX (dim2, slice200) (c) QMCPack (dim3, slice1000)

**Figure 7: Demonstrating high smoothness in scientific datasets by visualization.**

## 4.3 Global Synchronization (❸)

Global Synchronization in cuSZp can generate the index of each block in the final compressed data (for the whole scientific dataset). Recall that the compressed length for each block, decided by block fixed-length (see Equation (2)), varies a lot across different blocks. One cannot directly concatenate the compressed blocks with a uniform length per block, otherwise, there would be data loss for some blocks or extra holes in the final compressed data, drastically decreasing the compression ratio. Existing GPU lossy compressors, such as cuSZx [39], generally perform this step in the CPU for simplicity. Such a design is suitable for the CPU-GPU hybrid simulations but cannot achieve high end-to-end throughput in the pure GPU simulations (as stated in Section 2.2), due to the expensive CPU-GPU data movement overheads.

Global Synchronization in cuSZp is formulated as a classic *prefix-sum* (i.e.*scan*) problem [22]. Figure 8 explains this with a concatenation of 4 blocks in cuSZp. After the Fixed-length Encoding step,

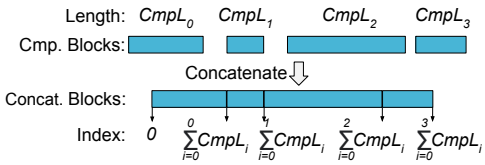



**Figure 8: Illustrating Global Synchronization is a prefix-sum problem with a toy example.**

each block $B_k$ is compressed from $L$ integers into $CmpL_k$ bytes. To concatenate those compressed blocks into the final consecutive memory, each compressed block should know its start index and end index. For $B_0$, its compressed length is $CmpL_0$, indicating it should be stored starting from index 0 to index $CmpL_0$. $B_1$ stores in the same way, starting from index $CmpL_0$ to index $CmpL_0 + CmpL_1$. Thus, for each block $B_k$, the compressed bytes should be stored from index $\sum_{i=0}^{k-1} CmpL_i$ to index $\sum_{i=0}^{k} CmpL_i$. Calculating those indexes can thus be formulated as an exclusive prefix-sum problem:

$$\begin{aligned} &\text{Input}: BlockOffsetArray = \{CmpL_0, ..., CmpL_N\} \\ &\text{Output}: SyncOffsetArray = \{0, \sum_{i=0}^{0} CmpL_i, ..., \sum_{i=0}^{N} CmpL_i\} \end{aligned} \tag{3}$$

where $N$ denotes the number of blocks in the scientific dataset $D$.

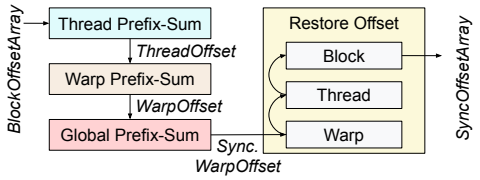



**Figure 9: Workflow of Global Synchronization in cuSZp**

In cuSZp, Global Synchronization is implemented in a hierarchical fashion, including thread-level, warp-level, and global-level (i.e. device-level), exploiting the NVIDIA GPU parallelism. Figure 9 illustrates the workflow of Global Synchronization in cuSZp. Given the input *BlockOffsetArray*, cuSZp first performs thread-level prefix-sum, which generates thread offset by iteratively adding up all block offsets that belong to the same thread. cuSZp utilizes one thread to operate multiple blocks, to achieve higher throughput with more register computations. Then, cuSZp performs warp-level prefix-sum by warp shuffle, generating warp offset, which is the local prefix-sum in a warp (i.e. 32 consecutive threads in the same thread block). Warp shuffle is a CUDA feature that allows threads within the same warp to exchange data with neglect latency (only slightly slower than register accesses). Based on our performance characterization, we set only one warp for each thread block, hence this step also completes the thread block-level prefix-sum. In the next step, cuSZp writes the warp offsets to a global variable to perform global prefix-sum and generates the synchronized warp offsets. Similar to existing solutions [22, 37], we implement the global prefix-sum using chained-scan parallelization, which is a single-pass approach to decode linear recurrences at a fine granularity. Since massive local prefix-sum computations are assigned to thread- and warp-level, the global variable accesses in this step are significantly reduced, hence increasing the performance. At last, cuSZp restores the offset to each block reversely by synchronized warp offset, generating the output *SyncOffsetArray*, where the last element denotes the compressed size of the whole dataset. To the best of our knowledge, we are arguably the first work that implements the global synchronization inside a GPU error-bounded lossy compressor (also within one kernel) and fine-tunes the performance via a hierarchical design.

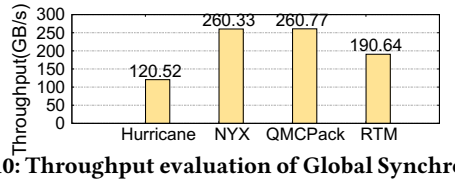



**Figure 10: Throughput evaluation of Global Synchronization.**

We profile the Global Synchronization in cuSZp performance on an NVIDIA A100 GPU with 4 scientific datasets. The selected datasets are the same as what we used in Figure 1 and 7. Figure 10 shows the performance, which is calculated by dividing the original data size (by GB) and runtime (by second). As we can see, our fine-tuned Global Synchronization can achieve on average of 208.06 GB/s throughput, varying from 120.52 GB/s in Hurricane to 260.77 GB/s to QMCPack due to intrinsic data characteristics, demonstrating our efficient Global Synchronization design.

## 4.4 Block Bit-shuffle (❹)

Block Bit-shuffle rearranges compressed fixed-length encoded integers before storing them in final compression memory space according to the indexes generated from Global Synchronization. The goal of Block Bit-shuffle is to transform encoded integers (lengths are different across different blocks) into a set of aligned bytes that can be easily stored. This design avoids irregular bit-shifting computation when the fixed-length is not divisible by 8, optimizing the program control flow, hence making this step a highly parallel process that can be well-suited for GPU implementation.

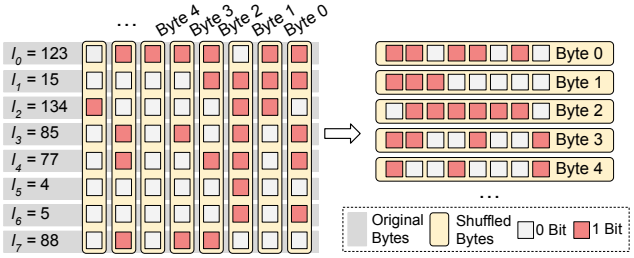



**Figure 11: Illustrating Block Bit-shuffle using example block from Figure 5.**

Figure 11 illustrates Block Bit-shuffle with a fixed-length encoded block ($L = 8$) obtained from Figure 5. Instead of directly storing the encoded integers $\{l_0, l_1, ..., l_7\}$, Block Bit-shuffle rearranges the bits with the $k$-th offset into Byte $k$. For example, the first bits from $\{l_0, l_1, ..., l_7\}$ are stored into Byte 0. For blocks with different lengths, the number of Bytes that store bits with the same offset can be calculated as $L/8$. After those shuffled bytes are generated, they will be stored in the final compression data, based on the indexes generated from Global synchronization.

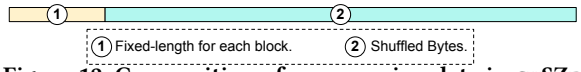



**Figure 12: Composition of compression data in cuSZp.**

Figure 12 shows the composition of compression data for the whole dataset $D$. There are two fractions in the compression data: Fixed-length for each block (①) and shuffled bytes for each block (②). Note that the *block offset array* mentioned in Figure 2 can be easily computed to this fixed-length information (①) based on Equation 2. In our implementation, we use one byte to store the fixed-length for each block, and this is utilized to guide Global Synchronization in cuSZp decompression.

# 5 EVALUATION

In this Section, we evaluate the performance of cuSZp with 6 representative scientific datasets on a supercomputer.

| Datasets | Dims per field | No. of fields | Total Size |
|---|---|---|---|
| Hurricane [1] | 500x500x100 | 13 | 1.3 GB |
| NYX [3] | 512x512x512 | 6 | 3.1 GB |
| QMCPack [17] | 288x115x69x69 | 2 | 1.2 GB |
| RTM [4] | 449x449x235 | 36 | 6.4 GB |
| HACC [9] | 280,953,867 | 6 | 6.3 GB |
| CESM-ATM [16] | 1,800x3,600 | 79 | 2.0 GB |

**Table 2: Real-world HPC datasets used in our evaluation.**

## 5.1 Experimental Setup

*5.1.1 Platforms.* We evaluate cuSZp on one NVIDIA Ampere A100 GPU (108 SMs, 40 GB), which is offered by Argonne Swing cluster[2]. This node is also equipped with 2 AMD EPYC 7742 CPUs (64C, 128T) @2.25GHz and 1 TB DDR4 host memory. The host operating system is Ubuntu 20.04.2 LTS and our GPU implementations are based on CUDA 11.4 toolkit and NVCC V11.4.152. We measure the GPU kernel performance using the Nsight system 2021.3.2.4, which is a professional profiling tool developed by NVIDIA.

*5.1.2 Dataset.* We conduct experiments based on 6 real-world HPC simulation datasets in various domains (shown in Table 2) from the Scientific Data Reduction Benchmarks [42]: Hurricane (weather simulation) [1], NYX (cosmology simulation) [3], QMCPack (quantum computing) [17], RTM (seismic imaging) [5], HACC (cosmic simulation) [9], and CESM-ATM (climate simulation) [16]. These datasets are widely adopted to evaluate lossy compressors in the data reduction community [28, 33, 41, 42].

*5.1.3 Evaluation Metrics.* Recall that our goal is to propose a GPU error-bounded lossy compressor that can achieve high throughput, high compression ratio, and high reconstructed data quality (Section 2.2), We formulate the related evaluation metrics in detail, which can be found as follows.

- **End-to-end throughput**: End-to-end throughput (GB/s) denotes how many gigabytes of data a compressor can process during duration between the timestamp when the original data are generated in GPU and the moment when the compressed data are stored back to GPU; and vice versa for decompression. The end-to-end throughput is very important for GPU HPC simulation.
- **Kernel throughput**: Kernel throughput (GB/s) denotes how many gigabytes of data a compressor can process in kernel execution time. If the compressor is executed in multiple kernels, this time can be calculated by adding up each kernel execution time.
- **Compression ratio (CR)**: Compression ratio can be calculated by $Size_{\text{ori}}/Size_{\text{cmp}}$, where $Size_{ori}$ and $Size_{cmp}$ denote the original and compressed data size, respectively. A higher compression ratio indicates the compressor has a stronger capability to aggregate information from the original data.
- **Rate distortion**: Rate distortion evaluates the reconstructed data quality under the same bit rate (i.e. the average number of bits per data point in compressed data). To quantify the data quality, we use both PSNR and SSIM (*c.f.* Section 2.2). Rate distortion is an important metric to statistically evaluate the data quality of a lossy compressor, whether this compressor is error-bounded (e.g. cuSZp, cuSZ, cuSZx) or fix-rated (cuZFP).
- **Visualization under the same CR**: In visualization evaluation, we compare the visualized data quality under the same compression ratio, for a fair comparison.

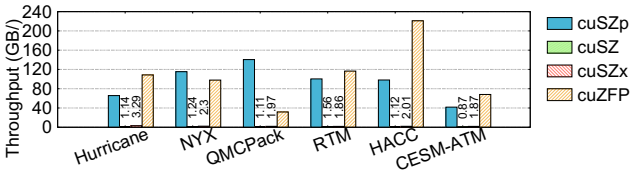
*5.1.4 Baseline.* We evaluate cuSZp with three state-of-the-art GPU lossy compressors, including cuZFP [21], cuSZ [33], and cuSZx [39]. Specifically, cuSZp, cuSZ, and cuSZx are error-bounded lossy compressors, and we evaluate them with 4 commonly used relative error bounds (definition can be found in 2.1), which are REL 1E-1, REL 1E-2, REL 1E-3, and REL 1E-4. cuZFP is a fast single kernel GPU lossy compressor with only fixed-rate mode supported. We exclude Bitcomp [2] in this evaluation as it is a closed-source implementation with an unknown compression algorithm.

## 5.2 Throughput

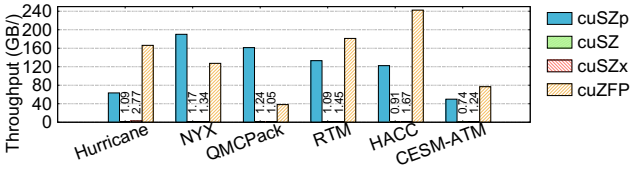
We evaluate the throughput of cuSZp in this Section. For error-bounded compressors (e.g. cuSZp, cuSZ, cuSZx), we measure the average throughput across the error-bounds REL 1E-1, REL 1E-2, REL 1E-3, and REL 1E-4 for each dataset. For cuZFP with only fixed-rate mode, we measure the average throughput across the fixed rates 4, 8, 16, and 24. First, we measure the end-to-end throughput of four compressors along with a performance breakdown analysis, which is a novel assessment way compared with other prior lossy compression research [7, 33, 39] to our knowledge. Then, we evaluate the kernel throughput, and this evaluation method is consistent with other GPU lossy compression work [7, 33, 39].

Figure 13 presents the end-to-end throughput for compression and decompression. We observe that cuSZp and cuZFP can achieve top-tier end-to-end performance, due to their single GPU kernel design, which significantly outperforms cuSZ and cuSZx (~100x). On average, cuSZp can achieve 93.63 GB/s and 120.04 GB/s for end-to-end compression and decompression throughput, respectively. Specifically, compression throughput in cuSZp varies from 41.77 GB/s in CESM-ATM to 140.44 GB/s in QMCPack, whereas this variation is from 49.91 in CESM-ATM to 190.11 in NYX for cuSZp decompression. In comparison, cuSZ and cuSZx can only achieve 1.04 GB/s~2.22 GB/s end-to-end throughput. The key reason is that both cuSZ and cuSZx require CPU computations to perform the linear recurrences which are hard to parallelize on GPU, such as building Huffman in cuSZ and global synchronization in cuSZx. Such operations introduce significant data movement cost between CPU and GPU, hence reducing the end-to-end performance inevitably. All in all, compared with state-of-the-art error-bounded GPU lossy compressors, cuSZp can improve the end-to-end performance by 95.53x with cuSZ and by 55.18× with cuSZx.

[2]https://www.lcrc.anl.gov/systems/resources/swing/



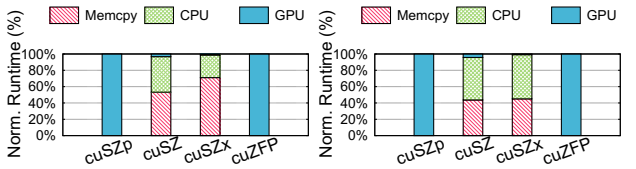

(a) End-to-end compression throughput.




(b) End-to-end decompression throughput.

**Figure 13: End-to-end throughput (GB/s).**

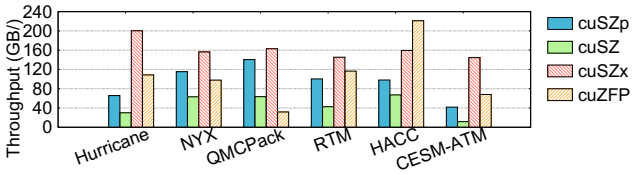
In order to understand the end-to-end throughput deeply, we do a performance breakdown using the dataset Hurricane (Field: U), and the results are shown in Figure 14. We normalize the compression/ decompression runtime into 100% and profile the percentage for each component. In general, the end-to-end throughput can be roughly divided into three parts, including GPU computations (denoted as GPU), CPU computations (denoted as CPU), and data movement overheads (denoted as Memcpy). Since cuSZp and cuZFP can be finished in one kernel, we can observe their GPU computations occupies 100% runtime. However, in cuSZ, the GPU computations take only 3.24% and 4.21% for the end-to-end compression and decompression, which indicates users have to spend over 20× runtime overhead if they adopt cuSZ for inline GPU HPC simulation in practice. Similar results can be observed in cuSZx. We can also find that cuSZx has higher CPU computation overheads in decompression. The reason is that cuSZx requires both CPU preprocessing and postprocessing in decompression, compared with only postprocessing in compression. All in all, the end-to-end throughput in CPU-GPU hybrid solution suffers from CPU computation and data movement overheads, making a single-kernel design extremely important for GPU HPC simulation.



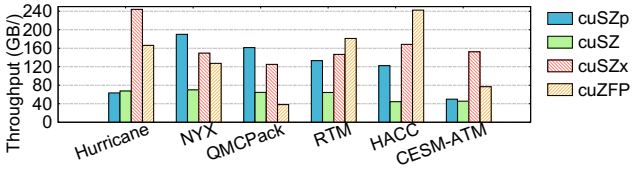

(a) Compression (b) Decompression

**Figure 14: Breakdown performance analysis for end-to-end throughput. Dataset: Hurricane (Field: U).**

To understand the effectiveness of our kernel design as well as implementation, we also measure the kernel throughput for all four GPU compressors, as presented in Figure 15. The kernel throughput we measured in our experiments is inline with the results shown in other existing works [21, 33, 39]. Specifically, cuSZ and cuSZx can have relatively high throughput. CuSZx can achieve an average of 161.51 GB/s in compression and 164.40 GB/s in decompression, whereas these numbers are 46.39 GB/s and 59.44 GB/s for cuSZ.




(a) Kernel compression throughput.




(b) Kernel decompression throughput.

**Figure 15: Kernel throughput (GB/s).**

The throughput of cuSZp and cuZFP remain the same between end-to-end throughput and kernel throughput, due to a single-kernel design. Note that even without CPU computations and expensive data movement overheads, cuSZp still increases the kernel throughput by over 100% compared with cuSZ, validating the effectiveness of the ultra-fast design in cuSZp.

## 5.3 Compression Ratio

Table 3 presents the compression ratio results of 3 error-bounded lossy compressors. The highest average compression ratio for each dataset under each error bound is underlined. We exclude the cuZFP in this table because it does not support error bound mode but only fixed-rate mode (its compression ratio is a fixed number, i.e. each data point preserves the same number of bits). For each compressor, same as Section 5.2, we adopt REL 1E-1, REL 1E-2, REL 1E-3, and REL 1E-4 error bounds. There are several results marked as "n/a" in Table 3, because the cuSZ would crash at those error bounds probably due to the bugs in its implementation. We confirm that the bugs are related to storing the Huffman codebooks in cuSZ, by communicating with the cuSZ developers.

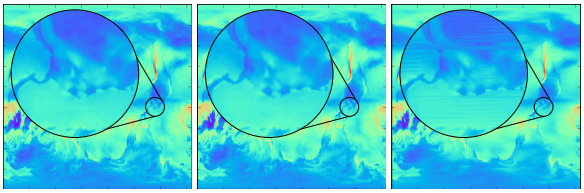
We can observe that cuSZp achieves the highest compression ratios on 16/24 benchmarks with different error bounds, even though cuSZp is devised under the single kernel design constraint. In absolute terms, for the Hurricane dataset, cuSZp improves the average compression ratios by 162.61%, 71.82%, 39.70%, and 39.55% in REL 1E-1, REL 1E-2, REL 1E-3, and REL 1E-4 error bounds, respectively, compared with cuSZ. In the RTM dataset, cuSZp improves the average compression ratios by 41.45%, 78.78%, 78.60%, and 49.29 % in the same four error bounds compared with cuSZx.

For the HACC dataset, cuSZx exhibits higher compression ratios compared with cuSZp based on the error bounds of REL 1E-1 and REL 1E-2. This is because HACC has a relatively large value range (e.g., 7614.87 in the vx field), making the error bounds of REL 1E-1 and REL 1E-2 relatively large, generating more constant blocks in the cuSZx compression phase, hence increasing the compression ratios. This also explains why cuSZx can achieve the best compression ratios in CESM-ATM dataset.

However, cuSZx's high compression ratio is achieved at the price of significantly degraded reconstructed data quality. Figure 16

**Table 3: Compression ratio of 3 error-bounded lossy compressors in GPU. The highest average compression ratios are underlined.**

| | | Hurricane | | | NYX | | | QMCPack | | | RTM | | | HACC | | | CESM-ATM | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **REL** | min | max | avg | min | max | avg | min | max | avg | min | max | avg | min | max | avg | min | max | avg |
| **cuSZp** | 1E-1 | 13.56 | 124.32 | 75.45 | 43.48 | 127.99 | 99.11 | 85.20 | 98.25 | 91.73 | 72.76 | 127.99 | 108.48 | 10.35 | 59.82 | 34.30 | 3.99 | 101.27 | 27.40 |
| | 1E-2 | 5.96 | 88.88 | 38.71 | 9.62 | 127.80 | 66.74 | 12.46 | 22.23 | 17.35 | 13.89 | 127.96 | 67.06 | 5.24 | 10.09 | 7.63 | 2.93 | 43.75 | 14.21 |
| | 1E-3 | 3.72 | 56.88 | 22.32 | 5.10 | 125.55 | 38.46 | 6.08 | 10.08 | 8.08 | 6.88 | 127.83 | 42.40 | 3.43 | 5.20 | 4.31 | 2.31 | 33.81 | 9.82 |
| | 1E-4 | 2.71 | 36.66 | 14.36 | 3.36 | 98.25 | 22.15 | 3.79 | 5.57 | 4.68 | 4.16 | 127.59 | 27.56 | 2.53 | 3.39 | 2.96 | 1.81 | 26.11 | 7.35 |
| **cuSZ** | 1E-1 | 26.42 | 29.98 | 28.73 | 31.24 | 31.58 | 31.47 | 19.41 | 23.41 | 21.41 | 29.47 | 30.87 | 30.45 | 30.31 | 31.30 | 30.81 | 23.31 | 25.43 | 24.63 |
| | 1E-2 | 15.35 | 28.62 | 22.53 | 28.71 | 31.57 | 30.22 | 7.50 | 21.55 | 14.53 | n/a | n/a | n/a | n/a | n/a | n/a | 19.18 | 25.33 | 22.89 |
| | 1E-3 | 8.91 | 23.61 | 15.97 | n/a | n/a | n/a | 4.26 | 17.70 | 10.98 | n/a | n/a | n/a | n/a | n/a | n/a | 11.34 | 25.16 | 18.48 |
| | 1E-4 | 3.37 | 17.25 | 8.36 | 10.75 | 31.28 | 16.22 | n/a | n/a | n/a | 3.67 | 30.84 | 11.63 | n/a | n/a | n/a | 5.38 | 24.43 | 12.47 |
| **cuSZx** | 1E-1 | 28.68 | 118.27 | 74.19 | 77.09 | 124.10 | 110.74 | 25.59 | 69.21 | 47.40 | 23.36 | 124.06 | 76.69 | 28.81 | 124.08 | 70.41 | 16.05 | 124.11 | 74.30 |
| | 1E-2 | 3.91 | 53.92 | 21.67 | 4.68 | 123.72 | 61.43 | 2.74 | 9.01 | 5.88 | 3.94 | 123.92 | 37.51 | 3.05 | 117.57 | 44.37 | 3.93 | 124.11 | 31.85 |
| | 1E-3 | 2.86 | 32.03 | 13.47 | 3.11 | 118.93 | 30.37 | 2.36 | 4.31 | 3.34 | 2.83 | 123.55 | 23.74 | 2.18 | 4.31 | 3.00 | 2.77 | 123.96 | 24.24 |
| | 1E-4 | 2.03 | 23.64 | 10.29 | 2.38 | 74.36 | 15.12 | 1.68 | 2.84 | 2.26 | 2.17 | 123.00 | 18.46 | 1.70 | 2.68 | 2.13 | 2.11 | 123.77 | 22.57 |



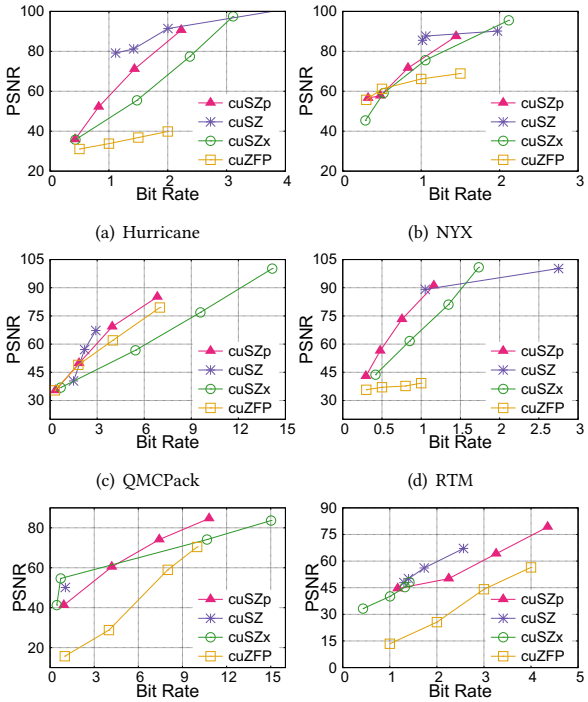
(a) Original (b) cuSZp (CR=~6.7) (c) cuSZx, (CR=~6.7)

**Figure 16: Visualization of CESM-ATM**

demonstrates the visualization of reconstructed data for a dataset field from the CESM-ATM application under the same compression ratio. It is clearly observed that the reconstructed data by cuSZx suffers from horizontal stripe artifacts, and we analyze the key reason as follows. In fact, in cuSZx, the first step is splitting the whole dataset into many fixed-length blocks (e.g., 128 data points per block) and the data values would be flushed as the range-middle-point (i.e., mean of min and max in this block) if all the data points in a block can be substituted by this middle-point according to the given error bound. This would introduce horizontal stripe artifacts especially when the data are relatively smooth and the error bound is relatively large.

## 5.4 Data Quality

Figure 17 and 18 present the rate distortion with respect to PSNR and SSIM, respectively. Here the x-axis denotes the bit rate, which is the mean number of required bits per data point in compressed data, so a lower bit rate indicates a higher compression ratio. The y-axis represents the PSNR or SSIM. For each of the three error-bounded compressors, we measure the PSNR and SSIM at the four selected error bounds used in Table 3 and calculate bit rates based on their compression ratios. For cuZFP(fixed-rate mode), we run it with the 4 fixed rate settings that are close to the measured bit rates of cuSZp's compressed data and then measure the PSNR and SSIM, for a fair comparison. As we can see, compared with other lossy compressors, cuSZp can preserve a higher fidelity in reconstructed data, achieving the highest PSNR and SSIM on certain fields such as RTM and HACC. Specifically, cuSZp significantly improves the PSNR and SSIM under the same bit rates over cuZFP especially because of its effective error controls. It is worth noting that cuZFP's PSNR and SSIM are very low for HACC dataset: only 28.77 dB and 0.1465, respectively, when the bit rate is set to 4, while these

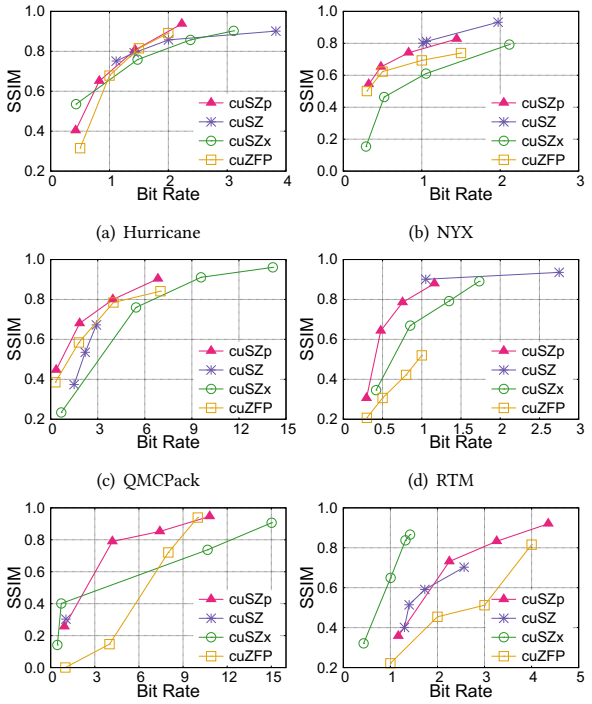



(a) Hurricane (b) NYX (c) QMCPack (d) RTM (e) HACC (f) CESM-ATM

**Figure 17: Rate Distortion (PSNR)**

numbers are up to 60.42 dB and 0.7892 in cuSZp. This is because cuZFP's orthogonal transform works particularly effective on multi-dimensional datasets instead of 1D array. As verified via multi-dimensional datasets such as Hurrican and Nyx (see Figure 17(a) and (b)), we can observe cuZFP achieves relatively competitive SSIM. In comparison, cuSZ exhibits a high reconstructed data quality, mainly because cuSZ adopts both multi-dimensional Lorenzo prediction and Huffman encoding, which is a nearly optimal variable-length encoder to preserve data fidelity. However, it requires expensive operations to build and store Huffman trees, drastically reducing the end-to-end throughput in turn, as demonstrated in Figure 13.

Figure 19 presents visualization results between cuSZp and cuZFP under the same error bounds. The fields and datasets we use are the same as Figure 7. We can observe that cuSZp can achieve




(a) Hurricane (b) NYX (c) QMCPack (d) RTM (e) HACC (f) CESM-ATM

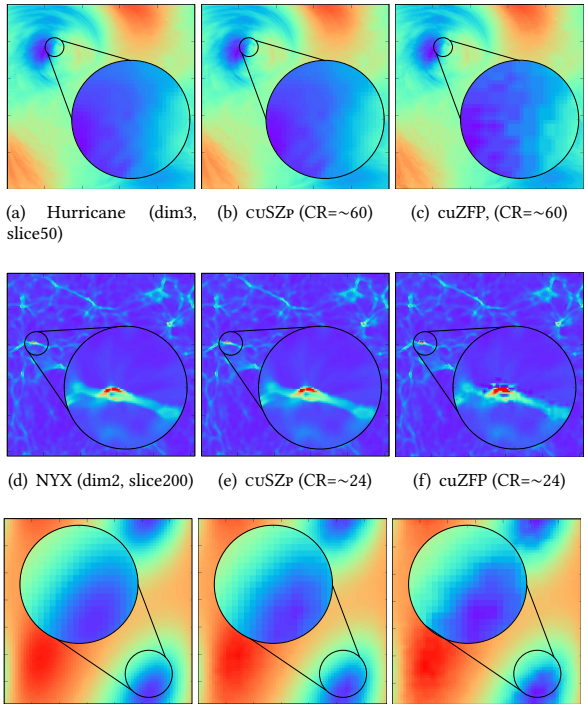
**Figure 18: Rate Distortion (SSIM)**

high visualization quality in three selected fields, preserving almost identical data patterns. In comparison, cuZFP also can preserve similar overall data patterns, but it introduces visible distortions, such as the distorted wavefield in NYX (Figure 19(f)) and blocky artifacts in Hurricane (Figure 19(c)). The reason can be explained as follows: when the compression ratio requirement is relatively high (e.g., >16), cuZFP preserves very few bits (≤2) per data point uniformly, inevitably losing fidelity in consecutive data points with sharp value differences. Such an observation demonstrates that cuSZp has a strong capability in maintaining data quality from the perspective of visualization.

Although visualization by slice has been widely adopted by existing lossy compression works [28, 39–41], we also perform isosurface visualization to further evaluate the reconstructed data quality by cuSZp. Figure 20 shows the results, where we visualize the same NYX field in Figure 19 and the isovalue is set as 0. When the compression ratio is set as ~8, cuSZp can obtain almost identical patterns compared with the original visualization, whereas the visualization reconstructed by cuZFP has visible artifacts. Such conclusions can also be drawn for fields in other datasets and are consistent with previous results, demonstrating the superiority of cuSZp in maintaining data quality.

## 6 DISCUSSION

**Breakdown Performance for cuSZp Kernel Throughput:** We perform breakdown performance analysis for cuSZp kernel throughput, of which results are shown in Figure 21. We use REL 1E-2 error bound here, and the results are consistent across other error bounds.

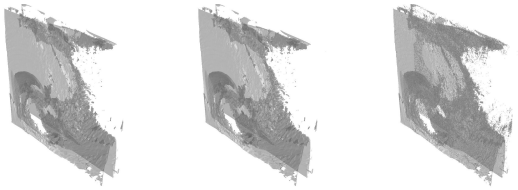


(a) Hurricane (dim3, slice50) (b) cuSZp (CR=~60) (c) cuZFP, (CR=~60)

(d) NYX (dim2, slice200) (e) cuSZp (CR=~24) (f) cuZFP (CR=~24)

(g) QMCPack (dim3, slice1000) (h) cuSZp (CR=~36) (i) cuZFP (CR=~36)

**Figure 19: Visualization of original data and reconstructed data decompressed by cuSZp and cuZFP, respectively.**

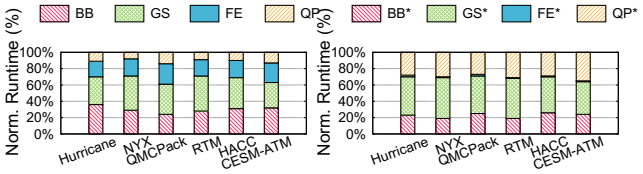


(a) Hurricane (dim3, slice50) (b) cuSZp (CR=~8) (c) cuZFP, (CR=~8)

**Figure 20: Isosurface visualization of original data and reconstructed data decompressed by cuSZp and cuZFP.**

"BB", "GS", "FE", and "QP" denote Block Bit-shuffle, Global Synchronization, Fixed-length Encoding, and Quantization and Prediction, respectively. "*" in Figure 21(b) means the reverse operation in decompression. As seen, in compression kernel (Figure 21(a)), Block Bit-shuffle, Global Synchronization, and Fixed-length Encoding are more dominant in throughput, with 21.67%, 37.50%, and 30.00%. The reason is that all these 3 steps include global memory access, which is more time-consuming than computations. Specifically, Block Bit-shuffle stores the compressed block data, Global Synchronization requires flag and status information stored in global memory, whereas Fixed-length Encoding saves the fixed-length for each block as illustrated in Figure 12. In cuSZp decompression kernel, the results are different (Figure 21(b)). Block Bit-shuffle, Global Synchronization, and Quantization and Prediction occupy more dominant time. The reason is that, in decompression, all operations are reversed – all read from global memory in compression

turns to be write operations in decompression. Additionally, the Global Synchronization reads the stored Fixed-length information from global memory, making Fixed-length Encoding itself a very lightweight step in decompression.




(a) Compression (b) Decompression

**Figure 21: Breakdown performance analysis for cuSZp kernel throughput with REL 1E-2.**

**cuSZp with Time-Varying Simulations:** We evaluate cuSZp with a time-varying HPC simulation RTM from seismic imaging domain [13]. The RTM executes for 3600 timesteps, generating 1 snapshot for each timestep. We select 1 snapshot every 100 timesteps in the overall RTM execution and perform cuSZp compression and decompression to explore its compatibility. The results are shown in Figure 22. We observe that compression and decompression throughput decreases as timestep increases. The cause for this is, the value ranges of the generated snapshots decrease along with time, due to the intrinsic design in RTM, leading to fewer zero blocks while using REL error mode. Such will also decrease the throughput over time. Hence, cuSZp is more efficient in processing sparse HPC data. In all, due to its single-kernel design, cuSZp is highly compatible with time-varying HPC simulations.

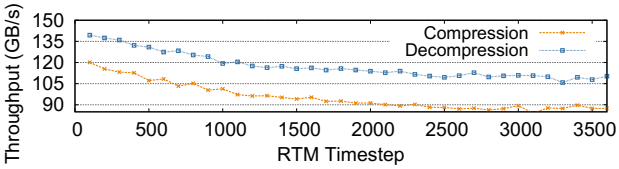



**Figure 22: cuSZp with time-varying HPC simulation.**

**Compatibility with Other Lower-End GPUs:** Beyond NVIDIA A100 GPU evaluated in this work, cuSZp is also compatible with other lower-end GPUs, such as V100 or RTX 3080 (10 GB VRAM). Still taking one RTM snapshot as an example, the kernel throughput for compression is 100.34, 87.44, and 80.13 GB/s on A100, V100, and RTX 3080 GPUs, respectively. The differences are due to the specific memory bandwidth in hardware specification, leading to different throughput while accessing global memory.

## 7 RELATED WORKS

There have been some works that target optimizing data compression in GPU during the past decade [7, 21, 33, 38, 39]. Yang et al. [38] proposed a GPU lossless compressor MPC for IEEE 754 floating point, but it provides only a limited compression ratio for HPC floating-point data with significant randomness. Lindstrom et al. [21] implemented ZFP algorithm, including transform and bit truncation, to NVIDIA GPU with a single kernel, achieving an ultra-fast compression and decompression throughput. However, cuZFP only supports fixed-rate and hence suffers performance [34]. Tian et al. [33] proposed cuSZ, which is a prediction-based error-bounded GPU lossy compression framework. Soon after that, Tian et al. [32] introduced a fine-grained variable length encoding strategy to cuSZ to further improve the compression ratios. Yu et al. [39] combined a block-wise design with lightweight bit-level operations and proposed an error-bounded compressor cuSZx, achieving an ultra-fast kernel throughput. Compared with existing works, our proposed cuSZp is the first error-bounded lossy compressor putting the entire computation into one kernel with a series of optimization strategies, which can achieve high throughput, high compression ratios, and high data quality (comprehensively evaluated by PSNR, SSIM and visualization) meanwhile.

## 8 CONCLUSION AND FUTURE WORKS

In this paper, we propose cuSZp, a GPU error-bounded lossy compressor with optimized end-to-end throughput. cuSZp consists of a bunch of lightweight bit-level operations in register with a fine-tuned global synchronization, to put the entire computation into a single GPU kernel, hence achieving ultra-fast throughput. We perform extensive experiments on four GPU lossy compressors, including cuSZp, cuZFP, cuSZx, and cuSZ, with 6 real-world scientific datasets. The main results are summarized as follows:

- Compared with two error-bounded GPU lossy compressors cuSZ and cuSZx, cuSZp improves the end-to-end performance by 95.53x and 55.18x. As for kernel throughput, cuSZp can achieve on average 93.63 GB/s and 120.04 GB/s for compression and decompression, which is faster than cuSZ and inline with cuSZx and cuZFP.
- cuSZp can obtain the best compression ratios, compared with cuSZ and cuSZx, on 16/24 cases while still preserving high-quality reconstructed data and an ultra-fast throughput.
- Compared with cuZFP, cuSZp not only improves the rate distortion for both PSNR and SSIM but also provides better visualization images.

In the future, we will explore this work in two directions. First, we plan to further improve the performance of cuSZp by exploiting the hardware resources in most latest NVIDIA GPU infrastructure, such as Hopper. Second, we aim to integrate cuSZp into real-world scientific simulations that have high-speed demands.

## ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant OAC-2003709, OAC-2104023, OAC-2211538, and OAC-2211539/2247060. We acknowledge the computing resources provided on Bebop (operated by Laboratory Computing Resource Center at Argonne) and on Theta and JLSE (operated by Argonne Leadership Computing Facility). We acknowledge the support of ARAMCO.

## REFERENCES


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

[2] [n. d.]. NVCOMP. https://github.com/NVIDIA/nvcomp

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

[4] 2021. Parallel RTM code by Saudi Aramco.

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

[6] 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 forfloating-point data in scientific datasets. *International Journal of High Performance Computing Applications (IJHPCA)* (2019).

[7] Jieyang Chen, Lipeng Wan, Xin Liang, Ben Whitney, Qing Liu, David Pugmire, Nicholas Thompson, Jong Youl Choi, Matthew Wolf, Todd Munson, et al. 2021. Accelerating multigrid-based hierarchical scientific data refactoring on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 859–868.

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

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

[10] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[11] Alain Hore and Djemel Ziou. 2010. Image quality metrics: PSNR vs. SSIM. In *2010 20th international conference on pattern recognition*. IEEE, 2366–2369.

[12] Yafan Huang, Shengjian Guo, Sheng Di, Guanpeng Li, and Franck Cappello. 2022. Mitigating silent data corruptions in HPC applications across multiple program inputs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

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

[14] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, Vol. 22. Wiley Online Library, 343–348.

[15] Jennifer E Kay, Allison H Baker, Dorit Hammerling, Sheri A Michelson, Haiying Xu, et al. 2016. Evaluating lossy data compression on climate simulation data within a large ensemble. *Geoscientific Model Development* 9, 12 (2016).

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

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

[18] Xin Liang, Sheng Di, Franck Cappello, Mukund Raj, Chunhui Liu, Kenji Ono, Zizhong Chen, Tom Peterka, and Hanqi Guo. 2022. Toward Feature-Preserving Vector Field Compression. *IEEE Transactions on Visualization and Computer Graphics* (2022), 1–16. https://doi.org/10.1109/TVCG.2022.3214821

[19] Xin Liang, Sheng Di, Sihuan Li, Dingwen Tao, Bogdan Nicolae, Zizhong Chen, and Franck Cappello. 2019. Significantly improving lossy compression quality based on an optimized hybrid prediction model. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–26.

[20] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. In *2018 IEEE International Conference on Big Data (Big Data)*. 438–447. https://doi.org/10.1109/BigData.2018.8622520

[21] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

[22] Sepideh Maleki and Martin Burtscher. 2018. Automatic Hierarchical Parallelization of Linear Recurrences. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 128–138.

[23] Md Hasanur Rahman, Sheng Di, Kai Zhao, Robert Underwood, Guanpeng Li, and Franck Cappello. 2023. A Feature-Driven Fixed-Ratio Lossy Compression Framework for Real-World Scientific Datasets. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1461–1474.

[24] P. Ratanaworabhan, Jian Ke, and M. Burtscher. 2006. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*. 133–142. https://doi.org/10.1109/DCC.2006.35

[25] Etienne Robein. November 15, 2016. EAGE E-Lecture: Reverse Time Migration: How Does It Work, When To Use It. https://youtu.be/ywdML8ndYeQ.

[26] Seung Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Weikeng Liao, and Alok Choudhary. 2014. Data Compression for the Exascale Computing Era - Survey. *Supercomput. Front. Innov.: Int. J.* 1, 2 (jul 2014), 76–88. https://doi.org/10.14529/jsfi140205

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

[28] SZ2.1. 2022. https://github.com/szcompressor/SZ.

[29] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. In-depth exploration of single-snapshot lossy compression techniques for N-body simulations. In *2017 IEEE International Conference on Big Data (Big Data)*. 486–493. https://doi.org/10.1109/BigData.2017.8257962

[30] 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.

[31] 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

[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] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, et al. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 3–15.

[34] 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. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 567–577.

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

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

[37] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 229–238.

[38] Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher. 2015. MPC: a massively parallel compression algorithm for scientific data. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 381–389.

[39] Xiaodong Yu, Sheng Di, Kai Zhao, Jiannan Tian, Dingwen Tao, Xin Liang, and Franck Cappello. 2022. Ultrafast Error-bounded Lossy Compression for Scientific Datasets. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 159–171.

[40] Kai Zhao, Sheng Di, Perez Danny, Zizhong Chen, and Franck Cappello. 2022. MDZ: An Efficient Error-bounded Lossy Compressor for Molecular Dynamics Simulations. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*.

[41] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1643–1654. https://doi.org/10.1109/ICDE51399.2021.00145

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

[43] Hua-Wei Zhou, Hao Hu, Zhihui Zou, Yukai Wo, and Oong Youn. 2018. Reverse time migration: A prospect of seismic imaging methodology. *Earth-Science Reviews* 179 (2018), 207–227. https://doi.org/10.1016/j.earscirev.2018.02.008

# Appendix: Artifact Description/Artifact Evaluation

## ARTIFACT DOI

https://github.com/szcompressor/cuSZp

## ARTIFACT IDENTIFICATION

This paper proposes CompX, which is an ultra-fast error-bounded GPU compressor for NVIDIA GPU. The workflow of CompX consists of 4 parts: (1) Quantization and Prediction, (2) Fixed-length Encoding, (3) Global Synchronization, and (4) Block Bit-shuffle. CompX performs compression or decompression in only one GPU kernel function, significantly improving the end-to-end performance compared with existing lossy compressors.

The experiments to evaluate CompX in this paper can be divided into 4 parts: (1) end-to-end throughput evaluation, (2) kernel throughput evaluation, (3) compression ratio evaluation, and (4) data quality evaluation. The evaluation in this paper is conducted under NVIDIA A100 GPU with CUDA 11.4 Toolkit.

Note that CompX has another name cuSZp, which belongs to the SZ lossy compression family. Due to copyright with our collaborators, the source code is temporarily not available, but it will soon be open-source in this link: https://github.com/szcompressor/cuSZp once the permission is granted.

## REPRODUCIBILITY OF EXPERIMENTS

**Artifact Persistent ID**: https://github.com/szcompressor/cuSZp

**Artifact name**: CompX

**Evaluation dataset**: https://sdrbench.github.io/

**Relevant hardware and software**: CPU: AMD EPYC 7742 CPU @2.25GHz; GPU: NVIDIA A100 GPU (108 SMs, 40 GB); RAM: 128GB DDR4; OS: Ubuntu 20.04.2 LTS; Software dependencies: CUDA 11.4 tookit, NVCC V11.4.152, Nsight system 2021.3.2.4.

**Descriptions**:

- Experiment workflow: First compile the source code into executable binary, then perform compression and decompression with the executable binary on the given scientific dataset.
- Estimation execution time: Less than 1 second, since the design of CompX targets to finish the compression and decompression with an ultra-fast speed.
- Expected results and evaluation: There are three parts to evaluate a CompX lossy compressor in this work: (1) End-to-end throughput (i.e. kernel throughput since CompX is a single kernel design), (2) Compression ratio, and (3) reconstructed data quality.
- Results between experiment workflow and paper: The results should be consistent with the data that is provided in the Evaluation section (i.e. Sec. V) of this paper.

## ARTIFACT DEPENDENCIES REQUIREMENTS

Due to the copyright issue with our collaborator, we cannot provide the source code of compressor CompX. However, to reproduce the results in this paper, we provide a prepared Docker image along with the executable binary and other visualization tools (such as QCAT). So the dependencies and requirements only include a Linux machine with Docker installed and an NVIDIA GPU. The NVIDIA GPU better be A100, so that the results will be consistent with the paper, but other lower-end NVIDIA GPUs (e.g. 3080 10GB) are also compatible.

To install and run the Docker we prepared, there are several steps:

- Download Docker image from DockerHub:
  *docker pull hyfshishen/sc23-compx-env*
- Start Docker image with NVIDIA GPU:
  *docker run –gpus all -it hyfshishen/sc23-compx-env /bin/bash*

Note that the second command "–gpus" contains two dashes. After you launch the Docker image as a running container, you are ready to run CompX.

## ARTIFACT INSTALLATION DEPLOYMENT PROCESS

After launching the Docker environment, the CompX compressor can be executed by demo command "*compx [Data-To-Be-Compressed] [Relative-Error-Bound]*". No installation and building processes are needed since *compx* command is already added into the local environment. Here we use, an NVIDIA 3080 GPU with NYX dataset from SDRBench, as an example to show how to obtain the CompX execution throughput, compression ratio, and data quality. It will be better to reproduce the experiments with an NVIDIA A100 GPU, and the results will be consistent with the paper.

First, move to root folder, and download and unzip NYX dataset from the link provided by SDRBench website:

```
$root@64e73de40a66:/# cd /root
$root@64e73de40a66:~# wget https://g-8d6b0.fd635.8443.data.globus.
$root@64e73de40a66:~# ls
SDRBENCH-EXASKY-NYX-512x512x512.tar.gz  software
$root@64e73de40a66:~# tar -xvf SDRBENCH-EXASKY-NYX-512x512x512.tar.
SDRBENCH-EXASKY-NYX-512x512x512/
SDRBENCH-EXASKY-NYX-512x512x512/dark_matter_density.f32
SDRBENCH-EXASKY-NYX-512x512x512/velocity_x.f32
SDRBENCH-EXASKY-NYX-512x512x512/template_data.txt
SDRBENCH-EXASKY-NYX-512x512x512/velocity_z.f32
SDRBENCH-EXASKY-NYX-512x512x512/temperature.f32
SDRBENCH-EXASKY-NYX-512x512x512/baryon_density.f32
SDRBENCH-EXASKY-NYX-512x512x512/velocity_y.f32
```

Then, use *temperature.f32* field as an example, we use CompX to compress it with REL 1E-4 error bound. The results can be shown as below:

```
$root@64e73de40a66:~# compx temperature.f32 1e-4
CompX Compression Kernel finished!
CompX Decompression Kernel finished!

CompX finished!
CompX Compression   end-to-end speed: 80.881768 GB/s
```

```
CompX Decompression end-to-end speed: 102.304867 GB/s
CompX Compression ratio: 6.624457

Pass error check!
```

The compression end-to-end (i.e. kernel, since CompX is a single kernel design) throughput, decompression end-to-end throughput, and the compression ratio will be printed on the screen. The throughput can also be checked using Nsight profiler, such as using command *nsys profile –stats=true compx tempreture.f32 1e-4*. And you will see, only two kernels (one for compression and one for decompression) are executed.

There will be two files generated after the CompX compression and decompression:

```
$root@64e73de40a66:~# ls
temperature.f32
temperature.f32.compx.cmp
temperature.f32.compx.dec
```

where xxx.compx.cmp denotes the compressed data and xxx.compx.dec denotes the reconstructed data.

```
$root@64e73de40a66:~# du -sh ./*
513M ./temperature.f32
78M ./temperature.f32.compx.cmp
513M ./temperature.f32.compx.dec
```

To obtain the data quality of the reconstructed data, such as PSNR, SSIM, and slice visualization. QCAT (already installed in this Docker image) can be used. To compute the PSNR between the original data and the reconstructed data, the commands can be shown below:

```
$root@64e73de40a66:~# compareData -f temperature.f32 temperature.f32.compx.dec
This is little-endian system.
reading data from temperature.f32
Min = 2280.975830078125, Max = 4782583.5, range = 4780302.524169921875
Max absolute error = 478.0390625000
Max relative error = 0.000100
Max pw relative error = 0.199965
PSNR = 84.770561, NRMSE = 5.773935552616106 6342E-05
normErr = 3197659.788566, normErr_norm = 0.015479
pearson coeff = 0.999845
```

Before computing SSIM, the dimension of the dataset is needed, and this information can be found in the dataset description table (in Sec.V) or SDRBench website. To compute the SSIM between the original data and the reconstructed data, the commands can be found below.

```
$root@64e73de40a66:~# calculateSSIM -f temperature.f32 temperature.f32.compx.dec 512 512 512
This is little-endian system.
reading data from temperature.f32
calcaulting....
ssim = 0.899964
```

To get a slice visualization of the reconstructed data, the command can be found below:

```
$root@64e73de40a66:~# PlotSliceImage -f -i temperature.f32.compx.dec -3 512 512 512 -m INDV -p 2 -s 200 -o test.png
Image file is plotted and put here: ./test.png
```

Note that plot slice visualization requires Gnuplot, which can be easily installed by “apt-get install -y gnuplot" inside the Docker container.

This is one example to reproduce the experiments in CompX paper, the rest dataset should also be the same way.