Pushing the Limits of GPU Lossy Compression: A Hierarchical Delta Approach

Boyuan Zhang
Indiana University
United States

Yafan Huang
University of Iowa
United States

Sheng Di[*]
Argonne National Laboratory
United States

Fengguang Song
Indiana University
United States

Guanpeng Li
University of Iowa
United States

Franck Cappello
Argonne National Laboratory
United States

Abstract

Modern GPU-based lossy compressors face significant challenges due to the imbalance in development between compression throughput and compression ratio. Most compressors tend to prioritize one metric while neglecting the other, leading to either redundant compressed files or inefficient kernel designs that hinder end-to-end throughput. This work introduces Aatrox, a generic single-kernel error-bounded lossy compressor specifically designed for GPUs, targeting applications that demand high performance and high compression ratio at the same time, such as large-scale quantum circuit simulations and large language model training. In particular, Aatrox features a novel hierarchical data blocking strategy, large-block delta encoding, and dual-level delta decoding, achieving substantial end-to-end throughput and an optimized compression ratio. Experiments on NVIDIA A100 GPU using nine real-world scientific datasets show that Aatrox achieves higher compression ratios while preserving high data quality, with an average throughput of 388.3 GB/s for compression and 718.0 GB/s for decompression. These results represent approximately 1.2× speedup compared to the throughput of existing pure-GPU compressors and 250× that of CPU-GPU hybrid compressors.

CCS Concepts

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

[*]Corresponding author: Sheng Di.

ACM ISBN 979-8-4007-1537-2/2025/06
https://doi.org/10.1145/3721145.3725743

Keywords

Parallel Computing, GPU, Data Reduction, Error-bounded Lossy Compression, In-situ Compression

ACM Reference Format:
Boyuan Zhang, Yafan Huang, Sheng Di, Fengguang Song, Guanpeng Li, and Franck Cappello. 2025. Pushing the Limits of GPU Lossy Compression: A Hierarchical Delta Approach. In *2025 International Conference on Supercomputing (ICS '25), June 8–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3725743

1 Introduction

The scale of scientific applications has grown larger than ever, creating a significant big data challenge for High-Performance Computing (HPC) systems. The large volume of data drives domain scientists to adopt efficient data reduction techniques. Lossy compression stands out due to its higher compression ratio compared to lossless compression. Error-bounded lossy compression enables users to customize the level of error, making it a popular solution in scientific applications such as climate simulation [8], materiel simulation [10], and cosmology simulation [27].

1.1 Motivation for High-Ratio, High-Throughput Compressor

Large-scale scientific applications running on HPC systems generate vast amounts of data for analysis, such as quantum circuit simulation. During the Noisy Intermediate-Scale Quantum (NISQ) era [45], quantum simulation remains critical for developing quantum algorithms and validating quantum computer results. However, the memory requirements for quantum circuit simulation [51, 55] grow exponentially with the number of qubits. Simulating a 48-qubit circuit would fully occupy the entire memory of Frontier [4] (4.6 petabytes of DDR4 memory), the second most advanced HPC system currently available [13]. Meanwhile, real-world quantum computers require validation for systems with more than 100 qubits [17]. Moreover, quantum simulation faces significant time overhead, including communication and computation overheads[31], highlighting the need for compressors with higher compression ratios and throughput. Another example is the benefit of lossy compression in Large Language Model (LLM) training. While quantization acceleration has been developed to reduce communication overhead and improve training performance, it faces limitations in the number of quantization bits. Recent work has utilized 4-bit quantization [25]. Further reducing the number of bits in quantization is highly challenging, making lossy compression a promising alternative [16]. However, quantization still holds an advantage in terms of low time overhead. To outperform quantization methods, lossy compression must achieve both high throughput and high compression ratios. These practical challenges continue to drive researchers to push the boundaries of lossy compression techniques.

1.2 Limitations of Existing Approaches

GPU-based lossy compression has developed rapidly over the past decade. These compressors generally achieve significantly higher compression throughput than CPU-based compressors, making them the preferred choice for HPC simulations [21, 55]. However, GPU-based lossy compressors face various challenges. For instance, while cuSZ [50], cuSZx [54], cuSZ-i [40], and MGARD-GPU [36] achieve impressive GPU kernel throughput, they rely on a CPU-GPU hybrid design. This approach requires the CPU to participate in tasks such as global synchronization or building a Huffman tree, which limits their end-to-end throughput. Moreover, cuSZ-i leverages GPU interpolation and Bitcomp from NVIDIA's nvcomp [42] library to improve compression ratios. However, its multi-kernel design limits kernel throughput. Pure GPU designs, such as cuZFP [39], FZ-GPU [57], cuSZp [22], and cuSZp2 [21], avoid these issues but suffer from other limitations, including low throughput or low compression ratios, which hinder their ability to provide significant overall speedups for practical applications. For example, while cuZFP achieves a high compression throughput, its fixed-rate error control scheme restricts the maximum attainable compression ratio. cuSZp2 employs a single-kernel design to significantly increase throughput, but the linear recurrence in its 1D Lorenzo prediction causes a reduction in compression ratio.

1.3 Our Solution: Aatrox

In this work, we propose Aatrox, a single-kernel GPU-based lossy compressor that supports user-customized error control schemes. It further advances the state-of-the-art in GPU-based lossy compression by improving both compression ratio and throughput. We introduce three key optimizations to achieve high compression ratios and high throughput: (1) Hierarchical Data Blocking, (2) Large-Block Delta Encoding, and (3) Dual-Level Delta Decoding.

The main contributions of our work are summarized as follows.

- A novel hierarchical data blocking strategy with three levels (thread, iteration, and warp layer) that solves the extra memory overhead caused by small or large block sizes.
- Large-block delta encoding, which leverages circular shift and tail rotation to solve the inefficient communication problem in the delta encoding process.
- A dual-level delta decoding design that addresses the linear recurrence in delta encoding using a dual-level prefix-sum and leverages tail element accumulation to reduce warp divergence, thereby increasing throughput.
- Evaluation on nine real-world scientific datasets demonstrates that Aatrox achieves a compression and decompression throughput of 388.3 GB/s and 718.0 GB/s on average across the datasets, which is approximately 1.2× faster compared to the best baseline. It also achieves the highest compression ratio among the baselines.

Aatrox will be maintained on GitHub.[1]

2 Understanding Limitations and Challenges in Existing GPU Lossy Compression Designs

In this section, we introduce the background of GPU-based lossy compression and the limitations of state-of-the-art (SOTA) works.

2.1 GPU Lossy Compression

The development of GPU-based compression techniques offers significantly higher compression throughput (exceeding 200 GB/s in state-of-the-art implementations [21]) compared to CPU-based methods, which achieve only 300 MB/s ∼ 1 GB/s in state-of-the-art works [37]. Consequently, an increasing number of lossy compression schemes are being adapted to align with GPU implementations. To leverage the massive parallelism of GPU computing resources, modern GPU-based lossy compressors commonly adopt a data blocking strategy. This approach removes the linear recurrence in the input data stream, enabling each data block to be processed independently. However, this data blocking strategy has certain disadvantages, such as disrupting the data patterns in the original stream, which may lead to a reduction in the compression ratio. A straightforward solution is to

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

increase the data block size; however, the low-latency memory space available for each GPU thread is limited, which consequently restricts the block size.

2.2 Delta Encoding

Delta encoding is a fundamental component in many SOTA lossy compressors. The key idea is to replace the original data with the differences between consecutive data points. By doing so, the entropy of the data is reduced, resulting in a higher compression ratio. For instance, as illustrated in Figure 1 (a), the bit-plane representation of the data block becomes sparse after the delta encoding process, enhancing the data's compressibility. More specifically, each data point is replaced by the difference between its predecessor and its own value. If the predecessor's value perfectly matches the current data point, the resulting value, known as the delta code, is zero. In cases of imperfect matches, the delta codes are values within a small range around zero. This low-entropy distribution of delta codes typically results in a higher compression ratio.

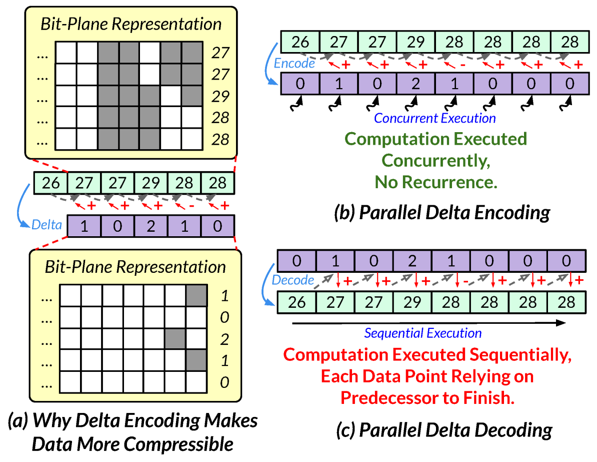

Bit-Plane Representation
27
27
29
28
28
26 27 27 29 28 28
Delta
1 0 2 1 0
Bit-Plane Representation
1
0
2
1
0
(a) Why Delta Encoding Makes Data More Compressible
26 27 27 29 28 28 28 28
Encode
0 1 0 2 1 0 0 0
Concurrent Execution
Computation Executed Concurrently, No Recurrence.
(b) Parallel Delta Encoding
0 1 0 2 1 0 0 0
Decode
26 27 27 29 28 28 28 28
Sequential Execution
Computation Executed Sequentially, Each Data Point Relying on Predecessor to Finish.
(c) Parallel Delta Decoding

Figure 1: An illustration of delta encoding and the challenges in the parallel implementation of encoding and decoding.

2.3 Challenges

Although delta encoding is widely employed in existing GPU solutions, it is not without inherent limitations. ① **Initial Value Overhead**: The delta decoding process relies on the initial value of a data block to reconstruct all other data points within the block. Consequently, this initial value must be stored in the compressed data to facilitate the decoding phase. However, delta encoding implementations in SOTA solutions often use extremely small block sizes to maximize parallelism and, consequently, increase throughput. This approach, however, results in a reduction in compression ratio. The conflicting objectives of achieving a high compression ratio and maximizing throughput make the selection of an optimal block size particularly challenging. ② **Irregular Communication in Delta Encoding**: As illustrated in Figure 1 (b), the basic operation in the delta encoding process does not involve recurrence, allowing it to be executed independently for each data point. In practical implementations, due to the limited number of low-latency registers available in GPU resources, a fixed number of data points is assigned to each thread. Each thread, however, must communicate with previous thread during the delta encoding of its head element, as the predecessor of each head element is stored in the register of the previous thread. This design introduces irregular communication, as the predecessor of thread 0 cannot be accessed in this pattern. The resulting warp divergence in this design degrades throughput. ③ **Linear Recurrences in Delta Decoding**: As illustrated in Figure 1 (c), during the decoding phase, each data point must add the difference value to reconstruct the original data. However, this process is inherently sequential, as each data point depends on the completion of its predecessor. This recurrence leads to sequential execution, which significantly reduces the throughput of the decompression kernel. A plain solution used in SOTA work is to decrease the size of the data block and store the initial value. However, as discussed earlier, this approach negatively impacts the compression ratio. A novel scheme for delta decoding is required.

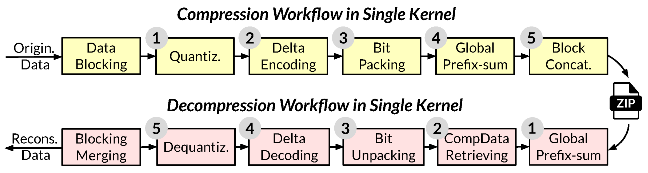


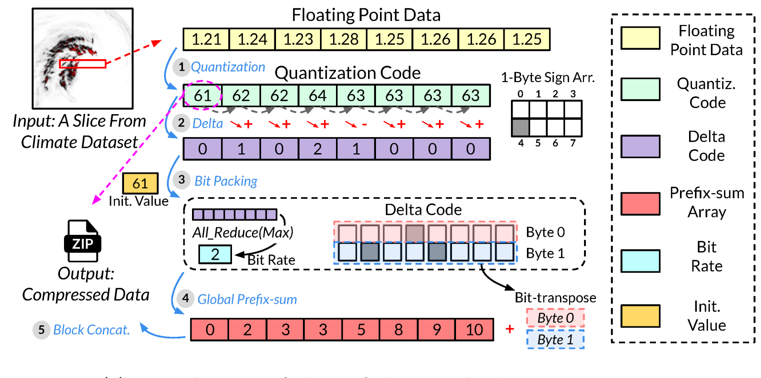
Figure 2: An overview of the workflow, illustrating the main components and their interactions.

3 Aatrox: High-level Workflow

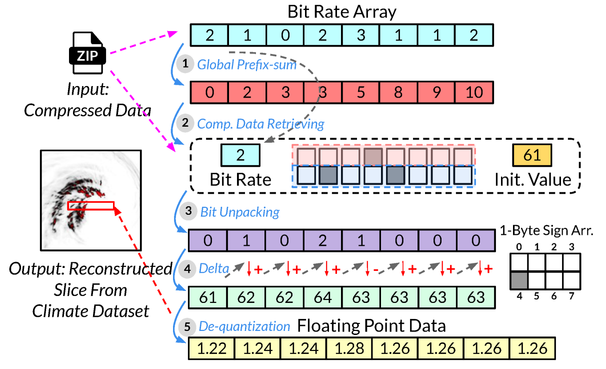
We propose Aatrox, an error-bounded, single-kernel lossy compressor that employs a hierarchy blocking strategy to enhance memory access efficiency and improve the compression ratio. Furthermore, it integrates an optimized delta encoding and decoding mechanism to eliminate inherent linear recurrence. This section provides a high-level overview of the workflow, as shown in Figure 2. We also use a single data block as an example to illustrate the data processing methodology in Aatrox, as depicted in Figure 3.

3.1 Compression Phase

Workflow Overview. We use Figure 2 to illustrate the workflow of Aatrox compression. Given the original input data,



(a) A Running Example for Each Compression Stage



(b) A Running Example for Each Decompression Stage

Figure 3: A practical example demonstrating the compression and decompression process for a single data block.

Aatrox partitions it into multiple levels of data blocks to enable parallelized processing at the warp and thread levels, as detailed in Section 4.1. The data compression process consists of five steps. First, we leverage ① **Quantization** to introduce user-controllable error into the input data, reducing entropy and thereby enhancing compressibility. Second, we apply ② **Delta Encoding**, where the delta codes replace the original quantized codes to exploit spatial redundancy, thereby enhancing compression ratio. Third, we employ ③ **Bit Packing** to determine the minimum number of bits required to represent each value after delta encoding within the data block. Fourth, we calculate the memory offsets in the compressed output for each data block using ④ **Global Prefix-Sum** with a decoupled look-back scheme, ensuring high throughput. Finally, using the offset information and compressed data blocks, we perform the ⑤ **Block Concatenation** step to write back compressed data. This process involves our proposed bit-transpose technique, which transposes the data at the bit level to enable byte-level consecutive memory writes, thereby further improving throughput. Upon completion, the original data is stored as compressed data.

Practical Example. We use a practical example, as shown in Figure 3 (a), to demonstrate the compression process for a single data block and further illustrate our design. For any given input data, it is evenly partitioned into data blocks, which serve as the fundamental units of data processing. The input data consists of floating-point values, and a quantization process is employed to transform these floating points into integers. In this example, we use an error bound of 1e−2. The quantization codes are calculated as $q = \text{round}(x/2eb)$, where q represents the quantization code, x denotes the input floating-point data, and eb represents the error bound. The quantization code is then subtracted from its predecessor to calculate the delta code. The initial quantization code is stored separately within the compressed data for use during the decompression process. A separate bit-sign array is utilized to store the sign of each delta code, while only the absolute values of the delta codes are stored. Additionally, a register is maintained to track the maximum absolute delta code within the data block. This information is used to determine the minimum number of bits required to represent each absolute value in the data block, which is calculated using a ceiling logarithmic transformation $\lceil \log_2(abs(d)) \rceil$, where d denotes for the Delta code. Leveraging this information, only the necessary bits for each integer in the data block are stored to optimize memory usage. We refer to this design as bit packing. If the maximum absolute delta code in a data block is zero, indicating that the block is pure zero, the block is skipped. Since the bit rates are initialized to zero, no additional operations are required. Finally, a global prefix sum is performed to compute the memory offsets for each compressed data block, after which the compressed data is stored in GPU global memory.

3.2 Decompression Phase

Workflow Overview. In the decompression phase, our design is intentionally asymmetric to the compression phase. An overview is provided in Figure 2. To enhance throughput and achieve a higher compression ratio, we only store the number of bits required for each data block (bit rate array) and omit additional information such as the memory offset for each block. Given the compressed data, we first apply ① **Global Prefix-Sum** to bit rate array to calculate the memory offset for each data block. This design leverages the computational power of GPUs to eliminate the need for saving extra information like memory offsets. Using the calculated offsets, we proceed with ② **Compressed Data Retrieval** to extract the packed bits and initial value for current processing data block. Subsequently, we utilize the bit-transpose technique to recover the data. This step, referred to as ③ **Bit Unpacking**, restores the original bit structure. Next, we employ ④ **Delta Decoding** to reconstruct the quantization codes from the delta codes. Finally, we multiply the quantization codes by twice the error bound to reconstruct the floating-point values ($q \times 2\,eb$, where q represents the quantization code and eb represents the error bound), a process referred to as ⑤ **De-quantization**. Upon completion, the compressed data is reconstructed into floating-point values. The difference does not exceed the error bound when compared to the original data.

Practical Example. We also use a practical example, as shown in Figure 3 (b), to demonstrate the decompression design. As mentioned in workflow overview, the memory offset for each data block is not stored explicitly; instead, a bit rate array is saved. During the decompression phase, a global prefix sum is first performed to calculate the memory offsets for each compressed data block. Using these offsets, the corresponding compressed data can be retrieved. Subsequently, a bit unpacking operation is conducted to recover the delta codes based on the bit rate reading from a dedicated location in compressed data, restoring the original integers. The delta decoding operation then reconstructs the quantization codes from delta codes. It is worth noting that the initial value is also required during the delta decoding process, and it is retrieved from a dedicated global memory location, just like the bit rate. Finally, the quantization codes are multiplied by $2eb$, where eb is the error bound, to reconstruct the original floating-point data.

4 Key Optimizations in Aatrox

We present the key components of Aatrox designed to address the challenges in Section 2.3: Hierarchical Data Blocking to address Challenge ①, Large-Block Delta Encoding to address Challenge ②, and Dual-Level Delta Decoding to address Challenge ③.

4.1 Hierarchical Data Blocking

Motivation for Multi-Level Blocking. Data blocking is a common strategy in GPU-based compression works [22, 57]. The primary goal is to eliminate recurrence in the compression workflow, enabling each data block to be processed as an independent input and thus leveraging the massive parallelism offered by GPU computing resources. Traditionally, the data blocking strategy is single-level, as it effectively breaks the recurrence. However, in the design of Aatrox, choosing a large block size leads to a significantly reduced compression ratio in bit packing. This is because the number of bits required to represent each delta code in the data block depends on the maximum absolute value of delta code within the block. Even if the prediction is highly accurate for most data points, other values still require more bits than necessary for storage. Conversely, choosing a very small block size mitigates the issue of excess bits for data points. However, this introduces a new problem: the initial value of each block must be stored separately for delta decoding in the decompression phase (as mentioned in Section 3.2), resulting in an overhead of one value for every data block. This overhead significantly reduces the compression ratio and limits the maximum achievable compression ratio. These conflicting goals make selecting an optimal block size challenging.

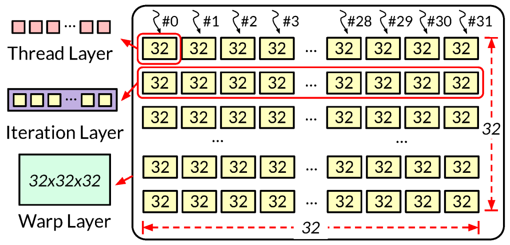


Figure 4: Illustration of hierarchical data blocking.

Hierarchical data blocking. To address this, we propose a hierarchical data blocking scheme that operates at multiple levels of granularity, effectively solving problems for both large data block and small data block simultaneously hence improving overall compression efficiency. More specifically, we propose a three-level data blocking strategy consisting of ① the Warp Layer, ② the Iteration Layer, and ③ the Thread Layer, arranged from top to bottom. Figure 4 illustrates the layer structures. For simplicity, we demonstrate the layers within a single warp (32 threads), as a warp is the minimum execution unit in our design and in the CUDA architecture. All warps share the same data blocking strategy.

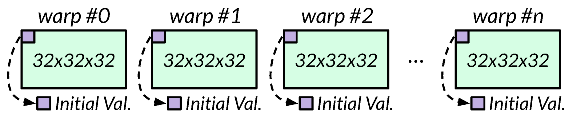


Figure 5: Each warp layer requires a single element overhead for storing initial values.

Warp Layer. First, we partition the input data into warp layer data blocks. As mentioned earlier, it is mandatory to save the initial value for each data block, as the decoding process requires the initial value to compute all subsequent delta codes. To achieve a higher compression ratio, it is critical to minimize the frequency of saving these initial values. To address this challenge, we propose the use of warp-layer data blocks with a large block size ($32 \times 32 \times 32 = 32768$ in our design). Each warp layer is mapped to a single GPU warp, which corresponds to a thread block in our design, for data processing. Delta encoding and decoding are performed at the granularity of the warp layer, requiring only one initial value to be stored for the entire warp layer, as illustrated in Figure 5. This design significantly reduces the memory space required for storing initial values. However, this design introduces a significant challenge: larger block sizes can negatively impact the bit packing compression ratio, potentially offsetting the benefits achieved by reducing the overhead of storing initial values.

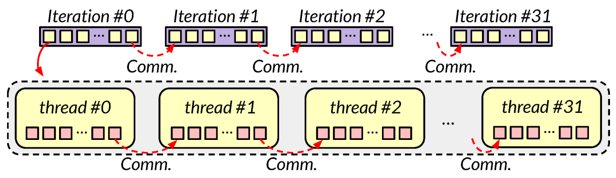


Figure 6: Communication is required for delta encoding of the head elements in each iteration layer and thread layer.

Iteration Layer and Thread Layer. To address the low compression ratio of bit packing caused by the large warp layer, a straightforward solution is to introduce smaller data blocks as a finer-granularity middle layer. However, a single GPU warp consists of only 32 threads. With a very large warp layer, each middle layer, which will be mapped to one thread, is assigned too many elements to process, resulting in long strides between thread memory accesses. This causes the uncoalesced memory access problem in GPU memory operations. Since compression is a memory-bound problem, inefficient memory access leads to a significant throughput drop in the compression kernel. To overcome this challenge, we propose adding two layers beneath the warp layer: the iteration layer and the thread layer. First, the warp layer is evenly partitioned into iteration layers. Then, each iteration layer is further evenly divided into thread layers. The structure of these two layers is illustrated in Figure 6. This design reduces the stride between consecutive thread memory accesses, thereby resolving the uncoalesced memory access issue. Each iteration layer contains $32 \times 32 = 1024$ elements and is mapped to a single iteration in a for loop in GPU kernel. Each thread layer, in turn, contains 32 elements and is mapped to an individual thread within the iteration layer.

4.2 Large-Block Delta Encoding

Communication Overhead. Although the three-level layer design addresses the compression ratio reduction issue, this approach requires additional communication between threads to complete the delta encoding, as shown in Figure 6. This is because each thread cannot access the registers of other threads, preventing each head element from knowing its predecessor (i.e., the tail element of the previous layer), which results in the failure of delta encoding. Straightforward solutions, such as extra global memory access or utilizing GPU shared memory for public access among threads within a warp, are time-consuming and reduce compression throughput. To address this challenge, we propose communication optimizations utilizing warp-level functions and a minimum warp divergence design to enhance compression throughput.

Algorithm 1: Delta Encoding

```
Input: Quantization code $Q$.
Output: Delta code $D$.
1  Initialize $prevQuant, lastTailEle, prevLastTailEle = 0$;
2  Initialize $laneId$ as the index of thread in warp;
3  for $iterationId$ in range(32) do
4      $lastTailEle$ = __shfl_sync ($Q[31]$, $(lane + 32 - 1)\%32$) to
         get the last tail element;                    // Circular shift.
5      if $laneId == 0$ then
6          if $iterationId == 0$ then
7              Save the initial value to global memory;
8          Swap $prevLastTailEle, lastTailEle$;        // Tail rotation.
9      $prevQuant = lastTailEle$;
10     for $i$ in range(32) do
11         $D[i] = Q[i] - prevQuant$;
12         $prevQuant = Q[i]$;                          // Save delta code.
```

Optimized Delta Encoding. We use Algorithm 1 to illustrate the concept underlying the communication optimizations in the Delta Encoding process. Initially, the quantization codes are directly stored in registers instead of shared memory. The warp-level function, such as `__shfl_up_sync()`, is then employed to enable direct communication between threads within a warp. This allows each thread to access the quantization code of the last tail element in the preceding thread layer to complete the delta encoding. However, this design only addresses the delta encoding process within the thread layer. For communication between iteration layers, an additional exchange is required between thread 0 in iteration i and thread 31 in iteration $i - 1$. This irregular communication degrades overall performance. To address this issue, we propose two techniques, named: ① Circular Shift and ② Tail Rotation.

Circular Shift. We first detail the design of ① Circular Shift optimization, as illustrated in Figure 7. We propose that, instead of using `__shfl_up_sync()`, which leaves the register in thread 0 unchanged, we employ `__shfl_sync()` to make the shuffle up operation wrap around the warp. This ensures that thread 0 receives the tail element from thread 31, and saves it temporarily. In the subsequent iteration, thread 0 can use this value, stored in a buffer referred to as *lastTailEle*, to compute the delta code. The corresponding implementation is shown on Line 4 in Algorithm 1.

Tail Rotation. However, the plain implementation of this Circular Shift operation introduces numerous if-else branches into the GPU kernel, leading to severe warp divergence and reduced throughput. For instance, a straightforward solution requires a separate if-branch for thread 0

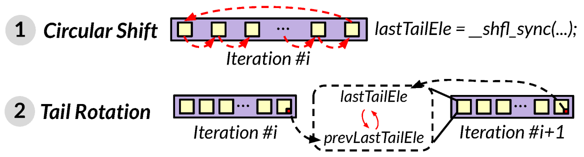


Figure 7: Illustration of two optimization techniques: circular shift and tail rotation.

to compute the delta code differently from other threads in the warp. To address this inefficiency, we propose the second optimization, 2 Tail Rotation, which minimizes warp divergence to optimize kernel performance, as illustrated in Figure 7. Specifically, we maintain a buffer named *prevLastTailEle* initialized as 0. Recall the buffer named *lastTailEle*, which is used in Circular Shift for warp-level shuffling to retrieve the wrap-around last tail element from the previous thread layer. Subsequently, thread 0 swaps the *lastTailEle* buffer with the *prevLastTailEle* buffer. This mechanism makes the tail element of the current iteration layer to be recorded in *prevLastTailEle*, while the tail element from the previous iteration layer is written to *lastTailEle*. This process is implemented in Line 8 of Algorithm 1. By employing this approach, the kernel requires only a single if-branch, thereby minimizing warp divergence and enhancing overall performance.

4.3 Dual-Level Delta Decoding

Asymmetric Design in Delta Decoding. The delta decoding process is inherently more complex compared to encoding. During encoding, the original values remain visible, enabling each thread to compute the delta encoding independently. In contrast, decoding depends on the completion of the delta decoding process for predecessor element. To reconstruct the quantization codes from the delta codes, the warp layer must be processed sequentially. To address this linear recurrence, we propose an asymmetric design for delta decoding. Specifically, we introduce two key optimizations: 1 Two-level Prefix-sum and 2 Tail Element Accumulation. The following sections provide a detailed explanation of these optimizations.

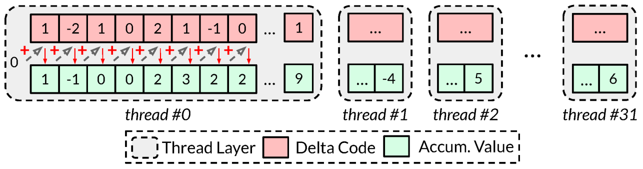


Figure 8: Illustration of thread-level prefix-sum.

Dual-Level Prefix-Sum. To address the linear recurrence of delta decoding, we propose the use of a dual-level prefix-sum approach to reconstruct the quantization codes hierarchically, aligning with our hierarchical data blocking strategy. This method enables parallel execution. Specifically, instead of directly reconstructing the quantization codes from the delta codes, we utilize prefix-sum to calculate the accumulated values for the entire warp based on the delta codes. These accumulated values are then combined with the initial values stored during the compression phase to reconstruct the quantization codes. First, we perform prefix-sum to calculate the accumulated values at the thread layer, as illustrated in Figure 8. In each thread layer, since the delta codes are stored in registers, the accumulation can be performed sequentially, a process we term as the thread prefix-sum. Subsequently, we extract the accumulated tail values from each thread layer to perform another prefix-sum, referred to as the warp prefix-sum, as shown in Figure 9. At the warp level, instead of relying on shared memory, which incurs high overhead, we employ `__shfl_up_sync` to enhance inter-thread communication. Specifically, values are passed to threads with stride ranges from 1 to 16, and these passed values are summed to compute the inclusive prefix-sum. The exclusive prefix-sum is obtained by subtracting the accumulated values. At this stage, the accumulated values for all iterations at the iteration layer are determined. However, recovering the quantization codes still requires the initial value for each iteration layer. While the initial value for the first iteration is saved, obtaining the initial values for subsequent iterations remains a challenge.

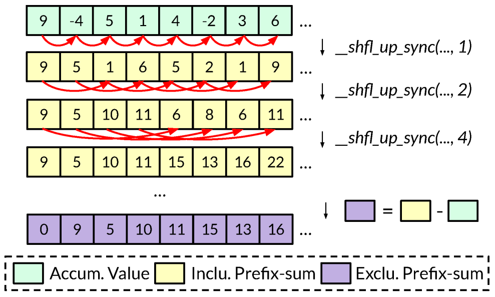


Figure 9: Illustration of warp-level prefix-sum.

Tail Element Accumulation. To address the issue of missing initial values, a straightforward solution is to introduce a third-level prefix-sum across the iteration layers. However, this approach requires completing all iteration layers, leading to sequential operations within the GPU kernel, which degrades overall throughput. Additionally, it incurs extra computational overhead. To address this issue efficiently, we propose a design termed Tail Element Accumulation, which dynamically calculates the initial value for each iteration layer without significant overhead. Specifically, we maintain a buffer for each thread in the warp, referred to as *accumTail*. Initially, this buffer is set to the initial value of the warp layer. This value is then used to recover the quantization codes for the first iteration layer. Subsequently, we employ a warp broadcast operation to broadcast the last element of the iteration layer to *accumTail*. This updated value is then utilized to recover the quantization codes for the next iteration layer, and the process continues iteratively.

Combining these two optimization designs, we demonstrate the whole asymmetric delta decoding process in Algorithm 2. For the two-level prefix-sum, thread prefix-sum is in Line 6-8, the warp prefix-sum is from Line 9-14. And the accumulated tail element is demonstrated in Line 19.

Algorithm 2: Delta Decoding

```
   Input: Delta code $D$, $initialValue$
   Output: Quantization code $Q$.
1  Initialize $tailEle, accumTail, tmpBuffer = 0$;
2  Initialize $laneId$ as the index of thread in warp;
3  Initialize $warpId$ as the index of warp(thread block) in grid;
4  for $iterationId$ in range(32) do
5      $tmpBuffer = 0$;
6      for $i$ in range(32) do
7          $tmpBuffer += D[i]$;                    // Thread prefix-sum.
8          $Q[i] = tmpBuffer$;
9      $tailEle = Q[31]$;                          // Warp prefix-sum.
10     for $i$ in 1,2,4,8,16 do
11         $tmpBuffer =$ __shfl_up_sync($tailEle, i$);
12         if $laneId >= i$ then
13             $tailEle += tmpBuffer$;
14     $tailEle -= Q[31]$;                         // Exclusive prefix-sum.
15     if $iterationId == 0$ then
16         $accumTail = initialValue[warpId]$;     // Read init.
17     for $i$ in range(32) do
18         $Q[i] += tailEle + accumTail$;
19     $accumTail =$ __shfl_sync($Q[31], 31$);     // Tail ele. accum.
```

5 Evaluation

In this section, we evaluate Aatrox against four SOTA GPU-based lossy compressors across various metrics, including compression ratio and throughput, using nine real-world scientific datasets. Our results demonstrate that Aatrox outperforms the baselines in each individual metric and, overall, delivers significant end-to-end acceleration when applied to real-world applications.

5.1 Experimental Setup

Platforms. We evaluate our approach on two platforms: ① One node from an HPC cluster equipped with two 64-core AMD EPYC 7742 CPUs operating at 2.25GHz and four NVIDIA Ampere A100 GPUs (108 SMs, 40GB), running CentOS 7.4 and CUDA 11.4.120. ② An in-house workstation equipped with one 28-core Intel Xeon Gold 6238R CPUs operating at 2.20GHz and two NVIDIA GTX A4000 GPUs (40 SMs, 16GB), running Ubuntu 20.04.5 and CUDA 11.7.99. While we use a single GPU for evaluation, multi-GPU processing is considered embarrassingly parallel with respect to single-GPU processing. This is because we partition data in a coarse-grained manner to fit into a single GPU, with each data chunk being independent of the others. Due to this lack of data dependency, multi-GPU comparisons would involve only varying numbers of data chunks.

Table 1: Real-world datasets used in the evaluation.

DATASETS	FIELD SIZE dimensions	DATASET SIZE #fields
Climate simulation	673.9 MB	20.71 GB
CESM-ATM [8]	3600×1800×26	33 in total
Cosmology: particle simulation	4.3 GB	23.99 GB
HACC [19]	1,073,726,487	6 in total
petroleum exploration	1.4 GB	3.99 GB
RTM [5, 23]	1008×1008×352	3 in total
Climate simulation	564.5 MB	6.31 GB
SCALE [38]	1200×1200×98	12 in total
Quantum Monte Carlo	630.7 MB	1.17 GB
QMCPack [46]	69×69×33120	2 in total
cosmology simulation	536.9 MB	3 GB
NYX [43]	512×512×512	6 in total
numerical simulation	6.7 GB	6.23 GB
JetIn [18]	1408×1080×1100	1 in total
Rayleigh-Taylor instability	4.3 GB	4.00 GB
Miranda [9]	1024×1024×1024	1 in total
octet truss	6.9 GB	6.42 GB
SynTruss [29]	1200×1200×1200	1 in total

Datasets. Our evaluation and comparative analysis are conducted on nine distinct datasets derived from real-world compression tasks. These datasets encompass a wide range of domains, showcasing the adaptability and versatility of our system. The datasets are sourced from the Scientific Data Reduction Benchmarks (SDRBench) [59] and the Open Scientific Visualization Datasets (Open-SciVis) [28]. Detailed descriptions and characteristics of these datasets are systematically presented in Table 1.

Baselines. We compare Aatrox with four state-of-the-art GPU-based lossy compressors: FZ-GPU [57], cuSZp [22], cuSZp2 [21], and cuZFP [39]. We evaluate performance using three typical relative error bounds (relative to the value range of the data field): 1e−2, 1e−3, and 1e−4. Note that cuZFP does not support the error-bound mode; it only supports the fixed-rate mode.

5.2 Evaluation Metrics

Our evaluation metrics include ① compression ratio, ② compression throughput, and ③ data quality, which are detailed as follows.

Compression Ratio. The compression ratio is one of the most commonly used metrics in compression research. It is defined as the ratio of the original data size to the compressed data size. A higher compression ratio indicates more efficient information aggregation relative to the original data.

Compression Throughput. Compression throughput refers to the amount of data a compressor can process within a unit of time. It is a key advantage of using GPU-based lossy compressors over CPU-based ones. Notably, we evaluate the end-to-end throughput for both the compression and decompression processes, which includes memory allocation, memory transfers (host-to-device and device-to-host), memory deallocation, and other related operations. This approach provides a more practical measure of throughput, accurately reflecting the performance of compression in real-world scientific applications.

Data Quality. Data quality evaluation is critical for assessing the performance of lossy compression in terms of data reconstruction accuracy. In this work, we primarily use the Peak Signal-to-Noise Ratio (PSNR) to quantify distortion. Additionally, we employ the Structural Similarity Index Measure (SSIM) to evaluate the quality of the reconstructed data. SSIM is a widely used metric for measuring the similarity between two images. The calculations of PSNR and SSIM are detailed in [20]. Moreover, we include visualizations of the reconstructed data alongside the original data to illustrate the impact of errors introduced by lossy compression.

5.3 Evaluation of Compression Ratio

First, we evaluate the compression ratio of four compressors in error-bounded mode: AATROX, cuSZp2, FZ-GPU, and cuSZp, with range-based error bounds of 1e−2, 1e−3, and 1e−4. We exclude cuZFP because it does not support the error-bounded mode. Its fixed-rate mode locks the compression ratio, making it unnecessary for this evaluation. The results are presented in Table 2. We calculate the average compression ratio for each dataset by averaging the compression ratios across all fields. Additionally, we report the minimum and maximum compression ratios for each dataset and use a bar to visually represent the average compression ratio. For FZ-GPU, missing bars indicate that it failed to execute on some datasets due to bugs in launching the 3D-Lorenzo predictor.

According to the results, *AATROX achieves the highest compression ratio in nearly all datasets across all error bounds, outperforming the baselines in 23 out of 27 cases.* In the three cases where AATROX does not outperform others, two occur in the JetIn dataset, where AATROX achieves compression ratios only 0.02% and 0.04% lower than cuSZp2, differences that are negligible in real-world applications. **Compared to cuSZp2**, AATROX provides 1.37× and 2.33× higher compression ratios on the CESM-ATM and HACC datasets, respectively, with an error bound of 1e−2. This improvement is attributed to the large block design, which reduces overhead by avoiding the need to store initial values. **Compared to FZ-GPU**, AATROX achieves approximately 1.5× and 2× higher compression ratios on the CESM-ATM and RTM datasets, respectively. **Compared to cuSZp**, AATROX achieves nearly 3× higher compression ratio on the Miranda dataset with an error bound of 1e−2.

Highlight I: AATROX achieves the highest compression ratio compared to other baselines in 23 out of 27 cases. The evaluation demonstrates that AATROX delivers up to 2.33× higher compression ratio at the same error bound compared to the best baseline, cuSZp2.

5.4 Evaluation of Throughput

Next, we evaluate the throughput of five compressors. The results are presented in Figure 10 and Figure 11 for compression and decompression, respectively. Regarding the missing bars for FZ-GPU, it fails to execute on some datasets due to bugs in 3D-Lorenzo launching. It is worth noting that in this evaluation, we measure end-to-end throughput instead of GPU kernel throughput. This is because end-to-end throughput is more practical and closely reflects the performance in real-world scientific applications. We vary the range-based relative error bounds to 1e−2, 1e−3, and 1e−4. Additionally, we compute the average compression and decompression throughput across all datasets to provide a more general assessment of the compressors. Note that we use fixed bit-rate at 4, 8, and 16 for cuZFP.

The results demonstrate that *AATROX achieves the highest compression throughput compared to all baselines and the highest decompression throughput in 23 out of 27 cases.* On average, AATROX achieves 388.3 GB/s compression throughput and 718.0 GB/s decompression throughput, the highest average throughput among all evaluated compressors.

Compared to cuSZp2, AATROX achieves a 1.2× speedup in compression throughput and a 1.6× speedup in decompression throughput. This performance gain is attributed to cuSZp2's strategy of handling outliers for each small data block separately, which reduces throughput. Another important observation is that compressors with higher compression ratios also tend to benefit throughput. This is because compressed data must be written to global memory, which can become a bottleneck in the workflow. A higher compression ratio reduces the amount of data written, thereby increasing compression throughput.

Compared to cuZFP, FZ-GPU, and cuSZp, AATROX achieves 3.6×, 2.3×, and 2.7× compression speedups, respectively. It is worth noting that the throughput of AATROX is exceptionally high in certain cases. This is because inspired by cuSZp2 [21], we adopt a skipping design for pure-zero blocks. This strategy significantly enhances the performance

Table 2: Compression ratios of four GPU-based error-bounded lossy compressors. The minimum and maximum compression ratios for each dataset are noted. The bar at the bottom of each data point represents the average compression ratio.

Compressor	REL	CESM-ATM	HACC	RTM	SCALE	QMCPack	NYX	JetIn	Miranda	SynTruss
Aatrox	1E-2	**26.33~95.12 (avg: 59.22)**	**12.24~69.28 (avg: 36.01)**	30.25~103.96 (avg: 61.41)	**23.61~109.34 (avg: 54.29)**	**12.09~24.13 (avg: 18.11)**	**15.83~127.30 (avg: 71.13)**	125.76~125.76 (avg: 125.76)	**12.03~12.03 (avg: 12.03)**	**12.99~12.99 (avg: 12.99)**
	1E-3	**16.09~62.59 (avg: 29.66)**	**5.93~16.26 (avg: 10.41)**	**12.18~85.03 (avg: 40.39)**	**12.64~80.46 (avg: 31.84)**	**5.91~13.93 (avg: 9.92)**	**11.27~125.01 (avg: 43.10)**	119.57~119.57 (avg: 119.57)	**6.46~6.46 (avg: 6.46)**	**6.49~6.49 (avg: 6.49)**
	1E-4	**8.82~41.15 (avg: 16.96)**	**3.72~6.92 (avg: 5.16)**	**6.60~67.91 (avg: 29.46)**	**6.87~51.39 (avg: 18.81)**	**3.73~7.49 (avg: 5.61)**	**5.68~97.56 (avg: 24.28)**	106.15~106.15 (avg: 106.15)	**3.99~3.99 (avg: 3.99)**	**4.26~4.26 (avg: 4.26)**
cuSZp2	1E-2	18.44~82.41 (avg: 42.98)	11.49~20.09 (avg: 15.50)	**30.12~104.18 (avg: 61.48)**	16.80~109.55 (avg: 46.19)	12.44~23.57 (avg: 18.01)	14.36~127.80 (avg: 69.14)	**126.28~126.28 (avg: 126.28)**	11.10~11.10 (avg: 11.10)	12.96~12.96 (avg: 12.96)
	1E-3	12.99~57.45 (avg: 24.53)	5.85~12.47 (avg: 8.82)	12.00~84.96 (avg: 40.24)	11.10~79.69 (avg: 29.52)	6.07~13.29 (avg: 9.68)	10.50~125.56 (avg: 41.75)	**120.04~120.06 (avg: 120.06)**	5.98~5.98 (avg: 5.98)	6.47~6.47 (avg: 6.47)
	1E-4	7.85~39.01 (avg: 14.97)	3.67~6.27 (avg: 4.84)	6.51~67.81 (avg: 29.36)	6.31~49.95 (avg: 17.92)	3.79~7.25 (avg: 5.52)	5.43~98.37 (avg: 24.12)	**106.50~106.50 (avg: 106.50)**	3.80~3.80 (avg: 3.80)	4.25~4.25 (avg: 4.25)
FZ-GPU	1E-2	17.62~100.02 (avg: 40.52)	N.A. (due to bugs)	12.25~70.09 (avg: 34.60)	16.39~124.25 (avg: 45.21)	7.53~19.04 (avg: 13.28)	13.38~222.62 (avg: 86.15)	N.A. (due to bugs)	N.A. (due to bugs)	N.A. (due to bugs)
	1E-3	12.03~58.03 (avg: 21.57)	N.A. (due to bugs)	6.37~43.76 (avg: 20.42)	10.89~69.61 (avg: 25.39)	4.33~12.08 (avg: 8.20)	9.81~183.98 (avg: 42.34)	N.A. (due to bugs)	N.A. (due to bugs)	N.A. (due to bugs)
	1E-4	7.10~36.03 (avg: 12.98)	N.A. (due to bugs)	4.02~30.70 (avg: 13.92)	7.26~39.22 (avg: 16.16)	2.99~8.26 (avg: 5.62)	5.98~59.98 (avg: 16.15)	N.A. (due to bugs)	N.A. (due to bugs)	N.A. (due to bugs)
cuSZp	1E-2	3.88~69.43 (avg: 32.56)	5.28~10.6 (avg: 7.92)	29.08~102.73 (avg: 60.10)	3.88~105.89 (avg: 37.76)	12.44~22.21 (avg: 17.33)	9.6~127.8 (avg: 66.73)	126.27~126.27 (avg: 126.27)	4.46~4.46 (avg: 4.46)	12.67~12.67 (avg: 12.67)
	1E-3	2.78~39.01 (avg: 14.53)	3.45~5.37 (avg: 4.41)	11.06~81.90 (avg: 38.43)	2.75~72.60 (avg: 21.11)	6.08~10.08 (avg: 8.08)	5.09~125.55 (avg: 38.44)	119.86~119.86 (avg: 119.86)	3.04~3.04 (avg: 3.04)	6.37~6.37 (avg: 6.37)
	1E-4	2.11~24.55 (avg: 8.26)	2.53~3.47 (avg: 3.00)	6.07~65.04 (avg: 28.04)	2.14~42.06 (avg: 12.34)	3.79~5.56 (avg: 4.68)	3.35~98.23 (avg: 22.14)	105.59~105.59 (avg: 105.59)	2.32~2.32 (avg: 2.32)	4.21~4.21 (avg: 4.21)

of Aatrox on sparse datasets. For instance, the decompression throughput of Aatrox exceeds 1,200 GB/s on the JetIn dataset with an error bound of 1e−2.

Highlight II: In terms of throughput, Aatrox achieves 388.3 GB/s compression throughput and 718.0 GB/s decompression throughput, representing a 1.2× and 1.6× speedup in compression throughput and decompression throughput compared to cuSZp2, the best-performing baseline.

5.5 Evaluation of Data Quality

In this section, we evaluate the quality of the reconstructed data. First, we visualize a slice in the field from the CESM-ATM dataset, a climate simulation dataset suitable for visualization. The results are shown in Figure 12. From left to right, the visualizations depict the original data, Aatrox, and cuZFP. We exclude the results of cuSZp2, FZ-GPU, and cuSZp because they share the same quantization strategy, which is the only lossy component in their workflows. Thus, their visualizations are identical but with lower compression ratios, as evaluated in Section 5.3. For this field, Aatrox achieves a compression ratio of 16.4. To provide a fair comparison, we set the bitrate of cuZFP to 2 to achieve a similar compression ratio. It is evident that Aatrox produces a visualization very similar to the original data, while the reconstructed visualization from cuZFP alters the value range. This is apparent in Figure 12, where the cuZFP result exhibits a different color distribution compared to the original image. Moreover, cuZFP introduces noticeable artifacts, as shown in the 100×100 cropped and zoomed-in image. Overall, Aatrox achieves better visualization quality, as confirmed by higher

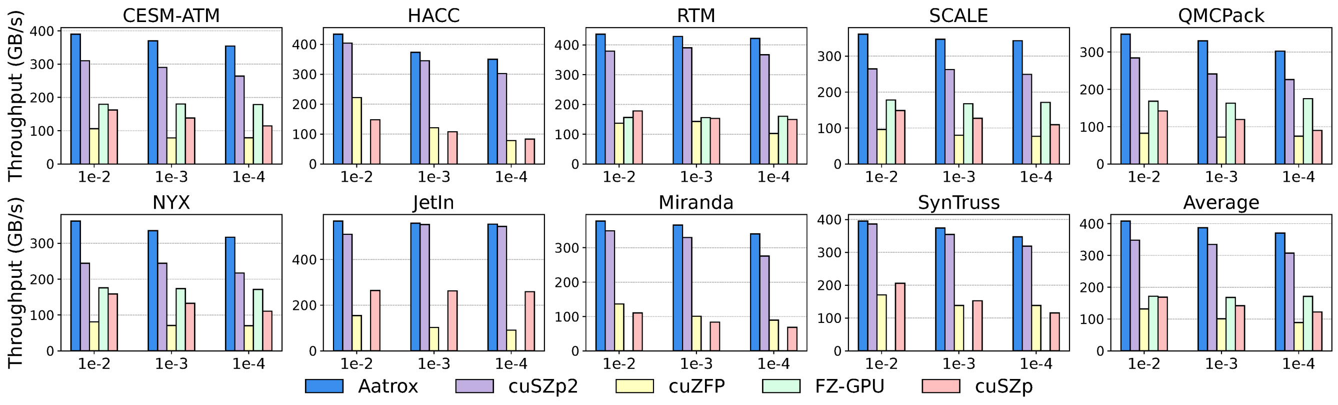


Figure 10: Compression throughput evaluation on A100 GPU.

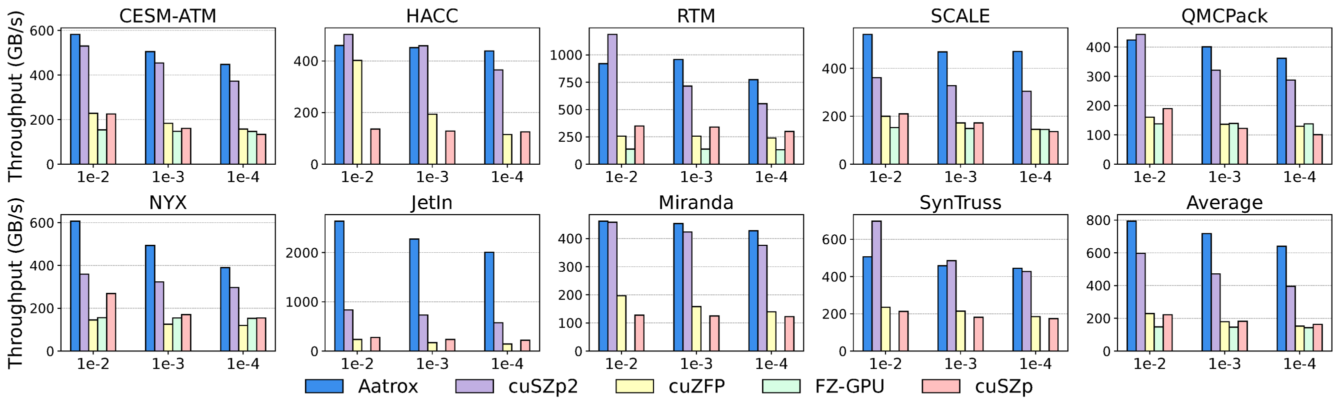


Figure 11: Decompression throughput evaluation on A100 GPU.

PSNR and SSIM values (higher is better). Similar observations can be made across other datasets.

Another way to evaluate data quality is through a rate-distortion map, where the x-axis represents the bitrate (inverse of compression ratio) and the y-axis represents the PSNR or SSIM values. As previously mentioned, Aatrox shares the same quantization strategy as cuSZp2, FZ-GPU, and cuSZp, and thus achieves identical PSNR and SSIM values at the same error bounds. For cuZFP, prior work such as FZ-GPU [57] has demonstrated that FZ-GPU achieves a better rate-distortion curve than cuZFP. Combined with the compression ratio evaluation in Section 5.3, where Aatrox achieves the best compression ratio in almost all cases, this illustrates that the rate-distortion performance of Aatrox outperforms the baselines.

Highlight III: Aatrox effectively preserves data quality, as demonstrated by visualizations and metrics such as PSNR and SSIM. Furthermore, Aatrox achieves the best rate-distortion curve compared to baselines.

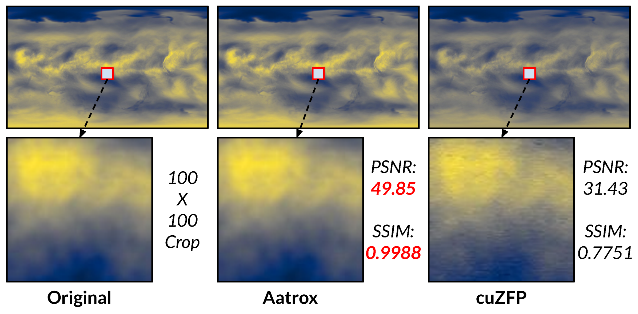


Figure 12: Visualization of slice 13 (the middle slice) from `RELHUM_1_26_1800_3600` field in CESM-ATM dataset.

5.6 Use Case: Reducing Data Transfer Overhead

Besides compression throughput (T_{compr}), we propose an additional metric called overall speedup to evaluate the performance of compressors. The overall speedup considers the improvement a compressor provides for end-to-end applications, making it more practical for real-world scientific scenarios. Specifically, the overall speedup is calculated as:

$$\text{speedup} = \frac{1}{\left((BW \times \text{CR})^{-1} + T_{\text{compr}}^{-1}\right) \times BW}$$

where BW represents the memory bandwidth through which compressed data is transferred, and CR denotes the compression ratio.

We use GPU-to-CPU data transfer as a use case for Aatrox. Our HPC cluster node is equipped with 4 A100 GPUs connected to the CPU via a 32-lane PCIe 4.0 interconnect; each GPU can utilize up to a 16-lane bandwidth (i.e., 32 GB/s). Based on our benchmarking results using [1], when all 4 GPUs simultaneously read/write data to/from the CPU, the bandwidth per GPU can drop to as low as 11.4 GB/s (aggregately about 45 GB/s). We measured the overall data-transfer speedup of various compressors, as shown in Figure 13. The results illustrate that Aatrox achieves the best overall speedup across all datasets and at all evaluated relative error bounds, outperforming the second-best compressor, cuSZp2, by up to 70%.

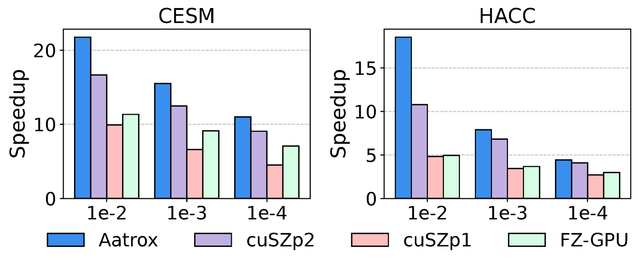


Figure 13: Speedup of applying different compressors for GPU-to-CPU data transfer operations.

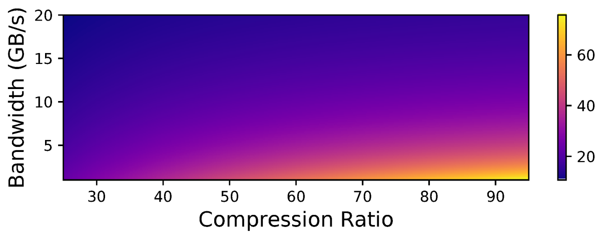


Figure 14: Speedup of applying Aatrox under different compression ratios and bandwidths.

Using the CESM-ATM dataset as an example, the compression ratio of Aatrox ranges from 26.3 to 95.2, with an average compression throughput of 372.6 GB/s. Using this information, we evaluated the speedup of Aatrox under different transfer scenarios with varying bandwidths, as shown in Figure 14. The results indicate that Aatrox efficiently accelerates data transfer, particularly in low-bandwidth scenarios. Our approach achieves a speedup of over 70×, with the lowest speedup under the given settings being 10×. This demonstrates the in-situ characteristics of Aatrox.

6 Discussion

In this section, we provide additional evaluations that supplement the main results and offer deeper insights.

6.1 Breakdown Performance Analysis

We evaluate the breakdown performance of different components in our design. Due to the single-kernel design, we evaluate each component by disabling other operations in the kernel that may affect throughput. The results are averaged over the datasets.

Compression Phase. The results show that quantization (including reading data from global memory) accounts for 10.2% of the compression kernel time, while delta encoding accounts for 12.3%. This demonstrates that the optimized design of delta encoding performs well and does not impose significant overhead on the compression workflow. Bit-packing takes negligible time (less than 1%) due to its embarrassingly parallel nature. The global prefix-sum accounts for 32.8% of the time. Despite utilizing the decoupled lookback strategy [41] for high-performance design, the device-level operation remains time-consuming, resulting in large overhead. Finally, block concatenation accounts for 41.5% of the total time. The irregular lengths of compressed data blocks cause non-coalesced memory access patterns, leading to a reduction in the process of writing to global memory, which becomes a bottleneck.

Decompression Phase. The global prefix-sum in decompression is identical to that in the compression phase. However, due to the higher decompression throughput, it accounts for 39.8% of the decompression kernel time, which is longer compared to 32.8% in the compression phase. The compressed data retrieval process takes 10.1% (including reading data from global memory), as the size of the compressed data is relatively small. Bit-unpacking remains negligible (less than 1%) due to its inherently parallel nature, similar to compression. Delta decoding accounts for 16.5% of the decompression kernel time, primarily due to the additional shuffled functions required. Finally, the dequantization process takes 54.6% of the decompression kernel time because of the writing operations to global memory. For sparse datasets, this process is much faster, as most data points are zero and can be skipped. However, for non-sparse datasets, dequantization becomes the primary bottleneck.

Compared to the data copy time between the CPU and GPU, the maximum bandwidth of the 16-lane PCIe 4.0 interconnect used in our evaluation platform is 32 GB/s. In contrast, the average compression throughput of AATROX is 388 GB/s—over 10× faster than the data movement speed.

6.2 Compatibility with Lower-End GPUs

In addition to evaluating performance on the NVIDIA A100 GPU, we also assess the throughput of AATROX on a lower-end GPU, the NVIDIA A4000 (16 GB), in platform 2. The results are presented in Figure 15. We use the QMCPack and NYX datasets, and similar observations can be made across other datasets. AATROX achieves an average of 120.7 GB/s compression throughput and 344.3 GB/s decompression throughput. This demonstrates that our optimizations maintain high performance across different GPU platforms.

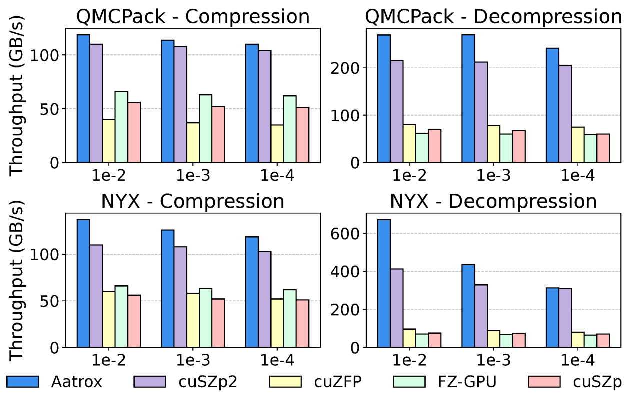


Figure 15: Throughput of AATROX on NVIDIA A4000 GPU for the QMCPack and NYX datasets.

6.3 Compression Ratio Gain by Hierarchical Delta Approach

We also evaluate the compression ratio gain achieved by our optimized hierarchical delta encoding design. We implement a prototype that saves the initial value for each thread layer (small data block), referred to as the plain approach. The gain is calculated as the compression ratio of our hierarchical delta approach divided by that of the plain solution. The results are presented in Table 3.

We observe that our approach achieves an average compression ratio improvement of 1.74×, 1.50×, and 1.39× for error bounds of 1e−2, 1e−3, and 1e−4, respectively, averaged across all datasets. It is worth noting that our approach achieves better compression ratios when the data is very smooth, as exemplified by the HACC dataset. However, when the data is sparse (with most values being zero), both the plain solution and the hierarchical design achieve high compression ratios, as observed in the JetIn dataset. Moreover, with optimized designs such as Circular Shift and Tail Rotation, our method incurs only around 10% overhead compared to the plain solution.

Table 3: Compression ratio gain by Hierarchical Delta.

	CESM	HACC	RTM	SCALE	QMCPack
1e-2	1.82×	4.56×	1.03×	1.44×	1.05×
1e-3	2.05×	2.37×	1.05×	1.51×	1.24×
1e-4	2.07×	1.78×	1.05×	1.53×	1.22×

	NYX	Jetih	Miranda	SynTruss	Average
1e-2	1.07×	1.00×	2.67×	1.03×	1.74×
1e-3	1.13×	1.00×	2.15×	1.03×	1.50×
1e-4	1.10×	1.01×	1.73×	1.01×	1.39×

6.4 Hybrid Compressors

Several GPU-based lossy compressors adopt a CPU-GPU hybrid design, leveraging GPUs as accelerators while delegating sequential processes to CPUs. For example: cuSZ [50] utilizes Lorenzo prediction and Huffman lossless encoding to provide error-bounded lossy compression. Yu *et al.* proposed cuSZx [54], an extension of the cuSZ framework, which achieves high compression throughput through lightweight bitwise operations. Chen *et al.* developed MGARD-GPU [7], which optimizes data refactoring kernels for GPUs, enabling efficient manipulation of data in multigrid-based hierarchical forms. Although these compressors achieve impressive kernel throughput, their end-to-end performance is often constrained by the overhead of CPU-GPU data movement.

7 Related Work

Compression has become popular in the development of data reduction techniques [6, 12]. While some lossless compressors provide high compression throughput [30, 42, 58], they still face challenges in achieving high compression ratios. In contrast, lossy compression introduces user-customized error to significantly improve the compression ratio, thereby benefiting HPC systems [2, 11, 26, 32, 37, 39, 49, 53]. Among these, two notable lossy compressors are SZ [11, 34, 35, 49] and ZFP [39]. SZ is a prediction-based compressor composed of data prediction, linear-scale quantization, variable-length encoding, and dictionary encoding. On the other hand, ZFP is a transform-based compressor that operates at block granularity, employing alignment, orthogonal transforms, and embedded encoding.

As the development of GPU-accelerated applications, lossy compressors have adapted to GPU-based implementations over the past decade [14, 15, 21, 22, 36, 39, 40, 44, 50, 52, 54, 56]. Lindstrom [39] developed cuZFP, which adapts the fixed-rate design of ZFP to GPUs, preserving high throughput and ensuring high visualization quality. Tian et al. [50] introduced cuSZ, the first prediction-based GPU error-bounded lossy compressor. Zhang et al. [57] proposed a pure-GPU implementation named FZ-GPU and introduced a novel lossless encoding method that achieves significant overall speedup. Huang et al. [21, 22] innovatively designed a single-kernel compression approach named cuSZp and further improved it to cuSZp2 with an optimized prefix-sum and a novel outlier fixed-length encoding method, significantly increasing throughput. Aatrox advances compressor efficiency by enhancing both the compression ratio and throughput compared to SOTA methods. Lossy compressors have also been adapted for diverse use cases on various platforms, such as Cerebras [47, 48] and Data Processing Units (DPUs) [33].

8 Conclusion and Future Work

In this paper, we develop a single-kernel error-bounded lossy compressor for scientific data on GPUs. Specifically, we propose Aatrox, a high-throughput and high-compression-ratio compressor that utilizes hierarchical data blocking and large-block delta encoding/decoding. We evaluate our proposed Aatrox on nine representative scientific datasets, demonstrating its high compression throughput and ratio. It achieves an average throughput of 388.3 GB/s for compression and 718.0 GB/s for decompression on an NVIDIA A100 GPU. Compared to state-of-the-art compressors such as cuSZp2, cuSZp, cuZFP, and FZ-GPU, Aatrox achieves approximately 1.2× speedup while delivering the highest compression ratios. In the future, we plan to adapt Aatrox to other GPU platforms by leveraging code translation tools such as HIPFY [3] for AMD GPUs and SYCLomatic [24] for Intel GPUs. The impact of parameters (e.g., layer size) on the compression ratio and throughput in Aatrox varies across datasets. We also plan to explore fine-grained parameter tuning in future work.

Acknowledgments

This research is partially supported by the National Science Foundation under Grant Nos. 2104023, 2311875, 2311876, 2326495, 2247060, 2247080, and 2211538, and the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contracts DE-SC0024559 and DE-AC02-06CH11357.

References

[1] [n. d.]. Benchmark of measuring bandwidth of multiple GPU. https://github.com/enfiskutensykkel/multi-gpu-bwtest.

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

[3] AMD. [n. d.]. HIPIFY. https://github.com/ROCm-Developer-Tools/HIPIFY.

[4] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, et al. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

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

[6] Franck Cappello, Mario Acosta, Emmanuel Agullo, Hartwig Anzt, Jon Calhoun, Sheng Di, Luc Giraud, Thomas Grützmacher, Sian Jin, Kentaro Sano, et al. 2025. Multifacets of lossy compression for scientific data in the Joint-Laboratory of Extreme Scale Computing. *Future Generation Computer Systems* 163 (2025), 107323.

[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] Community Earth System Model (CESM) Atmosphere Model. 2019. http://www.cesm.ucar.edu/models/. Online.

[9] Andrew W Cook, William Cabot, and Paul L Miller. 2004. The mixing transition in Rayleigh–Taylor instability. *Journal of Fluid Mechanics* 511 (2004), 333–362.

[10] Luis Rangel DaCosta, Hamish G Brown, Philipp M Pelz, Alexander Rakowski, Natolya Barber, Peter O'Donovan, Patrick McBean, Lewys Jones, Jim Ciston, MC Scott, et al. 2021. Prismatic 2.0–Simulation software for scanning and high resolution transmission electron microscopy (STEM and HRTEM). *Micron* 151 (2021), 103141.

[11] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE, 730–739.

[12] Sheng Di, Jinyang Liu, Kai Zhao, Xin Liang, Robert Underwood, Zhaorui Zhang, Milan Shah, Yafan Huang, Jiajun Huang, Xiaodong Yu, et al. 2024. A Survey on Error-Bounded Lossy Compression for Scientific Datasets. *arXiv preprint arXiv:2404.02840* (2024).

[13] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. 1997. TOP500 supercomputer sites. *Supercomputer* 13 (1997), 89–111.

[14] Alex Fallin, Noushin Azami, Sheng Di, Franck Cappello, and Martin Burtscher. [n. d.]. Fast and Effective Lossy Compression on GPUs and CPUs with Guaranteed Error Bounds. ([n. d.]).

[15] Alex Fallin and Martin Burtscher. 2024. Lessons Learned on the Path to Guaranteeing the Error Bound in Lossy Quantizers. *arXiv preprint arXiv:2407.15037* (2024).

[16] Hao Feng, Boyuan Zhang, Fanjiang Ye, Min Si, Ching-Hsiang Chu, Jiannan Tian, Chunxing Yin, Summer Deng, Yuchen Hao, Pavan Balaji, et al. 2024. Accelerating Communication in Deep Learning Recommendation Model Training with Dual-Level Adaptive Lossy Compression. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[17] Jay Gambetta. 2020. IBM's roadmap for scaling quantum technology. https://www.ibm.com/quantum/blog/ibm-quantum-roadmap?mhsrc=ibmsearch_a&mhq=condor.

[18] Ray W Grout, A Gruber, H Kolla, P-T Bremer, JC Bennett, A Gyulassy, and JH Chen. 2012. A direct numerical simulation study of turbulence and flame structure in transverse jets analysed in jet-trajectory based coordinates. *Journal of Fluid Mechanics* 706 (2012), 351–383.

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

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

2366–2369.

[21] Yafan Huang, Sheng Di, Guanpeng Li, and Franck Cappello. 2024. CUSZP2: A GPU Lossy Compressor with Extreme Throughput and Optimized Compression Ratio. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–18.

[22] Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. 2023. cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[23] 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.

[24] Intel. [n. d.]. SYCLomatic. https://github.com/oneapi-src/SYCLomatic.

[25] Jinda Jia, Cong Xie, Hanlin Lu, Daoce Wang, Hao Feng, Chengming Zhang, Baixi Sun, Haibin Lin, Zhi Zhang, Xin Liu, et al. 2024. SDP4Bit: Toward 4-bit Communication Quantization in Sharded Data Parallelism for LLM Training. *arXiv preprint arXiv:2410.15526* (2024).

[26] Wenqi Jia, Youyuan Liu, Zhewen Hu, Jinzhen Wang, Boyuan Zhang, Wei Niu, Junzhou Huang, Stavros Kalafatis, Sian Jin, and Miao Yin. 2024. NeurLZ: On Enhancing Lossy Compression Performance based on Error-Controlled Neural Learning for Scientific Data. *arXiv preprint arXiv:2409.05785* (2024).

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

[28] P. Klacansky. 2017. Open SciVis Datasets. https://klacansky.com/open-scivis-datasets/.

[29] Pavol Klacansky, Haichao Miao, Attila Gyulassy, Andrew Townsend, Kyle Champley, Joseph Tringe, Valerio Pascucci, and Peer-Timo Bremer. 2022. Virtual inspection of additively manufactured parts. In *2022 IEEE 15th Pacific Visualization Symposium (PacificVis)*. IEEE, 81–90.

[30] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip: A high-throughput parallel lossless compressor for scientific data. In *2021 Data Compression Conference (DCC)*. IEEE, 103–112.

[31] Ang Li, Bo Fang, Christopher Granade, Guen Prawiroatmodjo, Bettina Heim, Martin Roetteler, and Sriram Krishnamoorthy. 2021. Sv-sim: scalable pgas-based state vector simulation of quantum circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

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

[33] Yuke Li, Arjun Kashyap, Weicong Chen, Yanfei Guo, and Xiaoyi Lu. 2024. Accelerating lossy and lossless compression on emerging bluefield dpu architectures. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 373–385.

[34] 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.

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

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

[37] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. 2022. SZ3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data* 9, 2 (2022), 485–498.

[38] Guo-Yuan Lien, Takemasa Miyoshi, Seiya Nishizawa, Ryuji Yoshida, Hisashi Yashiro, Sachiho A Adachi, Tsuyoshi Yamaura, and Hirofumi Tomita. 2017. The near-real-time SCALE-LETKF system: A case of the September 2015 Kanto-Tohoku heavy rainfall. *Sola* 13 (2017), 1–6.

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

[40] Jinyang Liu, Jiannan Tian, Shixun Wu, Sheng Di, Boyuan Zhang, Robert Underwood, Yafan Huang, Jiajun Huang, Kai Zhao, Guanpeng Li, et al. 2024. CUSZ-i: High-Ratio Scientific Lossy Compression on GPUs with Optimized Multi-Level Interpolation. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[41] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[42] nvCOMP: A library for fast lossless compression/decompression on the GPU. [n. d.]. https://github.com/NVIDIA/nvcomp.

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

[44] Molly A O'Neil and Martin Burtscher. 2011. Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. 1–7.

[45] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[46] QMCPACK: many-body ab initio Quantum Monte Carlo code. 2019. http://vis.computer.org/vis2004contest/data.html. Online.

[47] Shihui Song, Yafan Huang, Peng Jiang, Xiaodong Yu, Weijian Zheng, Sheng Di, Qinglei Cao, Yunhe Feng, Zhen Xie, and Franck Cappello. 2024. Ceresz: Enabling and scaling error-bounded lossy compression on cerebras cs-2. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 309–321.

[48] Shihui Song, Robert Underwood, Sheng Di, Yafan Huang, Peng Jiang, and Franck Cappello. 2025. A Memory-efficient and Computation-balanced Lossy Compressor on Wafer-Scale Engine. In *2025 ieee international parallel and distributed processing symposium (ipdps)*. IEEE.

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

[50] 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.

[51] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. 2019. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–24.

[52] 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.

[53] Zhuoxun Yang, Sheng Di, Longtao Zhang, Ruoyu Li, Ximiao Li, Jiajun Huang, Jinyang Liu, Franck Cappello, and Kai Zhao. 2025. PSZ: Enhancing the SZ Scientific Lossy Compressor With Progressive Data Retrieval. *arXiv preprint arXiv:2502.04093* (2025).

[54] 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.

[55] Boyuan Zhang, Bo Fang, Fanjiang Ye, Yida Gu, Nathan Tallent, Guangming Tan, and Dingwen Tao. 2024. Overcoming memory constraints in quantum circuit simulation with a high-fidelity compression framework. *arXiv preprint arXiv:2410.14088* (2024).

[56] Boyuan Zhang, Luanzheng Guo, Jiannan Tian, Jinyang Liu, Daoce Wang, Fanjiang Ye, Chengming Zhang, Jan Strube, Nathan R Tallent, and Dingwen Tao. 2025. High-performance Visual Semantics Compression for AI-Driven Science. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 557–559.

[57] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. 2023. Fz-gpu: A fast and high-ratio lossy compressor for scientific computing applications on gpus. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 129–142.

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

[59] 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.