FZ-GPU: A Fast and High-Ratio Lossy Compressor for Scientific Computing Applications on GPUs

Boyuan Zhang*
Indiana University
Bloomington, IN, USA
bozhan@iu.edu

Jiannan Tian*
Indiana University
Bloomington, IN, USA
jti1@iu.edu

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

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

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

Xin Liang
University of Kentucky
Lexington, KY, USA
xliang@uky.edu

Dingwen Tao†
Indiana University
Bloomington, IN, USA
ditao@iu.edu

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

ABSTRACT

Today's large-scale scientific applications running on high-performance computing (HPC) systems generate vast data volumes. Thus, data compression is becoming a critical technique to mitigate the storage burden and data-movement cost. However, existing lossy compressors for scientific data cannot achieve a high compression ratio and throughput simultaneously, hindering their adoption in many applications requiring fast compression, such as in-memory compression. To this end, in this work, we develop a fast and high-ratio error-bounded lossy compressor on GPUs for scientific data (called FZ-GPU). Specifically, we first design a new compression pipeline that consists of fully parallelized quantization, bitshuffle, and our newly designed fast encoding. Then, we propose a series of deep architectural optimizations for each kernel in the pipeline to take full advantage of CUDA architectures. We propose a warp-level optimization to avoid data conflicts for bit-wise operations in bitshuffle, maximize shared memory utilization, and eliminate unnecessary data movements by fusing different compression kernels. Finally, we evaluate FZ-GPU on two NVIDIA GPUs (i.e., A100 and RTX A4000) using six representative scientific datasets from SDRBench. Results on the A100 GPU show that FZ-GPU achieves an average speedup of 4.2× over cuSZ and an average speedup of 37.0× over a multi-threaded CPU implementation of our algorithm under the same error bound. FZ-GPU also achieves an average speedup of 2.3× and an average compression ratio improvement of 2.0× over cuZFP under the same data distortion.

CCS CONCEPTS

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

KEYWORDS

Lossy compression; scientific data; GPU; performance.

*Boyuan Zhang and Jiannan Tian are co-first authors.
†Corresponding author: Dingwen Tao, Department of Intelligent Systems Engineering, Luddy School of Informatics, Computing, and Engineering, Indiana University.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
HPDC '23, June 16–23, 2023, Orlando, FL, USA
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0155-9/23/06…$15.00
https://doi.org/10.1145/3588195.3592994

ACM Reference Format:
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 (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3588195.3592994

1 INTRODUCTION

Motivation. Large-scale scientific applications running on high-performance computing (HPC) systems produce vast data for post hoc analysis. For instance, Hardware/Hybrid Accelerated Cosmology Code (HACC) [1, 2] may produce petabytes of data in hundreds of snapshots when simulating one trillion particles. Storing such a large amount of data could be vastly inefficient, especially to parallel file systems (PFS) with relatively low I/O bandwidth [3, 4].

Data reduction is becoming an effective method to resolve the big data issue in scientific applications. Although traditional lossless data reduction methods such as data deduplication and lossless compression can guarantee no information loss, they suffer from limited compression ratios on scientific datasets. Specifically, deduplication usually reduces the scientific data size by only 20% to 30% [5], and lossless compression achieves a compression ratio of up to ~2:1 [6]. However, the data reduction ratios provided by these methods are much lower than the ratios scientists desire [7].

Error-bounded lossy compressors have been studied for years to address this issue for scientific data reduction. Not only can they achieve very high compression ratios (e.g., over 100×) [3, 8–10], but they can also strictly control data distortion concerning user-set error bounds. Notably, a satisfying lossy compressor designed for scientific data reduction should address three primary concerns simultaneously: (1) high compression ratio, (2) high throughput, and (3) high compression quality (data fidelity). Most of the existing error-bounded lossy compressors (such as SZ [8, 9, 11], FPZIP [12], ZFP [10]), however, are mainly designed for CPU architectures, which cannot meet the high-throughput requirement. For example, X-ray imaging data generated on advanced instruments such as LCLS-II laser [13] can result in a data acquisition rate of 250 GB/s [7]. As such, high compression throughput is essential to store large amounts of data for scientific projects efficiently.

Limitations of state-of-the-art approaches. Existing error-bounded lossy compressors for GPUs (such as cuSZ [14], cuZFP [15], and

MGARD-GPU [16]) suffer from either low throughputs or low compression ratios. Specifically, although cuZFP has slightly higher throughput compared with cuSZ and MGARD-GPU, it supports only the fixed-rate mode [17], which suffers much lower compression quality than the fixed-accuracy mode (a.k.a error-bounded mode) [18], significantly limiting its adoption in practice. On the other hand, cuSZ and MGARD-GPU can achieve much higher compression ratios than cuZFP, but their compression throughputs are relatively low. This is because both MGARD and SZ algorithms require entropy and dictionary encoding to achieve high compression ratios due to the aggregate repeated symbols (e.g., quantization codes generated by the prediction and quantization stages in SZ). At the same time, (1) MGARD-GPU uses DEFLATE [19] (including Huffman entropy encoding [20] and LZ77 dictionary encoding [21]) on the CPU, causing low throughput, and (2) cuSZ adopts an inefficient GPU-based Huffman encoding [14].

Specifically, Huffman encoding and most dictionary encoding algorithms contain substantial data dependencies, making them difficult to parallelize on GPUs extensively. Moreover, designing an efficient parallel dictionary encoding is challenging because of the intrinsic dependency in its repeated sequence search. Thus, cuSZ leaves this part to the CPU, which incurs high time overhead, including data transfer. Furthermore, achieving high performance on GPUs requires maximizing the parallelism of GPU threads and using shared memory and mitigating issues of coherence, warp divergence, and bank conflicts. Thus, it is challenging to develop an efficient GPU-based error-bounded lossy compressor that simultaneously achieves a high compression ratio and high throughput.

Key insights and contributions. In this work, we propose a fast and high-ratio error-bounded lossy compressor (called FZ-GPU [1]) for scientific computing applications on GPU based on the cuSZ framework [14], which maximizes the overall throughput. Specifically, we first propose to use bitshuffle [22] to rearrange the quantization codes generated by the prediction-and-quantization step (called "dual-quantization") in cuSZ (will be introduced in §2.2) at the bit level to increase the data correlation for more effective encoding. We then design a new fast GPU lossless encoding method for bitshuffled data. We carefully design a GPU kernel to fuse bitshuffle and encoding operations to reduce unnecessary data movements between global and shared memories. Using the proposed bitshuffle and encoding approach, we can eliminate the inefficient Huffman encoding from cuSZ. Moreover, we optimize the performance of the dual-quantization method by eliminating the outliers handling mechanism and the data shift operation, hence improving the effectiveness of the subsequent bitshuffle process.

The main contributions of our work are summarized as follows.

- We propose a new compression pipeline based on the cuSZ framework, which consists of our optimized dual-quantization, bitshuffle, and our proposed lossless encoding after bitshuffle (to replace the slow Huffman encoding implementation) on GPUs.
- We optimize the dual-quantization by eliminating the data-shift and outlier-handling operations, improving both compression throughput and bitshuffle efficiency, thus increasing the compression ratio.
- We develop a GPU bitshuffle kernel with a warp-level optimization to avoid data conflicts in bit-wise operations. We also maximize the utilization of shared memory to improve the performance of this memory-intensive kernel.
- We design a new lossless encoding method that leverages the data characteristics after bitshuffle and the high parallelism of GPUs. It can effectively and efficiently remove the high redundancy from bitshuffle, thereby achieving a high compression ratio and throughput.
- We carefully fuse the bitshuffle kernel and the first phase of the encode kernel (i.e., recording zero blocks) into a single GPU kernel to eliminate unnecessary data movements between the GPU's global and shared memory.
- We evaluate FZ-GPU on six real-world scientific application datasets from *Scientific Data Reduction Benchmarks* [23] on two NVIDIA GPUs (i.e., A100 and RTX A4000) and compare it to four state-of-the-art compressors. Experiments show that on the A100 GPU, FZ-GPU significantly improves the compression throughput by up to 11.2× over cuSZ; and compared to cuZFP, FZ-GPU achieves an average of 2.0× higher compression ratio at the same data distortion with an average speedup of 2.3×.

Limitations of the proposed approach. Compared to cuSZ, FZ-GPU significantly improves the compression throughput in all cases, while it has a slightly lower compression ratio at low error bounds. Compared to cuSZx, FZ-GPU has much higher compression ratios and hence higher overall data-transfer throughput, but its compression throughput is lower than cuSZx. Compared to cuZFP, FZ-GPU has a slightly lower compression ratio and throughput at large error bounds (e.g., over 5e−3) in some datasets.

The remaining of this paper is organized as follows. In §2, we present the background about GPU lossy compression for scientific data, cuSZ framework, prediction-and-quantization method, and our problem statement. In §3, we describe the design of our proposed FZ-GPU. In §4, we evaluate FZ-GPU on different scientific datasets and compare it with other compressors. In §5, we discuss related work on GPU-based lossy compression. Finally, in §6, we conclude our work and discuss future work.

2 BACKGROUND AND PROBLEM STATEMENT

In this section, we introduce the background about GPU-based lossy compression and cuSZ framework and our problem statement.

2.1 GPU Lossy Compression for Scientific Data

There are two main data compression classes: lossless and lossy. Compared to lossless compression, lossy compression can provide a much higher compression ratio by trading an acceptable accuracy loss. Lossy compressors for image and video have been well studied, such as JPEG [24] and MPEG [25], but they are human perception-driven rather than designed for scientific postanalysis and lack error-controlling mechanism.

Recently, a new generation of lossy compression for scientific data, especially floating-point data, has been developed, such as SZ [3, 8, 9], ZFP [10], MGARD [26], and TTHRESH [27]. Unlike lossy compressors for images and video, these lossy compressors

[1]The code is available at https://github.com/szcompressor/FZ-GPU.

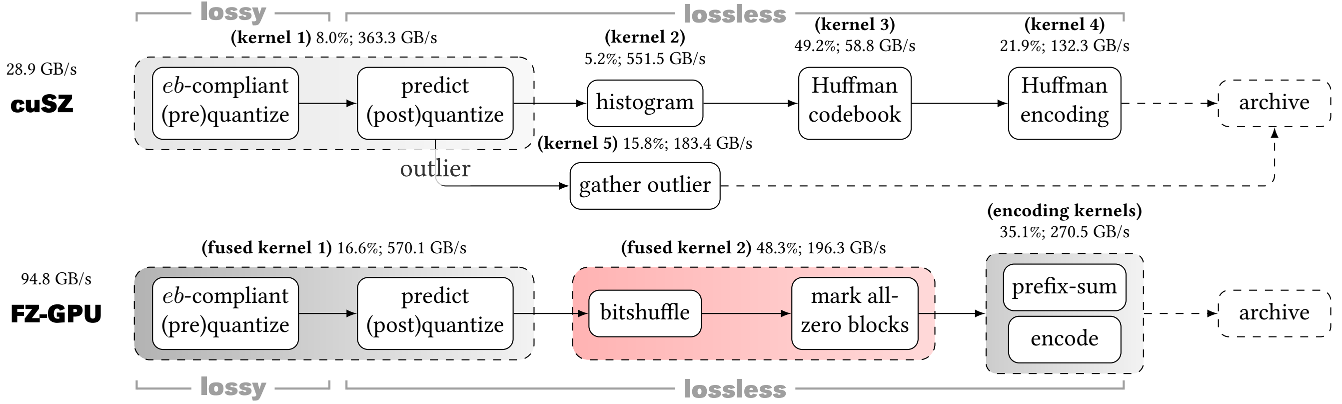

Figure 1: Our proposed new compression pipeline ("FZ-GPU") versus original cuSZ's compression pipeline. Each kernel is marked with its relative time (in percentage) and throughput (in GB/s) based on one field from the Hurricane dataset at an error bound of 1e-4.

provide strict error-controlling schemes, allowing users to control the accuracy loss in reconstructed data and even in post-analysis.

Considering the boom in GPU-based HPC systems and applications, SZ, ZFP, and MGARD are starting to roll out their GPU versions using CUDA [28] (i.e., cuSZ [14], cuZFP [15], and MGARD-GPU [16]), which provide much higher throughputs for compression compared with their CPU versions. cuZFP (a transform-based compressor) allows the user to specify the desired bitrate (i.e., the average number of bits per value after compression), while cuSZ (a prediction-based compressor) and MGARD-GPU (a multigrid-based compressor) allow the user to specify the maximum error that can be tolerated. cuZFP with the fixed-rate mode can provide stably higher compression throughput, whereas cuSZ and MGARD-GPU with the error-bounded mode tend to achieve a higher compression ratio. In addition, there are also some optimization works based on these compressors to improve either compression ratio or compression throughput, which will be discussed in §5.

2.2 cuSZ Framework

Since scientific data are mainly in floating-point representation, the randomly distributed bits in the exponent and mantissa are the major obstacle to significantly reducing the data size. This is because a change in floating-point values causes the exponent and mantissa representation to change from the most significant bit (MSB) to the least significant bit (LSB); even close values can have distinct bitsets. In comparison, a change in integer values results in fewer bit-level changes.

Thus, SZ framework converts the original floating-point data to integers in two stages: (1) it first *predicts* the value of each data point using a prediction function such as Lorenzo predictor [29] and generates prediction errors (still floating-point values), which are the differences between the predicted and the original values, and (2) it then *quantizes* the prediction errors to integers to reduce the bit randomness. After the prediction and quantization, lossless encoding works effectively on the integers (i.e., the approximation of prediction errors). The lossless encoding in SZ, such as gzip [30] or Zstd [31], includes Huffman encoding and a dictionary encoding. In addition, cuSZ, the GPU implementation of SZ framework, follows a similar compression pipeline with two primary adjustments in favor of performance, (1) it performs quantization on the original data before the prediction to remove the tight data dependency [14], and (2) it omits the dictionary encoding stage.

2.3 Dual-Quantization Method

For the prediction and quantization stages, cuSZ uses *dual-quantization* method to achieve fine-grained parallelization; not only can chunked data blocks be compressed independently, but each data point can also be processed in parallel. Specifically, cuSZ first splits the whole dataset into multiple chunks. Then, it performs pre-quantization, Lorenzo prediction, and post-quantization. Note that pre-quantization is the only lossy stage (introducing compression errors) in the entire compression pipeline.

We denote the input data as d and the user-specified error bound as eb; the compression conducts the error-controlling process illustrated in Figure 2. The error-boundness (i.e., the decompressed data error from the original is no greater than eb) can be guaranteed as

$$|\,\mathrm{round}(d_i/(2\cdot eb))\times(2\cdot eb)-d_i|\leq eb.$$

With the given parameter r, the output comprises two parts, quantization code $q' = q + r$ of limited numbers such that $q' = q - r$ and $-r < q < r$, and outlier that is out of range $(-r, r)$. We refer readers to the cuSZ papers [14, 32] for more details.

Figure 2: An illustration of error controlling in SZ.

2.4 Problem Statement

While flourishing to achieve high data processing capabilities, GPU-based compressors target high versatility and a wide range of usage scenarios, such as in-memory compression [33], compression of MPI messages [34], and reducing CPU-GPU data transfer time [35]. On the one hand, the current cuZFP only allows it to use where high compression throughput is the priority, as its compression quality is low compared to cuSZ under the same compression ratio. Moreover, cuZFP does not support error-bounded mode. MGARD-GPU can only provide very low compression throughput (will be shown in

the evaluation). On the other hand, cuSZ's modularized design enables us to investigate/design new compression components and replace specific ones in the pipeline if needed. For example, the current cuSZ is limited by its inefficient Huffman encoder[2]. As a result, in this work, we mainly focus on the error-bounded lossy compression framework cuSZ, which can provide high compression ratios, and aim to drastically improve the compression throughput by designing a new high-performance GPU encoding approach to replace Huffman encoding in the pipeline.

In this work, we assume the data to compress is generated by scientific applications on GPUs, and then the compression would be directly performed on the data from the GPU memory; finally, the compressed data would be saved from GPUs to disks via CPUs. There are several use cases that FZ-GPU targets. For example, it can reduce storage overhead as the compressed data will be saved from the GPU to the disk through the CPU for post-analysis. It can also reduce memory overhead as the compressed data will be cached in the GPU global memory and decompressed on the GPU directly when the reconstructed data is needed for computation.

3 DESIGN OF PROPOSED FZ-GPU

In this section, we present the design of our new GPU-based lossy compressor FZ-GPU with a series of optimizations.

3.1 Overview of New Compression Pipeline

Our proposed new compression pipeline is shown in Figure 1. To fully utilize the computing power of GPUs, we aim to design a pipeline that maximizes parallelism while achieving high compression ratios by exploiting potential data patterns.

Inspired by cuSZ, we adopt the dual-quantization method in the first stage of our compression pipeline for three reasons. (1) Its Lorenzo predictor exploits the spatial and dimensional information to reduce the entropy of input data significantly [9]. (2) It is fully parallelized, and its Lorenzo predictor is highly efficient due to the $O(n)$ time complexity. (3) Its quantization provides an error-controlling scheme for our lossy compression pipeline. However, the original quantization design in cuSZ sets a threshold to separate regular quantization codes and outliers (§2.3); though it favors a higher compression ratio, the amount of memory transaction hinders the performance, and hence it is not used in our design. Instead, we propose to optimize dual-quantization by neither shifting quantization codes nor handling outliers. Besides, we use the MSB to denote the sign of the data point. We will describe the detail of the optimized dual-quantization method in §3.2.

After that, we seek a new lossless encoding method that can provide high throughput and high compression ratios at the same time. On the one hand, cuSZ uses Huffman encoding that causes irregular memory access (i.e., the number of bits varies for each symbol). Thus, we look for a lossless encoding with more regular memory access. On the other hand, Huffman encoding achieves a high compression ratio but cannot handle sparse data (efficient prediction minimizes prediction errors in amplitude). Thus, it is critical to identify a representation for quantization codes that can expose as many continuous zero bits as possible.

[2] The Huffman encoding on the GPU includes building a large Huffman codebook and performing coarse-grained encoding based on the Huffman tree.

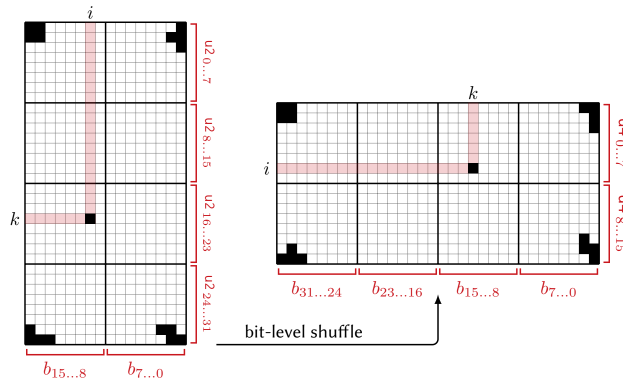

Figure 3: An illustration of bitshuffle algorithm.

To this end, we propose to adopt bitshuffle [22] (as illustrated in Figure 3) before performing encoding. The advantages of using bitshuffle are twofold, (1) it transforms the data representation to create more space redundancy for subsequent lossless compression, and (2) it is a highly parallel process well-suited for GPU processing. However, the bitshuffle operation is more time-consuming than the dual-quantization operation. Thus, we propose to optimize its performance by using warp-level functions and utilizing shared memory. We will present bitshuffle and our optimization in §3.3.

Lastly, we propose a sparsification-style fast lossless encoding after bitshuffle. Specifically, we partition the data into many data blocks and then go through each data block to check if all values are zero: if so, we use a 0-bit to record the block; otherwise, we use a 1-bit to record it and copy this block to the output compressed array. However, this lossless encoding process is hard to achieve high performance on the GPU because the encoded address offsets are unknown for different data blocks. Therefore, we need to precompute the offset (i.e., the starting point of the memory address) and encode each data block according to its offset. The detail of our proposed lossless encoding method will be described in §3.4.

Compared to cuSZ, both FZ-GPU and cuSZ use prediction and quantization to reduce the entropy of the datasets. However, the lossless encoding of our work is entirely different from cuSZ. Instead of utilizing the time-consuming Huffman encoding to compress the quantization code (output of prediction and quantization), we propose to use a simple but effective pipeline with bitshuffle and our proposed lossless encoding. By doing this, the compression throughput is significantly increased, as shown in Figure 3. Moreover, the compression ratio is no longer limited by Huffman encoding (an upper bound of 32), which means it is potentially increased. It is worth noting that we also modify the pre-quantization kernel to fit our pipeline by integrating the outliers and using the most significant bit to store the sign of the number, which also increases the throughput of the pre-quantization kernel.

3.2 Proposed Optimized Dual-Quantization

We employ dual-quantization in the first stage of our compression pipeline because it can significantly reduce the entropy of input data by exploring the spatial correlation through the Lorenzo predictor [9]. Furthermore, its fine-grained parallelism with low time complexity (i.e., $O(n)$) further facilitates an efficient GPU implementation. Moreover, its quantization provides an error-controlling

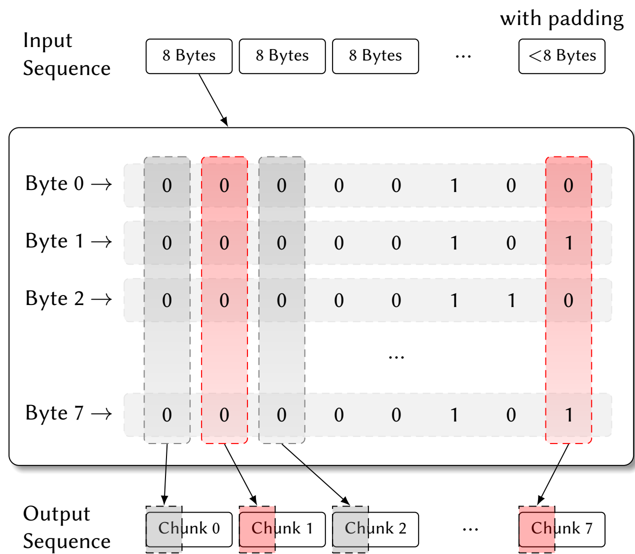

Figure 4: A simplistic fine-grained parallel bitshuffle [36].

scheme for lossy compression, which enables an error-bounded mode similar to cuSZ. However, the original dual-quantization method handles outliers by compressing them separately and shifts all quantization codes by a radius for a higher compression ratio, which leads to throughput degradation. Therefore, we propose an optimized dual-quantization method for higher performance. The main differences between the original and our optimized dual-quantization methods are threefold, (1) we remove shift operations to formulate values symmetrically distributed around zero, (2) we avoid separate handling of outliers for high performance, and (3) we use 1 bit to handle the sign of each quantization code instead of using 2's complement. We describe them in detail as follows.

First, we optimize the representation of quantization codes. According to our empirical analysis, although the original data type is a float of four bytes, most data can be denoted as less than four bytes after quantization. Thus, we propose to use two bytes to represent the quantization code, which indirectly achieves compression by transforming the data type. Note that the out-of-range data points are very few compared to the whole dataset. Thus, even losing these elements' precision will not significantly affect the decompressed data quality, such as peak signal-to-noise ratio (PSNR).

Next, we optimize the mechanism that handles the outliers in the prediction. The prediction in cuSZ sets a threshold to distinguish outliers and normal data points. cuSZ will compress outliers separately because the compression ratio of Huffman encoding in the next step depends on the entropy of the data. If the entropy is too high, Huffman encoding will need more bits to denote patterns. We note that it is unnecessary to separate the outliers and normal data points when replacing Huffman encoding with our proposed lossless encoder. Therefore, we propose to discard the outliers handling in our pipeline.

Furthermore, we modify the negative numbers' data format to fit our encoding kernel's design. Specifically, instead of storing the data as a signed integer, we use an unsigned integer. We use the corresponding positive number with the most significant bit set as one for the negative number. This is because a negative number is represented as two's complement, consisting of many set bits when its absolute value is small. This is unsuitable for our design because we expect the data bytes to have as many zeros as possible. To solve this issue, we propose to use the first bit of unsigned int to denote the positive and negative because the efficient prediction will keep the data in a small range around zero, which guarantees the valid number that two bytes can represent is more than enough for the quantization code. As aforementioned, few data points are out of range, so this modification accelerates the dual-quantization kernel due to fewer if-else branches and easier operations.

3.3 Optimization of Bitshuffle on GPUs

Bitshuffle is an algorithm that re-organizes the dataset bit-wise by gathering the n-th bits of all the bytes in eight chunks. Figure 3 shows an example of bitshuffle. Bitshuffle fits our compression pipeline for two reasons: (1) it creates more spatial redundancy for the following lossless compression, and (2) there is no data dependency in bitshuffle, meaning it is highly parallelizable. However, the bitshuffle operation is time-consuming and needs bit-level operations. Therefore, we propose a series of optimizations to improve its performance, as described below.

The first optimization is fully leveraging shared memory in each thread block. Bitshuffle is a memory-intensive process that needs to access the same memory multiple times to get different bits of the same byte. Direct access to global memory has much higher latency than shared memory. Therefore, we propose to use shared memory to reduce the memory access overhead. Since we need to combine as many bits as possible to create more spatial redundancy at the bit level, we need to set shared memory size as large as possible to store more data. In our kernel design, we use a 32-by-32 array of unsigned integers, 4 bytes per array element (each element saves two quantization codes) to store the data. We set the thread block size to 32-by-32, corresponding to shared memory size. Note that the actual size of the 2D array in shared memory is 32-by-33 with padding to avoid bank conflicts.

After loading the data into shared memory, we need to extract the corresponding bits and put them together. This operation is challenging for GPU, since writing to the same memory location by all threads in a warp will cause data access conflicts. However, if we perform this 1-bit operation at a time, the parallelism advantage of GPU is much undermined; in other words, there would always be threads waiting for others to complete. To solve this issue, we use a warp-level vote function, `__ballot_sync()` (requiring `uint32_t` as the input type), to speed up this operation. The vote function takes variable v_a of each thread in a warp as input and outputs a `uint32_t` v_b. More specifically, v_a of thread i is used in the predicate to set i-th bit of v_b with true (1) or false (0). Therefore, we can extract certain bits of the element in the array and use the vote function to implement the shuffle process without sacrificing the parallelism.

Then, we need to put the bitshuffled result back into the global memory to continue the encoding process. The simplistic way is to store the shuffled data independently in eight chunks, as shown in Figure 4. However, the memory access of this simplistic solution is non-coalesced, which would cause a significant drop in throughput. To solve this issue, we propose another optimization, as shown in Figure 5. We store the result locally in the same thread block. The

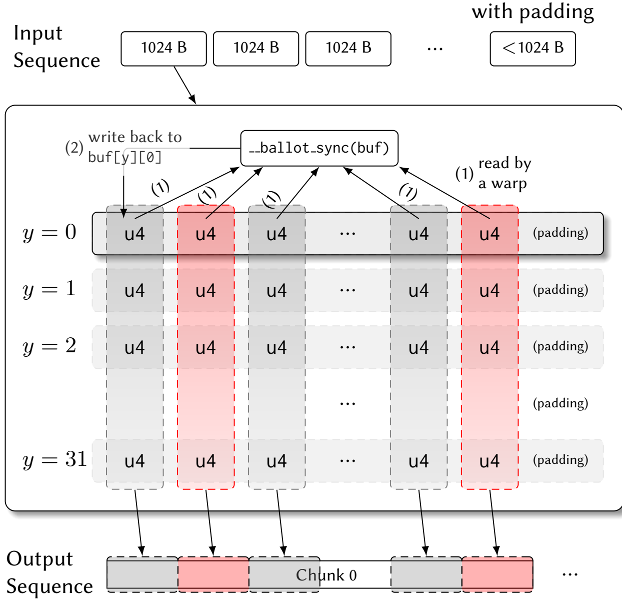

Figure 5: Our proposed scalable GPU bitshuffle method. u4 stands for 4-byte unsigned integer type.

compression ratio will not be affected if the granularity is coarser than the following encoder. We process the data row-wise so that the bits in the same order are stored in the same column. We then write back column-wise. Note that padding allows us to access the data points column-wise without bank conflict.

3.4 Proposed Fast GPU Lossless Encoder

The prior study [22] finds that bitshuffle works well with LZ4 lossless encoding on scientific floating-point data. However, the LZ4 algorithm is unsuitable for GPU architectures [3] due to the sequential nature of its search for repeated strings (similar to all LZ-family compression algorithms such as LZ77 and LZSS), leading to relatively low throughput. To this end, we propose a new lossless encoding method to replace the LZ4 encoder and couple it with bitshuffle. The overview of our proposed encoding method is presented in Figure 6. Specifically, this encoder has two phases. The first phase is to partition the data into data blocks and then iterate all data blocks to record whether all values in the data block are zeros or not in an array, called the "bit-flag array". Then, the second phase encodes the data based on the flag array generated in the first phase. During the encoding, if the corresponding bit flag is '0', we use this zero to denote it; if the bit flag is '1', we copy the whole block to our compressed data. This sparsification-style encoder is highly suitable for bitshuffle because bitshuffle reorganizes the representation of quantization codes at the bit level, creating many consecutive zero bits/bytes and hence zero blocks.

However, it is non-trivial to implement this encoding method efficiently on the GPU because the size of each data block after compression varies. Thus, we need to determine the memory offset for every data block before encoding it. We use the prefix-sum to

[3] LZ4 from nvCOMP [37] can only achieve 6.3 GB/s on our evaluation datasets.

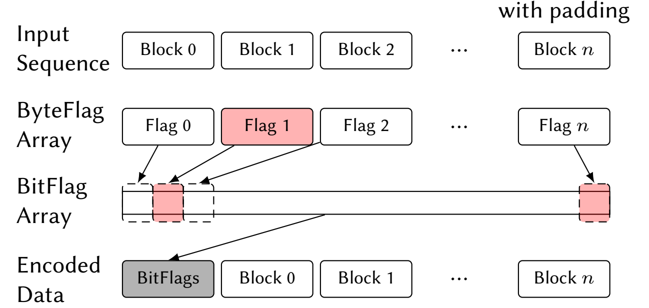

Figure 6: Our proposed fast GPU encoding method.

compute the memory offsets. To implement an efficient GPU prefix sum, we need device-wide synchronization to ensure sizes for all compressed blocks are ready. Two approaches can achieve this global synchronization. The first approach is to use the cooperative group API [38]. However, the maximum possible number of threads is limited and unsuitable for our problem. Another approach is to split one kernel into two since a synchronization can be conveniently triggered when a GPU kernel exits. Thus, we propose two phases in our encoding method, with the two corresponding optimized kernels detailed below.

To take advantage of the result stored in shared memory in the bitshuffle kernel to save the time to read again from global memory, we propose to fuse the bitshuffle and the first phase of our encoding in a single kernel. Note that while the granularity of the two processes is different, meaning that some threads in the kernel will be idle while others are executing, it is still more cost-effective than accessing global memory (which will be proved in the evaluation). Once the data is ready, we use statically allocated buffers in shared memory (Lines 2–3) to store the flags of each data block. We then use fewer threads than in bitshuffle to iterate over the data blocks and record a flag indicating whether all data points in the same data block are zero (Lines 14–16). The result will be temporarily stored in an `uint8_t` array, called `ByteFlagArr` (declared in Line 3). Finally, we convert the byte-flag array to the bit-flag array through the bit-level operations (Lines 18–20). Note that we also use the warp-level vote function to generate the bit-flag array to avoid data access conflicts (Line 21). In addition, instead of wasting the byte-flag array, we will use it to calculate the prefix-sum for the offsets of data blocks in the second phase. The proposed fused kernel is shown below in detail.

```
1  __shared uint32_t buf[32][33]
2  __shared uint32_t BitFlagArr[8]
3  __shared uint8_t  ByteFlagArr[256]
4  uint32_t cur
5  ltid = get_linear_threadid()
6
7  buf[Idx.y][Idx.x] = input[offset]; __syncthreads()
8
9  cur = buf[Idx.y][Idx.x]
10 for i in range(32):
11   buf[Idx.y][Idx.x] = __ballot_sync(cur & (1U << i))
12 output[offset] = cur = buf[Idx.x][Idx.y]
13
14 if Idx.x * 4 < 32:
15   for i in range(4):
16     ByteFlagArr[ltid] = any(buf[Idx.x*4+i][Idx.y] != 0)
```

```
17
18 if Idx.y < 8:
19   cur = ByteFlagArr[ltid]
20   BitFlagArr[Idx.y] = __ballot_sync(buf)
21
22 WriteBackToGlobalMem(ByteFlagArr)
23 WriteBackToGlobalMem(BitFlagArr)
```

For the second phase, we directly call the high-performance ExclusiveSum function (Line 1) from `NVIDIA::CUB` library [39]. Then, we can obtain the memory offset of each compressed data block. After that, we launch our encode kernel to write the compressed data back to the output array in the global memory (Lines 8-9). Note that if the corresponding data block has a valid offset[4], the compressed data block will be saved; otherwise, it will be discarded. The detail of our second kernel is shown below.

```
1 PrefixSum(ByteFlagArr, PreSum)
2 __shared uint32_t sumArr[33]
3
4 ltid = get_linear_threadid()
5 SumArr[0] = PreSum[ltid]
6 SumArr[Idx.x+1] = PreSum[gid+1]
7
8 if SumArr[Idx.x+1] != SumArr[Idx.x]:
9   output[offset] = input[ltid]
```

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

Platforms. We use two platforms in our evaluation: (1) One node from an HPC cluster equipped with two 64-core AMD EPYC 7742 CPUs 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 two 28-core Intel Xeon Gold 6238R CPUs at 2.20GHz and two NVIDIA GTX A4000 GPUs (40 SMs, 16 GB), running Ubuntu 20.04.5 and CUDA 11.7.99. While we use one GPU for evaluation, multi-GPU processing is considered embarrassingly parallel with regard to single-GPU processing. This is because we partition data in a coarse-grained manner to fit into a single GPU, with a data chunk independent from another. With no data dependency, the multi-GPU comparison will only involve different numbers of data chunks.

Table 1: Real-world `float`-type datasets used in evaluation.

datasets	FIELD DATA SIZE
	dimensions
		#FIELDS
		examples(s)

COSMOLOGY HACC	1,123.81 MB 280,953,867	6 in total xx, vx
CLIMATE CESM	25.92 MB 1,800×3,600	70 in total CLDICE, RELHUM
COSMOLOGY NYX	536.87 MB 512×512×512	6 in total baryon_density
CLIMATE HURRICANE	100 MB 100×500×500	13 in total CLDICE, QRAIN
QUANTUM CIRCUITS QMCPACK	630.74 MB 7,935×69×288	1 in total einspline
PETROLEUM EXPLORATION RTM	189.50 MB 449×449×235	16 in total snapshot_1200

[4]The offset is valid if it is different from its previous offset.

Datasets. We conduct our evaluation and comparison based on six typical real-world HPC simulation datasets from the Scientific Data Reduction Benchmarks [23]: HACC (cosmology particle simulation) [1], CESM (climate simulation) [40], Hurricane (ISABEL weather simulation) [41], Nyx (cosmology simulation) [42], QMCPACK (quantum Monte Carlo simulation) [43], and RTM (reverse time migration, seismic imaging for petroleum exploration) [44], which have been widely used in previous compression studies [14, 32, 45–53]. The details are shown in Table 1.

Note that to compress particle datasets such as the HACC dataset with a minimum impact on the probability density function, prior work [54] proposes to use point-wise relative error bound. To readily achieve that, an existing work [4] proposes to transform the original data using a logarithmic function and compress the log-transformed data with the corresponding absolute error bound (computed from the point-wise relative error bound). Thus, in this paper, we evaluate the log-transformed HACC dataset.

Baselines. We compare FZ-GPU with four state-of-the-art GPU lossy compressors, including cuZFP [15], cuSZ [14], cuSZx [55], and MGARD-GPU [16]. We exclude bitcomp [56] from the evaluation as it is closed-source software with an unknown compression algorithm. We use five typical relative error bounds (relative to the value range of the data field), i.e., 1e−2, 5e−3, 1e−3, 5e−4, and 1e−4. Note that when comparing the compression throughput, we evaluate cuSZ, cuSZx, and MGARD-GPU under the same error bound. In contrast, we evaluate cuZFP under the same PSNR as ours as cuZFP does not support the error-bounded mode.

4.2 Evaluation Metrics

Our evaluation metrics include (1) compression ratio, (2) distortion between original and reconstructed data, (3) compression throughput, and (4) overall throughput, which are detailed as follows.

(1) *Compression ratio* is one of the most commonly used metrics in compression research. It is a factor of the original data size to the compressed data size. Higher compression ratios mean denser information aggregation against the original data. It is worth noting that the bitrate is the average number of bits per value after compression. Since all of our evaluation datasets are single-precision floating-point data, the bitrate is calculated as 32 (bits) divided by the compression ratio. We evaluate how each compressor corresponds to data quality at a specific bitrate, which will be presented in the rate-distortion curve in §4.3.
(2) *Distortion* evaluation is crucial for evaluating lossy compression performance in data reconstruction quality. In this work, we mainly use PSNR to measure the distortion quality. Similar to prior work, we plot the rate-distortion curve for a fair comparison among different compressors and their diverse compression modes, which compares the distortion quality at the same bitrate. Moreover, we also adopt the Structural Similarity Index Measure (SSIM) to evaluate the reconstructed data quality. SSIM is a metric used to measure the similarity between two images. Details on the calculation can be found in [57].
(3) *Compression throughput* is how much data a compressor can process in one unit of time. It is a key advantage of using a GPU-based lossy compressor instead of a CPU-based one.

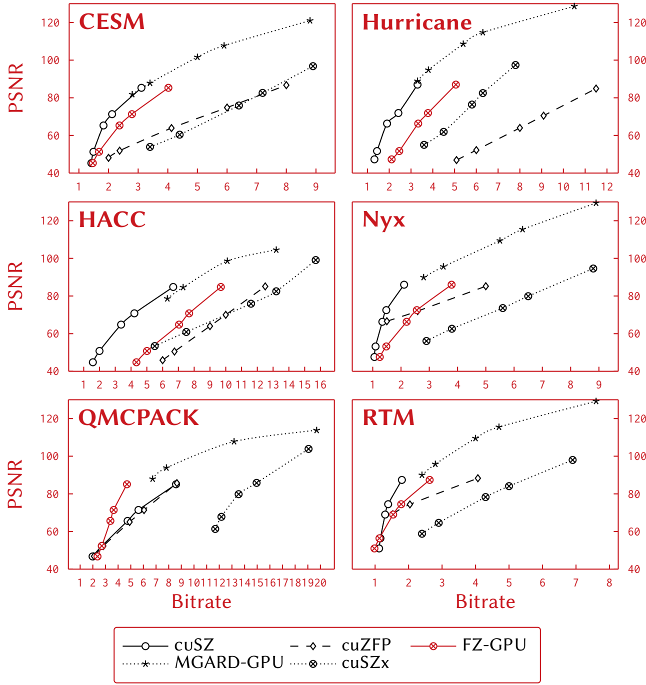

Figure 7: Rate-distortion of five GPU lossy compressors.

(4) *Overall data-transfer throughput* is to measure the performance of transferring compressed data (through the network or CPU-GPU interconnect), including compression overhead. This metric is a composite indicator of compression ratio and speed. Higher compression ratio and higher compression throughput, higher overall data transfer throughput.

4.3 Evaluation of Compression Quality

First, we compare the five compressors' rate-distortion curves (i.e., distortion in PSNR versus bitrate), as shown in Figure 7. Specifically, our platform uses different experimental settings to get the rate-distortion curves. We apply five different relative error bounds (relative to the value range) to cuSZ, MGARD-GPU, cuSZx, and FZ-GPU. Due to the fact that cuZFP does not support the error-bounded mode, we investigate a series of bitrates and select the bitrates with the same average PSNR as ours. Note that on Nyx and RTM, cuZFP cannot achieve a similar PSNR as ours with the error bounds of 1e−2 and 5e−3. As shown in Figure 7, FZ-GPU has a similar compression ratio compared to cuSZ. On the RTM dataset with high error bounds, the compression ratio of FZ-GPU is up to 1.1× higher than cuSZ and 1.7× higher than cuZFP on average. Note that FZ-GPU has good stability regarding distortion. For example, on RTM with 400 timesteps, PSNR varies only from 86.1 dB to 87.5 dB under the relative error bound of 1e−4. The analysis is detailed in the following sections.

Comparison with cuSZ. Since the lossy part (i.e., dual-quantization) of FZ-GPU is the same as cuSZ, their PSNR is the same when we use the same error bound. Therefore, our bitrate is very close to cuSZ. In some cases of the high error bound, FZ-GPU has a higher compression ratio than cuSZ. For example, Figure 7 shows that FZ-GPU has a 13.9% improvement in compression ratio with an error bound of 1e−2 on the RTM dataset. This is because the RTM dataset contains many zero values and other highly smooth values. Therefore, after we apply bitshuffle, the shuffled data is mostly zero, resulting in a high compression ratio of our designed sparsification-like encoding method. In comparison, cuSZ has the compression ratio upper bounded by 32 due to Huffman encoding. Moreover, cuSZ does not fully utilize the spatial redundancy of the RTM dataset. In contrast, our lossless encoder guarantees that the spatial redundancy is effectively compressed and the compression ratio is up to 128. Thus, the smooth values on the RTM dataset make the bitshuffled data more suitable for our lossless encoder. We note that cuSZ cannot work correctly on 3D QMCPACK due to a Huffman encoding error; therefore, we apply cuSZ on the 1D QMCPACK (flattened) for a comparison.

Comparison with cuZFP. FZ-GPU achieves a much higher compression ratio under the same average PSNR on almost all datasets compared to cuZFP, except for some high error-bound cases on Nyx and RTM. For example, cuZFP has a compression ratio of 21.3 on Nyx with the error bound of 1e−2, while FZ-GPU has a compression ratio of 14.5. This is because the two datasets under high error bounds are very smooth (most quantization codes are zeros), where cuZFP is highly effective. But cuZFP loses this advantage quickly when the error bound is lower. That is because the lower error bound gives the Nyx and RTM datasets higher entropy (like other datasets) after dual-quantization. The compression method of cuZFP cannot handle such a complex dataset effectively.

Comparison with MGARD-GPU. We note that due to memory issues, MGARD-GPU cannot work correctly on 1D datasets. For example, it cannot compress HACC on A100 with the relative error bound of 1e−4 and on A4000 with all the relative error bounds. Also, MGARD-GPU fails to compress QMCPACK with the error bound of 1e−4 because the compressed size is larger than the original size. As a result, the rate-distortion curve in Figure 7 only contains 4 points for MGARD-GPU on QMCPACK and HACC. Figure 7 shows that under the same relative error bound, MGARD-GPU has higher PSNR on all datasets because MGARD-GPU over-preserves the data distortion. Regarding rate-distortion, MGARD-GPU is similar to cuSZ and slightly better than FZ-GPU on CESM, Hurricane, and Nyx, since it uses a multi-grid-based approach with high time complexity (a large coefficient before $O(N)$) to achieve an accurate approximation. FZ-GPU is close to MGARD-GPU on RTM. For example, FZ-GPU has a bitrate of 2.6 at the error bound of 1e−4, while MGARD-GPU has a bitrate of 2.4 at the error bound of 1e−2, with similar PSNR values. On QMCPACK, FZ-GPU outperforms MGARD-GPU: specifically, FZ-GPU has a bitrate of 4.7 at the error bound of 1e−4, while MGARD-GPU has a bitrate of 6.7 at the error bound of 1e−2, again with similar PSNR values.

Comparison with cuSZx. Under the same relative error bound, FZ-GPU has a much higher compression ratio than cuSZx. Specifically, FZ-GPU has an average compression ratio improvement of 2.4× and 4.3× higher compression ratio than cuSZx at most on the QMCPACK dataset with a relative error bound of 1e−2. Although cuSZx has higher PSNR than FZ-GPU under the same error bound, FZ-GPU has a higher compression ratio under similar PSNR according to the curve shown in Figure 7. This is because FZ-GPU employs the Lorenzo predictor (to reduce the entropy of the input data)

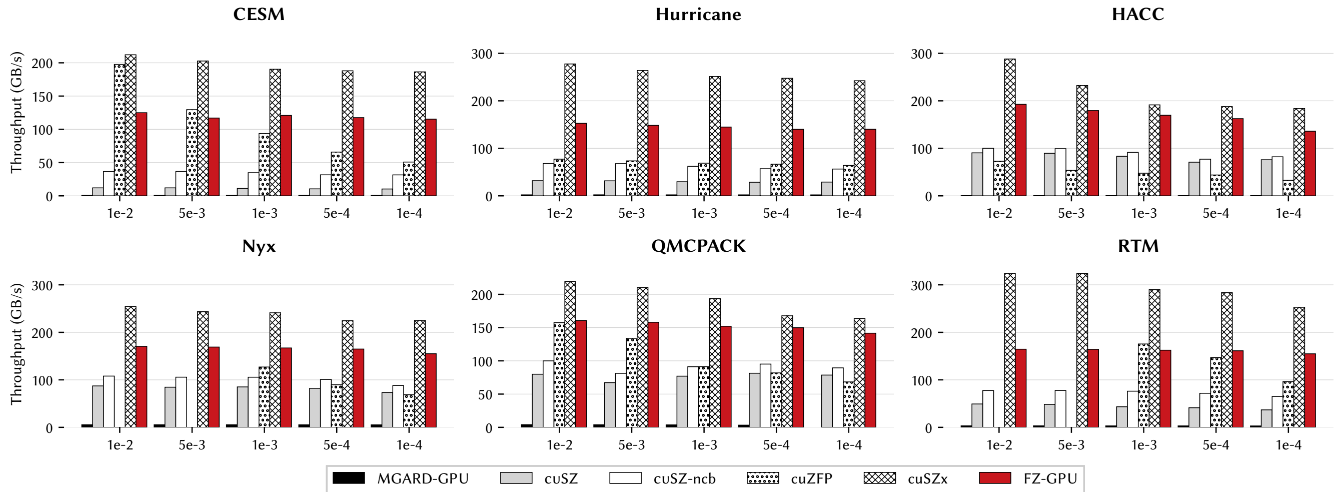

Figure 8: Compression throughput of cuZFP, cuSZ, cuSZ-ncb (cuSZ with no codebook building), cuSZx, MGARD-GPU, and FZ-GPU on NVIDIA Tesla A100. cuZFP's throughput corresponds to FZ-GPU with the same average PSNR.

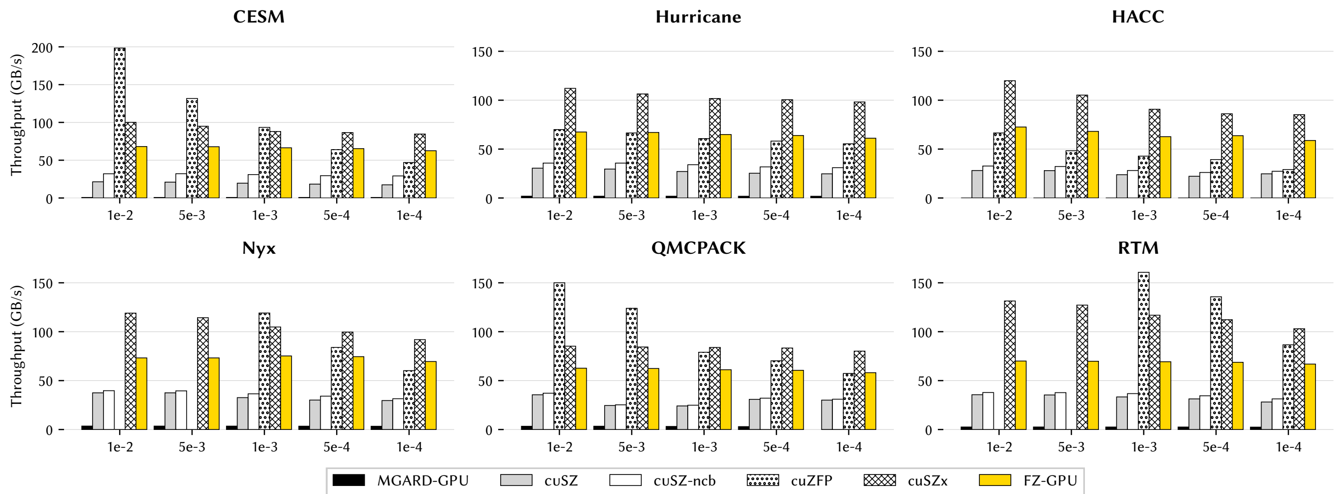

Figure 9: Compression throughput of cuZFP, cuSZ, cuSZ-ncb (cuSZ with no codebook building), cuSZx, MGARD-GPU, and FZ-GPU on NVIDIA RTX A4000. cuZFP's throughput corresponds to FZ-GPU with the same average PSNR.

and minimizes the *bitwise* data redundancy, whereas cuSZx only reduces the *blockwise* data redundancy.

4.4 Evaluation of Compression Throughput

Next, we evaluate the compression throughput of these three methods on A100 and A4000 GPUs, as shown in Figure 8 and Figure 9, respectively. Specifically, we measure their kernel time and apply five relative error bounds to cuSZ and FZ-GPU. Due to the fact that cuZFP does not support the error-bounded mode, we investigate a series of bitrates and select the bitrates that have the same average PSNRs as ours. Then, we use these bitrates to get the corresponding compression throughput. In some cases, since cuZFP cannot achieve similar PSNR, we use fewer bars to display valid results. Figure 8 illustrates that on A100, FZ-GPU achieves a speedup of up to 11.2× over cuSZ, and a speedup of up to 4.2× over cuZFP. On the A4000 platform, Figure 9 shows that FZ-GPU achieves a speedup of up to 3.6× over cuSZ, and a speedup of up to 2.0× over cuZFP. Note that cuSZ includes all cuSZ's kernels, while cuSZ-ncb does not include the time to generate the Huffman codebook (as this part can be done on the CPU). Note that the decompression pipeline is highly symmetrical to the compression pipeline, exhibiting throughput nearly identical to that of compression. Due to space constraints, we do not present a detailed evaluation.

Comparison with cuSZ. On A100, we observe that FZ-GPU has higher throughput than cuSZ on all datasets. FZ-GPU has an average speedup of 4.2× than cuSZ overall. on the CESM dataset,

FZ-GPU achieves an even higher average speedup of 10.7×. The performance compared with cuSZ on CESM is better than other datasets because the Huffman codebook generating time in cuSZ is almost the same among all datasets. For datasets like CESM, the field size is smaller than others, and the throughput of Huffman codebook generation is relatively lower. The result of cuSZ-ncb also proves this; the ratio of cuSZ-ncb to FZ-GPU is around 0.5 on almost all datasets. In contrast, FZ-GPU is highly stable across different datasets because our bitshuffle and fast encode kernels have almost the same amount of operations for the same data size.

The result on A4000 also demonstrates that FZ-GPU has more stability and higher throughput on different datasets compared to cuSZ. Our throughputs are consistently around 70 GB/s, an average of 2.4× higher than cuSZ. However, it is unusual that cuSZ on A4000 has a higher throughput for the CESM dataset than on A100. This is due to the CESM dataset's small data size per field (i.e., 24.7 MB), which is sufficient for A4000 to warm up but not for A100. This phenomenon is less pronounced with datasets having larger field sizes than CESM.

Comparison with cuZFP. On A100, FZ-GPU has an average speedup of 2.3× than cuZFP. Furthermore, we observe that FZ-GPU achieves higher throughput on almost every experimental setting, except for the high error-bound cases on CESM and RTM. For example, on CESM with the error bound of 1e−2, our throughput is 125.0 GB/s, while the throughput of cuZFP is 197.6 GB/s. The reason is that cuZFP employs discrete cosine transform and bit truncation, which can be efficiently performed on the GPU by matrix operations, and achieves high efficiency when the data is super smooth (as aforementioned, RTM and CESM after pre-quantization have many zero values with a high error bound). But this advantage of cuZFP disappears quickly as the error bound becomes lower.

On A4000, FZ-GPU has an average speedup of 1.3× than cuZFP. However, we notice that the throughput of cuZFP maintains almost the same between A4000 and A100. This is because cuZFP is limited by GPU memory bandwidth rather than peak performance.

Comparison with MGARD-GPU. As mentioned in §4.3, MGARD-GPU cannot work correctly on 1D datasets due to memory issues. Moreover, although it can work on A100 in some error bounds, the compression throughput is unsteadily low (e.g., 0.018 GB/s on 1D HACC with 1e−2 error bound). With excluding the extreme cases, FZ-GPU averages 87.0× and 45.7× in throughputs, compared to MGARD-GPU. We also note that MGARD-GPU does not scale well from A4000 to A100. For example, its throughput on CESM with the relative error bound of 1e−2 is 0.62 GB/s on A100 and 0.67 GB/s on A4000, far less distinguishable than the hardware specifications. This demonstrates that MGARD-GPU does not respond well to different grades of modern GPUs.

Comparison with cuSZx. On A100, cuSZx has higher compression throughput on all datasets. The compression throughput of cuSZx is 1.5× higher than FZ-GPU in average. This is because cuSZx uses a straightforward compression pipeline, which divides the input data into blocks and handles the constant blocks and non-constant blocks separately. This makes cuSZx highly efficient but also results in a fairly low compression ratio (as illustrated in §4.3). Note that the throughput of cuSZx on QMCPACK is relatively lower compared with other datasets. This is because the QMCPACK dataset consists of many unsmooth floating data points. Thus, non-constant blocks are much more than constant blocks. In contrast, FZ-GPU is highly stable over different datasets. On A4000, cuSZx also has advantages in throughput; the improvement is the same as that on A100, which is 1.5× on average.

Comparison with the CPU implementation. We also implement our proposed lossy compression algorithm on multi-core CPUs using OpenMP (called "FZ-OMP") and compare it with FZ-GPU. The evaluation results show that FZ-GPU with A100 has speedups of 38.8×, 42.4×, 36.3×, 31.8×, 34.8×, and 37.6× over FZ-OMP with Intel Xeon Gold 6238R CPUs (32 cores/threads) on HACC, CESM, Nyx, Hurricane, QMCPACK, and RTM, respectively. Moreover, FZ-OMP has higher throughput than SZ-OMP due to its efficient compression algorithm. For example, the average throughput of FZ-OMP is 1.7×, 2.5×, and 2.0× higher than that of the original SZ-OMP (v.2.1.12.5) [58] on the 3D Hurricane, Nyx, and RTM datasets, respectively, with the Intel CPUs using 32 cores/threads [5] (SZ-OMP only supports 3D data). This demonstrates that both our proposed GPU performance optimizations and our compression algorithm contribute to the significant performance improvement over SZ/cuSZ.

4.5 Evaluation of Proposed Optimizations

Finally, we present the evaluation of each of our optimizations in detail. The breakdown of the performance improvement on A100 is illustrated in Figure 10. Different bars represent different versions of each compression kernel, detailed as follows:

1. **pred-quant-v1**: The original dual-quantization kernel.
2. **pred-quant-v2**: Our optimized dual-quantization kernel without shifting and outlier handling.
3. **bitshuffle-mark-v1**: Two separate kernels for bitshfuffle and mark operations.
4. **bitshuffle-mark-v2**: One fused kernel for both bitshfuffle and mark operations.
5. **prefix-sum-encode-v1**: Our prefix-sum and fast encode kernel.
6. **prefix-sum-encode-v2**: The same kernel as v1, while the encoding is improved due to dual-quantization optimization.

Figure 10 shows that the dual-quantization kernel has a speedup of up to 1.7× because we remove the branches in the original kernel. More specifically, GPU executes instructions at the warp level, and different branches incur warp divergence, which is resolved sequentially. The kernel fusion of bitshuffle and bit-flag array also brings a speedup of up to 1.1×. The kernel fusion can avoid extra access to the global memory by directly caching the bitshuffled data in shared memory. Our prefix-sum-encode kernel also has a speedup of up to 1.9× because of dual-quantization optimization. This is because fewer data blocks are encoded, so the encoding time is much lower than before. In addition, on the HACC dataset, the encoding time differs from other datasets (i.e., v1 has higher throughput than v2). This is because Lorenzo prediction is less effective for unsmoothed data like HACC; it generates many large irregular integers, affecting the encoding performance. In contrast, cuSZ does not have this phenomenon as it otherwise handles these irregular integers as outliers.

[5]Note that the performance of both FZ-OMP and SZ-OMP increases as the number of threads increases to 32 (with up to 21.2× speedup), but it does not increase much with more than 32 threads on some datasets due to the limited workload per core.

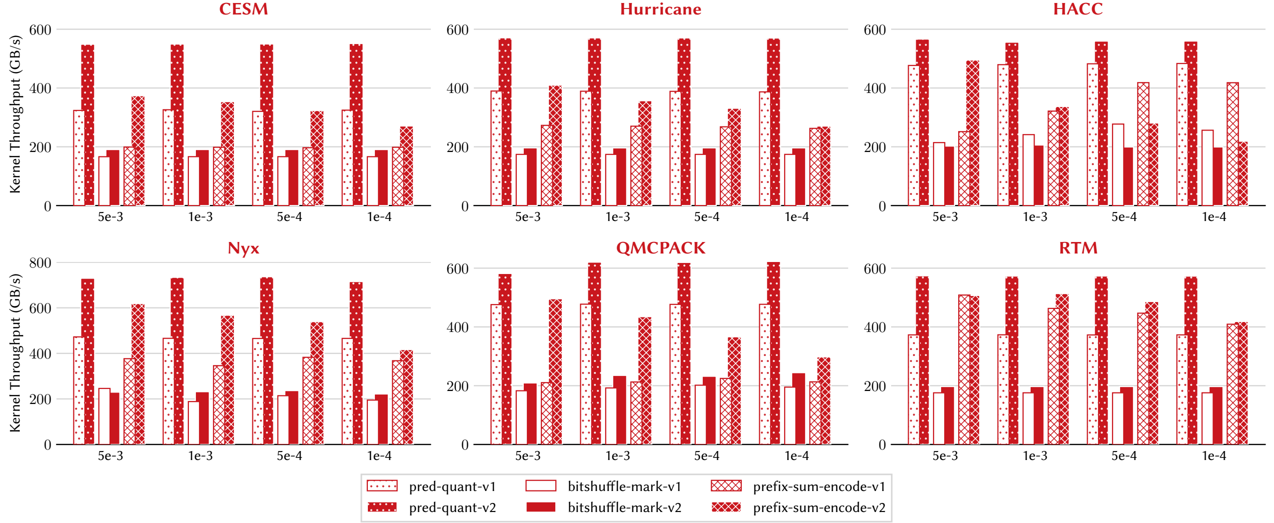

Figure 10: Performance improvements of our proposed optimizations for different compression kernels on NVIDIA A100.

GPU-CPU Data Transfer Throughput in GB/s for Range-Based Relative Error Bounds

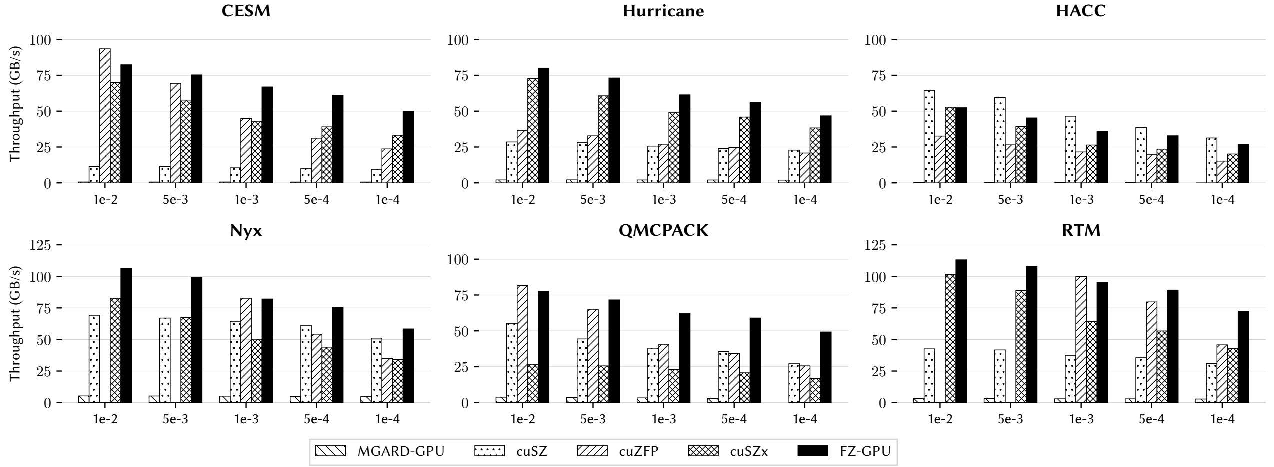

Figure 11: Overall CPU-GPU data-transfer throughput of cuZFP, cuSZ, cuSZx, MGARD-GPU, and FZ-GPU on NVIDIA A100.

4.6 Evaluation of Overall Throughput

Besides compression throughput (T_{compr}), overall throughput considering the time in moving compressed data between GPU and CPU, is also a critical metric for overall application performance. Thus, we propose to use this metric to further evaluate the efficiency of different compressors in practice. Specifically, the overall throughput can be calculated as

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

where BW is the memory bandwidth between GPU and CPU and CR is the compression ratio. 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 leverage up to 16-lane bandwidth (i.e., 32 GB/s). Based on our benchmarking result using [59], when the 4 GPUs read/write data from/to the CPU simultaneously, the bandwidth for each GPU can be as low as 11.4 GB/s (aggregately about 45 GB/s). Finally, we measure the overall data-transfer throughputs of different compressors and show them in Figure 11. It illustrates that FZ-GPU achieves the best overall throughput on almost all datasets at all evaluated relative error bounds. Note that for the interconnections with effective bandwidth lower than 15 GB/s (e.g., networks), FZ-GPU method can achieve the optimal balance of compression ratio and throughput. We leave the evaluation in node communication for future work.

4.7 Evaluation of Reconstructed Data Quality

Finally, we use Figure 12 to demonstrate the reconstructed data quality for all five lossy compressors by utilizing PSNR and SSIM. We select a similar compression ratio at approximately 22.8× for a fair comparison, with different error bounds or bitrate configured.

Specifically, FZ-GPU has the identical reconstructed data quality to that of cuSZ because of the shared error control scheme in our

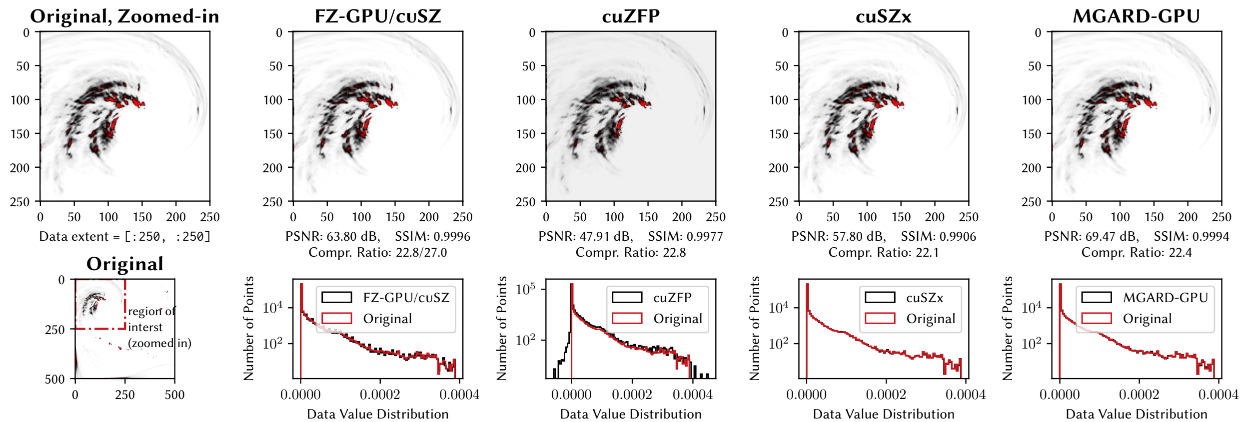

Figure 12: Reconstructed data quality using various GPU-based lossy compressors on field QSNOWf48 (slice 50) in the Hurricane dataset, under a similar compression ratio. The first row shows the visualization of the region of interest, while the second row shows the data distribution comparison between the decompressed and the original data for each compressor.

pipeline; thus, they share the same data visualization. Moreover, FZ-GPU has the highest SSIM among all compressors. Given that SSIM is a metric designed based on image structure, contrast, and luminance [60], this demonstrates that FZ-GPU has a higher capability to preserve the quantity of interests or features than other compressors. On the other hand, the PSNR of our proposed pipeline is much higher compared to cuZFP and cuSZx under a similar compression ratio. Although the PSNR of FZ-GPU is slightly lower than MGARD-GPU, MGARD-GPU has a very low throughput (4.9 GB/s compared to 65.4 GB/s in FZ-GPU). This is because MGARD-GPU uses a multi-grid-based approach with high time complexity (a large coefficient before $O(N)$) to achieve an accurate approximation.

5 RELATED WORK

Some GPU-based lossy compression works have been optimized for scientific data, with a focus on CUDA architectures [28]. For example, cuSZ is the first lossy compression framework that provides the error-bounded mode (detailed in §2.2). cuZFP is the CUDA implementation of ZFP algorithm [10], which performs near orthogonal transform and bit truncation over the split blocks of the data. Tian et al. [32] proposed using run-length encoding in place of Huffman encoding to improve the compression ratio of cuSZ for high error-bound scenarios. Yu et al. proposed cuSZx [55] based on the cuSZ framework that achieves very high compression throughput by using lightweight bitwise operations. Chen et al. developed MGARD-GPU [16] that optimizes data refactoring kernels for GPU accelerators to enable efficient creation and manipulation of data in multigrid-based hierarchical forms. Bitcomp [56] is a proprietary lossy compression developed by NVIDIA for scientific data, which has a similar performance as cuSZx.

In addition to lossy compression on GPUs, there are some GPU-based lossless compression works for scientific data. For example, Tian et al. [47] proposed and implemented an efficient Huffman encoding approach for modern GPU architectures to parallelize Huffman encoding algorithm and utilize the GPU's high memory bandwidth. Rivera et al. [48] bitwise a deep architectural optimization for two Huffman decoding algorithms to take advantage of CUDA GPU architectures. Knorr et al. [61] proposed an efficient GPU lossless compression of scientific floating-point data on GPUs using integer Lorenzo transform and vertical bit packing. Masui et al. [22] proposed a CPU vectorized compression using bitshuffle and LZ4, and [36] is its simple GPU implementation.

6 CONCLUSION AND FUTURE WORK

In this paper, we develop a fast and high-ratio error-bounded lossy compressor on GPUs for scientific data. Specifically, we design a new compression pipeline that consists of dual-quantization, bitshuffle, and fast lossless encoding. We also propose a series of architectural optimizations for each GPU compression kernel, including warp-level optimization for bitwise operations, maximization of shared memory utilization, and multi-kernel fusion. Finally, we evaluate our proposed FZ-GPU on six representative scientific datasets and demonstrate its high compression throughput and ratio.

In the future, we plan to (1) exploit fusing all GPU kernels into one to improve the performance further, (2) adapt FZ-GPU to other GPU platforms by using code translation tools such as HIPFY [62] for AMD GPUs and SYCLomatic [63] for Intel GPUs, and (3) evaluate FZ-GPU with real-world applications requiring fast compression, such as memory compression.

ACKNOWLEDGMENT

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. This work was also supported by the National Science Foundation under Grants OAC-2003709, OAC-2104023, OAC-2303064, OAC-2247080, and OAC-2312673. This research was also supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute.

REFERENCES

[1] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, V. Vishwanath, T. Peterka, J. Insley, *et al.*, "HACC: Extreme scaling and performance across diverse architectures," *Communications of the ACM*, vol. 60, no. 1, pp. 97–104, 2016.

[2] S. C. V. Vishwanath and K. Harms, *Parallel i/o on mira*, https://www.alcf.anl.gov/files/Parallel_IO_on_Mira_0.pdf, Online, 2019.

[3] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *2018 IEEE International Conference on Big Data*, IEEE, 2018, pp. 438–447.

[4] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, "An efficient transformation scheme for lossy data compression with point-wise relative error bound," in *IEEE International Conference on Cluster Computing*, Belfast, UK: IEEE, 2018, pp. 179–189.

[5] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in HPC storage systems," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, USA: IEEE, 2012, p. 7.

[6] S. W. Son, Z. Chen, W. Hendrix, A. Agrawal, W.-k. Liao, and A. Choudhary, "Data compression for the exascale computing era-survey," *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, pp. 76–88, 2014.

[7] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use cases of lossy compression for floating-point data in scientific data sets," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, 2019.

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

[9] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium*, Orlando, FL, USA: IEEE, 2017, pp. 1129–1139.

[10] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[11] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappello, "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors," *IEEE Transactions on Big Data*, pp. 1–14, 2022.

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

[13] https://lcls.slac.stanford.edu/lasers/lcls-ii, Online.

[14] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, *et al.*, "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*, 2020, pp. 3–15.

[15] cuZFP, https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp, Online, 2019.

[16] J. Chen, L. Wan, X. Liang, B. Whitney, Q. Liu, D. Pugmire, N. Thompson, J. Y. Choi, M. Wolf, T. Munson, I. Foster, and S. Klasky, "Accelerating multigrid-based hierarchical scientific data refactoring on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2021, pp. 859–868.

[17] Compression Modes, https://zfp.readthedocs.io/en/release0.5.4/modes.html.

[18] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. Ahrens, "Understanding GPU-based lossy compression for extreme-scale cosmological simulations," in *2020 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2020, pp. 105–115.

[19] P. Deutsch, *Rfc1951: Deflate compressed data format specification version 1.3*, 1996.

[20] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[21] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[22] K. Masui, "Bitshuffle: Filter for improving compression of typed binary data," *Astrophysics Source Code Library*, ascl–1712, 2017.

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

[24] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[25] D. Le Gall, "Mpeg: A video compression standard for multimedia applications," *Communications of the ACM*, vol. 34, no. 4, pp. 46–58, 1991.

[26] M Ainsworth, O Tugluk, B Whitney, and S Klasky, "Mgard: A multilevel technique for compression of floating-point data," in *DRBSD-2 Workshop at Supercomputing*, 2017.

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

[28] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

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

[30] L. P. Deutsch, *GZIP file format specification version 4.3*, 1996.

[31] Zstd, https://github.com/facebook/zstd/releases, Online, 2019.

[32] J. Tian, S. Di, X. Yu, C. Rivera, K. Zhao, S. Jin, Y. Feng, X. Liang, D. Tao, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data on gpus," in *2021 IEEE International Conference on Cluster Computing*, IEEE, 2021, pp. 283–293.

[33] S. Jin, C. Zhang, X. Jiang, Y. Feng, H. Guan, G. Li, S. L. Song, and D. Tao, "Comet: A novel memory-efficient deep learning training framework by using error-bounded lossy compression," *Proceedings of the VLDB Endowment*, vol. 15, no. 4, pp. 886–899, 2021.

[34] Q Zhou, C Chu, N. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda, "Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters," in *2021 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2021, pp. 444–453.

[35] K. Y. Besedin, P. S. Kostenetskiy, and S. O. Prikazchikov, "Increasing efficiency of data transfer between main memory and intel xeon phi coprocessor or nvidia gpus with data compression," in *International Conference on Parallel Computing Technologies*, Springer, 2015, pp. 319–323.

[36] Jon Wright, *Bslz4 decoding*, https://github.com/jonwright/bslz4decoders, Online, 2022.

[37] *Nvcomp*, https://github.com/NVIDIA/nvcomp.

[38] M Harris and K Perelygin, *Cooperative groups: Flexible cuda thread programming*, https://developer.nvidia.com/blog/cooperative-groups/, Oct. 2017.

[39] *Nvidia/cub: Cooperative primitives for cuda c++*. https://github.com/NVIDIA/cub.

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

[41] Hurricane ISABEL Simulation Data, http://vis.computer.org/vis2004contest/data.html, Online, 2019.

[42] NYX simulation, https://amrex-astro.github.io/Nyx/, Online.

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

[44] S. Jin, S. Di, J. Tian, S. Byna, D. Tao, and F. Cappello, "Improving prediction-based lossy compression dramatically via ratio-quality modeling," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, IEEE, 2022, pp. 2494–2507.

[45] J. Wang, T. Liu, Q. Liu, X. He, H. Luo, and W. He, "Compression ratio modeling and estimation across error bounds for lossy compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 7, pp. 1621–1635, 2019.

[46] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu, *et al.*, "Understanding and modeling lossy compression schemes on hpc scientific data," in *2018 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2018, pp. 348–357.

[47] J. Tian, C. Rivera, S. Di, J. Chen, X. Liang, D. Tao, and F. Cappello, "Revisiting huffman coding: Toward extreme performance on modern gpu architectures," in *2021 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2021, pp. 881–891.

[48] C. Rivera, S. Di, J. Tian, X. Yu, D. Tao, and F. Cappello, "Optimizing huffman decoding for error-bounded lossy compression on gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2022, pp. 717–727.

[49] T. Liu, J. Wang, Q. Liu, S. Alibhai, T. Lu, and X. He, "High-ratio lossy compression: Exploring the autoencoder to compress scientific data," *IEEE Transactions on Big Data*, 2021.

[50] R. Underwood, S. Di, J. C. Calhoun, and F. Cappello, "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, 2020, pp. 567–577.

[51] M. Barrow, Z. Wu, S. Lloyd, M. Gokhale, H. Patel, and P. Lindstrom, "Zhw: A numerical codec for big data scientific computation," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2022, pp. 1–9.

[52] R. Underwood, J. C. Calhoun, S. Di, A. Apon, and F. Cappello, "Optzconfig: Efficient parallel optimization of lossy compression configuration," *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[53] J. Liu, S. Di, K. Zhao, X. Liang, Z. Chen, and F. Cappello, "Dynamic quality metric oriented error bounded lossy compression for scientific datasets," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society, 2022, pp. 892–906.

[54] S. Di, D. Tao, X. Liang, and F. Cappello, "Efficient lossy compression for scientific data based on pointwise relative error bound," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 331–345, 2018.

[55] X. Yu, S. Di, K. Zhao, J. Tian, D. Tao, X. Liang, and F. Cappello, "Ultrafast error-bounded lossy compression for scientific datasets," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22, Association for Computing Machinery, 2022, 159–171.

[56] *Nvcomp*, https://github.com/NVIDIA/nvcomp, 2022.

[57] J. Nilsson and T. Akenine-Möller, "Understanding ssim," *arXiv preprint arXiv:2006.13846*, 2020.

[58] *Sz parallel mode with openmp*, https://github.com/szcompressor/SZ/blob/master/sz/src/sz_omp.c.

[59] *Benchmark of measuring bandwidth of multiple gpu*, https://github.com/enfiskutensykkel/multi-gpu-bwtest.

[60] D. R. I. M. Setiadi, "Psnr vs ssim: Imperceptibility quality assessment for image steganography," *Multimedia Tools and Applications*, vol. 80, no. 6, pp. 8423–8444, 2021.

[61] F. Knorr, P. Thoman, and T. Fahringer, "Ndzip-gpu: Efficient lossless compression of scientific floating-point data on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[62] *Hipify*, https://github.com/ROCm-Developer-Tools/HIPIFY.

[63] *Syclomatic*, https://github.com/oneapi-src/SYCLomatic.