cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data

Jiannan Tian
Washington State University
jiannan.tian@wsu.edu

Sheng Di
Argonne National Laboratory
sdi1@anl.gov

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

Cody Rivera
The University of Alabama
cjrivera1@crimson.ua.edu

Megan Hickman Fulp
Clemson University
mlhickm@g.clemson.edu

Robert Underwood
Clemson University
robertu@g.clemson.edu

Sian Jin
Washington State University
sian.jin@wsu.edu

Xin Liang
Oak Ridge National Laboratory
liangx@ornl.gov

Jon Calhoun
Clemson University
jonccal@clemson.edu

Dingwen Tao*
Washington State University
dingwen.tao@wsu.edu

Franck Cappello
Argonne National Laboratory
cappello@mcs.anl.gov

ABSTRACT

Error-bounded lossy compression is a state-of-the-art data reduction technique for HPC applications because it not only significantly reduces storage overhead but also can retain high fidelity for postanalysis. Because supercomputers and HPC applications are becoming heterogeneous using accelerator-based architectures, in particular GPUs, several development teams have recently released GPU versions of their lossy compressors. However, existing state-of-the-art GPU-based lossy compressors suffer from either low compression and decompression throughput or low compression quality. In this paper, we present an optimized GPU version, cuSZ, for one of the best error-bounded lossy compressors—SZ. To the best of our knowledge, cuSZ is the FIRST ERROR-BOUNDED lossy compressor on GPUs for scientific data. Our contributions are fourfold. (1) We propose a DUAL-QUANTIZATION scheme to entirely remove the data dependency in the prediction step of SZ such that this step can be performed very efficiently on GPUs. (2) We develop an efficient customized Huffman coding for the SZ compressor on GPUs. (3) We implement cuSZ using CUDA and optimize its performance by improving the utilization of GPU memory bandwidth. (4) We evaluate our cuSZ on five real-world HPC application datasets from the Scientific Data Reduction Benchmarks and compare it with other state-of-the-art methods on both CPUs and GPUs. Experiments show that our cuSZ improves SZ's compression throughput by up to 370.1× and 13.1×, respectively, over the production version running on single and multiple CPU cores, respectively, while getting the same quality of reconstructed data. It also improves the compression ratio by up to 3.48× on the tested data compared with another state-of-the-art GPU supported lossy compressor.

*Corresponding author: Dingwen Tao (dingwen.tao@wsu.edu), School of EECS, Washington State University, Pullman, WA 99164, USA.

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10.
https://doi.org/10.1145/3410463.3414624

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

Lossy Compression; Scientific Data; GPU; CUDA; Performance

ACM Reference Format:
Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3410463.3414624

1 INTRODUCTION

Large-scale high-performance computing (HPC) applications can generate extremely large volumes of scientific data. For instance, the Hardware/Hybrid Accelerated Cosmology Code (HACC) [1] can simulate 1~10 trillion particles in one simulation and produce up to 220 TB of data per snapshot, bringing up a total of 22 PB of data during the simulation [2] with only one hundred timesteps/snapshots. Such a large volume of data is imposing an unprecedented burden on supercomputer storage and interconnects [3] for both storing data to persistent storage and loading data for postanalysis and visualization. Therefore, data reduction has attracted great attention from HPC application users for reducing the volumes of data to be moved to/from storage systems. The common approaches are simply decimating snapshots periodically and adopting an interpolation for data reconstruction. However, such approaches result in a significant loss of valuable information for postanalysis [4]. Traditional data deduplication and lossless compression have also been used for shrinking data size but suffer from very limited reduction ratios on HPC floating-point datasets. Specifically, deduplication generally reduces the size of scientific datasets by only 20% to 30% [5], and lossless compression achieves a reduction ratio of up to about 2:1 in general [6]. This is far from scientists' desired compression ratios, which are around 10:1 or higher (such as Community Earth Simulation Model (CESM) [7]).

Error-bounded lossy compression has been proposed to significantly reduce data size while ensuring acceptable data distortion for users [8]. SZ [8, 9] is a state-of-the-art error-bounded lossy

compression framework for scientific data (to be detailed in §2), which often offers higher compression qualities (or better rate distortions) than other state-of-the-art techniques [3]. However, as illustrated in prior work [8, 9], SZ suffers from low compression and decompression throughput, which is only tens to hundreds of megabytes per second on a single CPU core. This throughput is far from enough for extreme-scale applications or advanced instruments with extremely high data acquisition rates, which is a major concern for corresponding users. The LCLS-II laser [10], for instance, may produce data at a rate of 250 GB/s [11], such that corresponding researchers require an extremely fast compression solution that can still have relatively high compression ratios—for example, 10:1—with preserved data accuracy. In order to match such a high data production rate, leveraging multiple graphics processing units (GPUs) is a fairly attractive solution because of its massive single-instruction multiple-thread (SIMT) mechanism and its high programmability as opposed to FPGAs or ASICs [12]. Moreover, the SZ algorithm follows $O(n)$ time complexity and employs large amounts of read and write operations in the memory, and hence its performance is eventually bounded by memory bandwidth. State-of-the-art GPUs cannot only provide high computation capability but also provide high memory bandwidth. For example, NVIDIA V100 GPU can provide at least one higher order magnitude of memory bandwidth than state-of-the-art CPUs can [13].

SZ, however, cannot be run on GPUs efficiently because of the lack of parallelism in its design. The main challenges are twofold: (1) the tight dependency in the prediction-quantization step of the SZ algorithm incurs expensive synchronizations across iterations in a GPU implementation; and (2) during the customized Huffman coding step of the SZ algorithm, coding and decoding each symbol based on the constructed Huffman tree involve many different branches (see §2 for more details). This process causes serious warp divergence and random memory access issues, which inevitably lead to low GPU memory bandwidth utilization and performance.

To solve these issues, this paper presents an optimized GPU version of the SZ algorithm, called cuSZ, and proposes a series of optimization techniques for cuSZ to achieve high compression and decompression throughputs on GPUs. Specifically, we focus on the main performance bottlenecks (Lorenzo prediction [14] and customized Huffman coding [8]) and improve their performance for GPUs. We propose a novel technique called dual-quantization that can be applied to any prediction-based compression algorithms to alleviate the tight dependency in its prediction step. Moreover, according to prior work [11], a strict error-controlling scheme of lossy compression is needed by many HPC applications for their scientific explorations and postanalyses. However, the state-of-the-art GPU-based lossy compressors such as cuZFP [15] are not error-bounded. To the best of our knowledge, cuSZ [1] is the first strictly error-bounded lossy compressor on GPU for scientific data. Our contributions are summarized as follows.

- We propose a generic dual-quantization scheme to entirely remove the data dependencies in the prediction-quantization step of lossy compression and apply it to Lorenzo predictor in SZ algorithm.
- We develop an efficient customized Huffman coding for SZ on GPUs with fine- and coarse-grained parallelism.
- We carefully implement cuSZ and optimize its performance on CUDA architecture. In particular, we fine-tune the chunk size in Huffman coding and develop an adaptive method that selects 32-bit or 64-bit representation dynamically for Huffman code and can significantly improve GPU memory bandwidth utilization.
- We evaluate our proposed cuSZ on five real-world HPC application datasets provided by a public repository, *Scientific Data Reduction Benchmarks* [16], and compare it with other state-of-the-art methods on both CPUs and GPUs. Experiments show that the cuSZ can significantly improve both compression throughput by up to 370.1× and 13.1× over the production version of SZ running on single CPU core and multiple CPU cores, respectively. cuSZ has up to 3.48× higher compression ratio than another advanced GPU supported lossy compressor with reasonable data distortion.

The rest of the paper is organized as follows. In §2, we discuss the SZ lossy compression in detail. In §3, we propose our novel optimizations for the GPU version of SZ and implement it using CUDA. In §4, we present the evaluation results based on five real-world simulation datasets from the Scientific Data Reduction Benchmarks and compare cuSZ with other state-of-the-art compressors on both CPU and GPU. In §5, we discuss related work. In §6, we present our conclusions and discuss our future work.

2 SZ BACKGROUND

Many scientific applications require a strict error-bounded control when using lossy compression to achieve accurate postanalysis and visualization for scientific discovery, as well as a high compression ratio. SZ [8, 9] is a prediction-based lossy compression framework designed for scientific data that strictly controls the global upper bound of compression error. Given a user-set error bound eb, SZ guarantees $|d - d^\bullet| < eb$, where d and $d^\bullet$ are the original value and the decompressed value, respectively. SZ's algorithm involves five key steps: preprocessing, data prediction, linear-scaling quantization, customized variable-length encoding, and optional lossless compression, e.g., `gzip` [17] and `Zstd` [18].

1) **Preprocessing** SZ performs a preprocessing step, such as linearization in version 1.0 or a logarithmic transform for the pointwise relative error bound in version 2.0 [19].
2) **Data Prediction** SZ predicts the value of each data point by a data-fitting predictor, e.g., a *Lorenzo* predictor [14] (abbreviated as ℓ-predictor) based on its neighboring values. In order to guarantee that the compression error is always within the user-set error bound, the predicted values must be exactly the same in between the compression procedure and decompression procedure. To this end, the neighbor values used in the prediction have to be the decompressed values instead of the original values.
3) **Linear-Scaling Quantization** SZ computes the difference between the predicted value and original value for each data point and performs a linear-scaling quantization [8] to convert the difference to an integer based on the user-set error bound.

[1] The code is available at https://github.com/hipdac-lab/cuSZ.

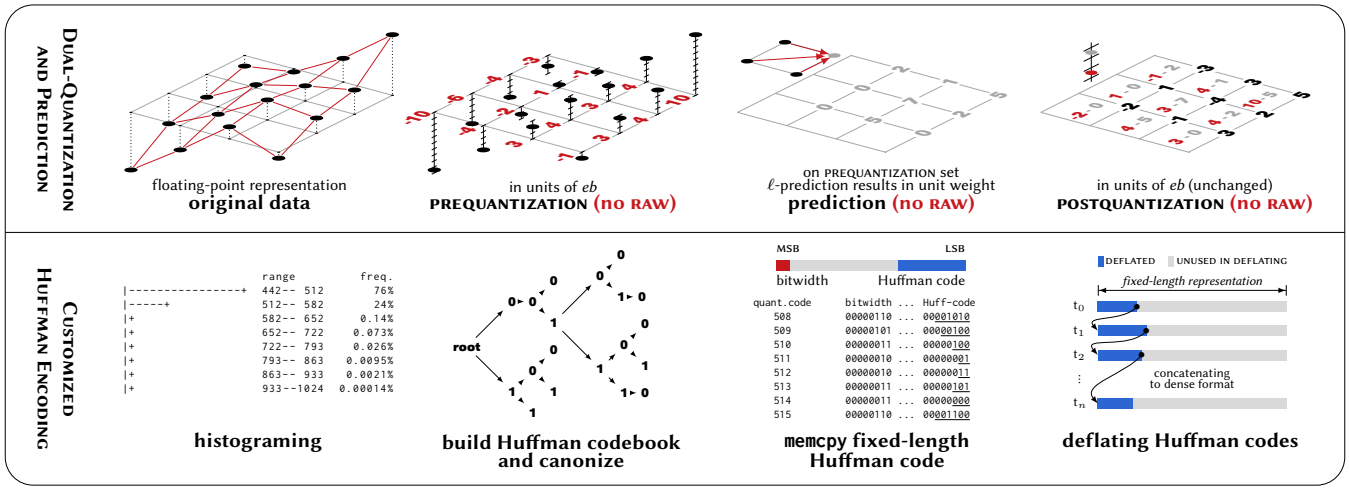

Figure 1: The system overview of cuSZ. The top 4 figures illustrate a dual-quant example, which has no loop-carried RAW. The bottom 4 figures correspond to the four subprocedures of our customized Huffman coding described in §3.2.

4) **Customized Variable-Length Coding** SZ adopts a customized Huffman coding to reduce the data size significantly, because the integer codes generated by the linear-scaling quantization are likely distributed unevenly, especially when the data are mostly predicted accurately.
5) **Lossless Compression** The last step optionally further compresses the encoded data by a lossless compressor such as `Zstd` [18], which may significantly reduce the size due to potential repeated patterns in the bitstream.

In this work, we focus mainly on the SZ compressor, because much prior work [3, 20, 21, 22, 23, 24, 25, 26] has verified that SZ yields the best compression quality among all the prediction-based compressors. However, it is nontrivial to port SZ on GPUs because of the strict constraints in its compression design. For instance, the data used in the prediction must be updated by decompressed values, such that the data prediction in the SZ compressor [8, 9] needs to be performed one by one in a sequential order. This requirement introduces a loop-carried *read-after-write* (RAW) dependency during the compression (will be discussed in §3.1.2), making SZ hard to parallelize.

We mainly focus on SZ-1.4 instead of SZ-2.0 because the 2.0 model is particularly designed for low-precision use cases with visualization goals, in which the compression ratio can reach up to several hundred while the reconstructed data often have large data distortions. Recent studies [11], however, demonstrate that scientists often require a relatively high precision (or low error bound) for their sophisticated postanalysis beyond visualization purposes. In this situation (with relatively low error bounds), SZ-2.0 has very similar (or even slightly worse, if not for all the cases) compression qualities to those of SZ-1.4, as demonstrated in [3]. Accordingly, our design for the GPU-accelerated SZ lossy compression is based on SZ-1.4 and takes advantage of both algorithmic and GPU hardware characteristics. Moreover, the current CPU version of SZ does not support SIMD vectorization and has no specific improvement on the arithmetic performance. Therefore, the CPU baseline used in our following evaluation is based on the nonvectorized single-core and multicore implementation.

3 DESIGN METHODOLOGY OF CUSZ

In this section, we propose our novel lossy compression design, cuSZ, for CUDA architectures based on the SZ model. A system overview of our proposed cuSZ is shown in Figure 1. We develop different coarse- and fine-grained parallelization techniques to each subprocedure in compression and decompression. Specifically, we first employ a data-chunking technique to exploit coarse-grained data parallelism. The chunking technique is used throughout the whole cuSZ design, including lossless (step 2 and 3) and lossy (step 1, 4, and 5) procedures in both compression and decompression. We then deeply analyze the RAW data dependency in SZ and propose a novel two-phase prediction-quantization approach, namely, dual-quantization, which totally eliminates the data dependency in the prediction-quantization (abbreviated as predict-quant) step. Furthermore, we provide an in-depth breakdown analysis of Huffman coding and develop an efficient Huffman coding on GPUs with multiple optimizations. A summary of our parallelization techniques is shown in Table 1.

	sequential	coarse-grained	fine-grained	atomic
compression				
DUAL-QUANTIZATION			•	
histogram			•	•
build Huffman tree	•			
canonize codebook	•		•	•
Huffman encode (fix-length)			•	
deflate (fix- to variable-length)		•		
decompression				
inflate (Huffman decode)		•		
reversed DUAL-QUANTIZATION		•		

Table 1: Parallelism implemented for cuSZ's subprocedures (kernels) in compression and decompression.

3.1 Parallelizing Prediction-Quantization in Compression

In this section, we discuss our proposed optimization techniques to parallelize SZ's predict-quant procedure on GPU architectures. We first chunk the original data to gain coarse-grained parallelization,

and then we assign a thread to each data point for fine-grained in-chunk parallel computations.

3.1.1 Chunking and Padding. Figure 2 illustrates our chunking and padding technique. For each chunked data block, we assign a thread to each data point (i.e., fine-grained parallelism). To avoid complex modifications to the prediction function after chunking, we add a padding layer to each block in the predict-quant step. We set all the values in the padding layer to `0` such that they do not affect the predicted values of the points neighboring to the padding layer, as shown in Figure 2. We note that in the original SZ, the uppermost points and leftmost points (denoted by "outer layer", shaded in Figure 2) are saved as unpredictable data directly. In our chunking version, however, directly storing these points for each block would significantly degrade the compression ratio. Therefore, we apply ℓ-prediction to the outer layer instead, such that every point in the block is consistently processed based on the ℓ-predictor, avoiding thread/warp divergence. Moreover, we initialize the padding layer with `0`s; the prediction for each outer-layer point falls back to 1D 1-order Lorenzo, as shown in Figure 2. Based on our empirical result, we adopt 32 for 1D data, 16×16 for 2D data, and 8×8×8 for 3D data.

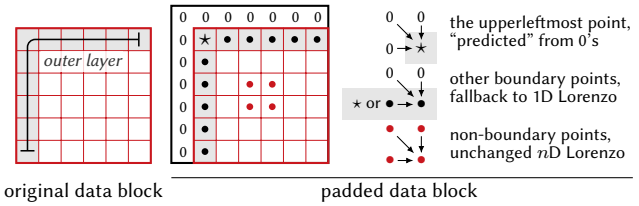

Figure 2: Data chunking and padding in cuSZ.

3.1.2 Dual-Quantization Scheme. In the following discussion, we use circle ∘ and bullet • to denote the compression and decompression procedure, respectively. We use star ⋆ to denote all the values related to the data reconstruction in compression. The subscript $(\cdot)_k$ represents the kth iteration.

Read-After-Write in SZ. In the original SZ algorithm, all data points need to go through predict-quant, and in situ reconstruction iteratively, which causes intrinsic *read-after-write* (RAW) dependencies (as illustrated in Figure 3).

We describe the loop-carried RAW dependency issue in detail below. For any data point at the $(k-1)$th iteration in SZ compression, given a predicted value p_{k-1}, the prediction error e°_{k-1} (i.e., $d_{k-1} - p^\circ_{k-1}$) is converted to an integer and a corresponding quantization code q_{k-1} based on the user set error bound eb. Then, the reconstructed prediction error $e^{\circ\star}_{k-1}$ and the reconstructed value $d^{\circ\star}_{k-1}$ are generated by using q_{k-1}, eb, and p°_{k-1}. After that, $d^{\circ\star}_{k-1}$ is written back to replace d_{k-1}. This procedure ensures that $d^{\circ\star}_{k-1}$ is equivalent to the reconstructed $d^\bullet_{k-1}$ during decompression (as shown in Figure 3); however, the kth iteration must wait until the update completes at the end of the $(k-1)$th iteration, which incurs loop-carried data dependency. Also note that $d^{\circ\star}$ is written back in the last step of the current iteration, and its written value is used at the beginning of the next iteration, therefore, the two consecutive iterations cannot overlap. Hence, under the original design of the predict-quant in SZ, it is infeasible to effectively exploit fine-grained parallelism and efficiently utilize SIMT on GPUs. We present the original SZ's predict-quant step in Algorithm 1 in detail.

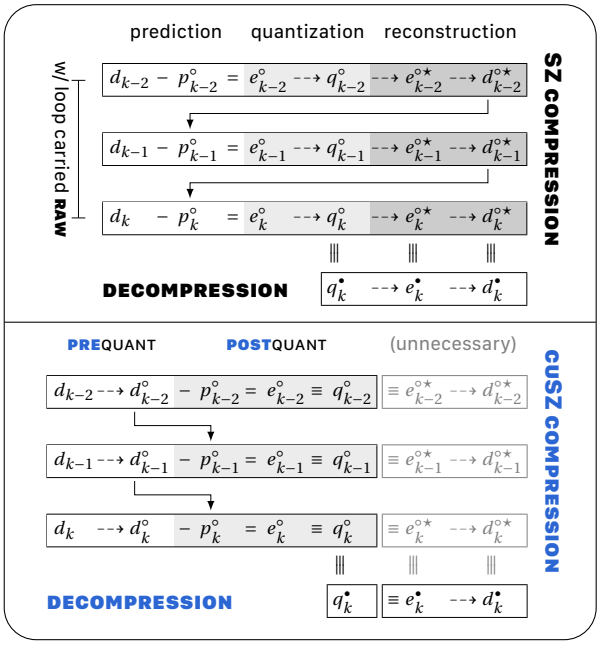

Figure 3: Diagram of original quantization (*top*) and dual-quantization (*bottom*) procedures. Arrow means data dependency.

Algorithm 1: Original SZ of predict-quant

```
for d ∈ D do                                              ▷ compression
    p° ← ℓ(d_SR), e° ← p° − d
    if e°/eb < cap (in-cap) then                          ▷ quantization
        e°_D ← integerize(e°/(2×eb))
        rehearsal ← p° + 2 · e°_D · eb
        watchdog(rehearsal − d < eb, fallback: outlier)
    else
        outlier: e°_D ← 0 and record the verbatim x ← d
    end if
    d ← rehearsal or x accordingly                        ▷ incurs RAW
end for

for d• ∈ D• to reconstruct cascadingly do                 ▷ decompression
    p• ← ℓ(d•_SR)
    d• ← p• + 2 · e°_D · eb if in-cap else verbatim x
end for
```

Proposed Dual-Quantization Approach. To eliminate the RAW dependency, we propose a dual-quantization scheme by modifying the data representation during the predict-quant procedure. Our dual-quantization (abbreviated as dual-quant) consists of two steps: prequantization and postquantization.

Given a dataset D with an arbitrary dimension, we first quantize it based on the user-set eb and convert it to a new dataset

$$D^\circ = \left\{ d^\circ \mid d^\circ = \text{round}\left\langle d/(2\times eb)\right\rangle,\ d \in D \right\},$$

where any $d^\circ \in D^\circ$ is strictly a multiple of $(2\times eb)$. We call this step prequantization (abbreviated as prequant). In order to avoid

overflow, d° is stored in floating-point data type. We note that the error introduced by PREQUANT (defined as POSTERROR) is strictly bounded by the user-set error bound, that is, $|d - 2 \cdot d^\circ \cdot eb| < eb$.

After the PREQUANTIZATION, we can calculate each predicted value based on its surrounding values (denoted by d°_{SR}) and the ℓ-predictor as

$$P^\circ = \{p^\circ \mid p^\circ = \ell(d^\circ_{\text{SR}}),\ d^\circ \in D^\circ\}.$$

The second step, called POSTQUANTIZATION (abbreviated as POSTQUANT), serves as the counterpart of the linear-scaling quantization in the original SZ. POSTQUANT computes the *difference* between the predicted value and the PREQUANT-ized value. Different from the original SZ, such difference does not cause any compression error (will be discussed later), we use δ instead of e to denote this difference:

$$\Delta^\circ = \{\delta^\circ \mid \delta^\circ = d^\circ - p^\circ,\ d^\circ \in D^\circ,\ p^\circ \in P^\circ\}.$$

Then, the quantization code q° is generated based on δ. Note that q° is quantitatively equivalent to δ°, represented differently: δ° is a floating-point number to avoid subnormal values (i.e. under/overflow), while q° is an integer, which is used in the subsequent lossless coding (e.g., Huffman coding). It is worth noting that, during decompression, d° can be reconstructed (as $d^\bullet$) based on losslessly decoded $q^\bullet \equiv q^\circ$ (hence exactly δ°) and predicted p°, thus this POSTQUANT step does not introduce any further error.

Eliminating RAW. In the following text, we explain in detail why the DUAL-QUANT method effectively eliminates the RAW dependency. Conceptually, similar to the original SZ, we can construct $\delta^{\circ\star}$ and $d^{\circ\star}$ during the compression, as shown in Figure 3. In fact, for $(k-1)$th iteration, $\delta^{\circ\star}_{k-1}$ is strictly equal to δ°_{k-1}, because casting quantization code q°_{k-1} to $\delta^{\circ\star}_{k-1}$ is a exact reversed procedure of casting δ°_{k-1} to q°_{k-1}.

Similarly, $d^{\circ\star}_{k-1}$ and d°_{k-1} are also strictly equivalent. Consequently, unlike the original SZ that must write $d^{\circ\star}_{k-1}$ back to update $d^{\circ\star}_{k-1}$ before the kth iteration, $d^{\circ\star}_{k-1} \equiv d^\circ_{k-1}$ always holds in our proposed DUAL-QUANT approach. As illustrated in Figure 3, after PREQUANT, all d° are dependency free for POSTQUANT. By eliminating the loop-carried RAW dependency (marked as arrows in Figure 3), we can effectively parallelize the DUAL-QUANT procedure by performing fine-grained (per-point) parallel computation, which is commonly seen in image processing [27]. We illustrate the detailed DUAL-QUANT procedure in Algorithm 2.

Lorenzo Predictor with Binomial Coefficients. According to Tao et al. [8], the generalized ℓ-predictor is given by

$$\sum_{0\le k_{1\ldots m}\le n}^{k_{1\ldots d}\neq \mathbf{0}} \left\langle \prod_{j=1}^{m} (-1)^{k_j+1} \binom{n}{k_j} \right\rangle \cdot d_{x_1-k_1,\cdots,x_d-k_d},$$

where $\sum_{0\le k_{1\ldots m}\le n}^{k_{1\ldots d}\neq \mathbf{0}} \left\langle \prod_{j=1}^{m} (-1)^{k_j+1} \binom{n}{k_j} \right\rangle = 1$ and $d \in D$. For example, 1D 1-order ℓ-predictor is $p^\circ_a = d^\circ_{a-1}$, and 2D 1-order ℓ-predictor is $p^\circ_{(a,b)} = \ell\left(d^\circ_{\text{sr}}\right) = d^\circ_{a-1,b} + d^\circ_{a,b-1} - d^\circ_{a-1,b-1}$, as illustrated in Figure 1. We note that all the coefficients in the formula of the ℓ-predictor are integers; thus, the prediction computation consists of mathematically integer-based operations (additions and multiplications) and results in unit weight. This ensures that no division is needed, and the data reconstruction based on DUAL-QUANT is fairly precise and robust with respect to machine ϵ, however, the original SZ using precise floating-point operations suffers from underflow. Note that the predicted values which are integers will be completely corrected by the saved quantization codes in decompression, so the final error is still bounded by eb.

Algorithm 2: cuSZ of DUAL-QUANT

```
1  for ∀d ∈ D concurrently do                          ▷ compression
2      d° ← d/(2×eb)                                   ▷ (FP representation) PREQUANT
3      d  ← d°                                         ▷ BARRIER
4      p° ← ℓ(d°_SR), δ° ← p° − d°
5      if δ° < CAP/2 (IN-CAP) then                     ▷ POSTQUANT
6          δ°_D ← CAST<FLOAT2INT>(δ°)
7      else
8          OUTLIER: δ°_D ← 0 and record the verbatim x ← d°
9      end if
10 end for

11 for d• ∈ D• to reconstruct cascadingly do           ▷ decompression
12     p• ← ℓ(d•_SR)
13     d• ← (p• + δ°_D) · (2×eb) if IN-CAP else verbatim x
14 end for
```

3.2 Efficient Customized Huffman Coding

To efficiently compress the quantization codes generated by DUAL-QUANT, we develop an efficient customized Huffman coding for SZ on GPUs. Specifically, Huffman coding consists of the following subprocedures: (1) calculate the statistical frequency for each quantization bin (as a symbol); (2) build the Huffman tree based on the frequencies and generate a base codebook along with each code bitwidth; (3) transform the base codebook to the canonical Huffman codebook (called canonization); (4) encode in parallel according to the codebook, and concatenate Huffman codes into a bitstream (called *deflating*). And Huffman decoding is composed of (1) retrieving the reverse codebook and (2) decoding accordingly.

Note that the fourth subprocedure of encoding can be further decomposed into two steps for fine-grained optimization. Codebook-based encoding is basically memory copy and can be fully parallelized in a fine granularity, whereas deflating can be performed only sequentially (except blockwise parallelization discussed in §3.1.1) because of its atomic operations. We discuss Huffman coding on GPUs step by step as follows.

3.2.1 Histogram for Quantization Bins. The first step of Huffman coding is to build a histogram representing the frequency of each quantization bin from the data prediction step. The GPU histogramming algorithm that we use is derived from the algorithm proposed by Gómez-Luna et al. [28]. This algorithm minimizes conflicts in updating the histogram bin locations by replicating the histogram for each thread block and storing the histogram in shared memory. Where possible, conflict is further reduced by replicating the histogram such that each block has access to multiple copies. All threads inside a block read a specified partition of the quantization codes and use atomic operations to update a specific replicated histogram. As each block finishes its portion of the predicted data, the replicated histograms are combined via a parallel reduction into a single global histogram, which is used to construct the final codebook in Huffman coding.

3.2.2 Constructing Huffman Codebook. In order to build the optimal Huffman tree, the local symbol frequencies need to be aggregated to generate the global symbol frequencies for the whole dataset. By utilizing the aggregated frequencies, we build a codebook according to the Huffman tree for encoding. Note that the number of symbols—namely, the number of quantization bins—is a limited number (generally no greater than 65,536) that is much smaller than the data size (generally, millions of data points or more). This leads to a much lower number of nodes in the Huffman tree compared with the data size, such that the time complexity of building a Huffman tree is considered low. We note that building Huffman tree sequentially on CPU benefits from high CPU-frequency and low memory-access latency. However, it requires CPU-to-GPU/GPU-to-CPU transfer of frequencies/codebook before/after building the tree, and communicating these two small messages would incur non-negligible overheads. Therefore, we adopt one GPU thread to build the Huffman tree sequentially to avoid such overheads.

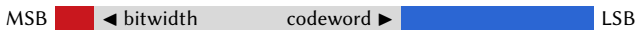

Figure 4: Fixed-length representation of Huffman codeword and its bitwidth.

We propose an adaptive codeword representation to enhance the utilization of memory bandwidth, which improves the Huffman encoding performance in turn. We illustrate the organization of the codebook in Figure 4. The codebook is organized by units of unsigned integers, and each contains a variable-length Huffman codeword from LSB (the rightmost or the least significant bits) and its bitwidth from MSB (the leftmost or the most significant bits). According to the pessimistic estimation of maximum bitwidth of optimal Huffman codeword [29], one is supposed to use `uint64_t` to hold each bitwidth-codeword representation. For example, the maximum bitwidth could be 33 bits for `CLDHGH` field from `CESM-ATM` dataset in the worst case. However, we note that using `uint32_t` to represent a bitwidth-codeword tuple can significantly improve the Huffman coding and decoding performance compared with using 64-bit unsigned integers (i.e., `uint64_t`), because of higher GPU memory bandwidth utilization. Thus, we propose to dynamically select `uint32_t` or `uint64_t` representation for the Huffman bitwidth-codeword based on the practical maximum bitwidth instead of pessimistic estimation. We show the performance evaluation with different representations in §4.

The theoretical time complexity is $O(k \log k)$ for building a Huffman tree and $O(k)$ for a traversing tree, where k is the number of symbols (quantization bins). Our experiments show that the real execution time of building a Huffman tree is consistent with the theoretical time complexity analysis (see Table 3). On the other hand, the number of symbols is determined by the smoothness of the dataset and the user-desired error bound (1,024 by default). For example, with a relatively large error bound such as the value-range-based relative error bound [2] of 10^{-3}, we observe that most of the symbols are concentratedly distributed near the central of codebook. As the error bound decreases, the symbols become more evenly distributed. Thus, determining a suitable number of quantization bins is important for high performance in constructing a codebook.

3.2.3 Canonizing Codebook. A canonical Huffman codebook [30] holds the same bitwidth of each codeword as the original Huffman codebook (i.e., base codebook), while its bijective mapping (between quantization code and Huffman codeword) and variable codeword make the memory layout organized more efficiently. The time complexity of sequentially canonizing codebook from the base is $O(k)$, where k is the number of symbols (i.e., the number of quantization bins) and is sufficiently small compared with the data size. By using a canonical codebook, we can (i) decode without the Huffman tree, (ii) efficiently cache the reverse codebook for high decoding throughput, and (iii) maintain exactly the same compression ratio as the base Huffman codebook.

The process of canonizing codebook can be decomposed into the following subprocedures: (1) linear scanning of the base codebook (sequentially $O(k)$), which is parallelized at fine granularity with atomic operations; (2) loosely radix-sorting of the codewords by bitwidth (sequentially $O(k)$), which cannot be parallelized because of the intrinsic RAW dependency; and (3) building the reverse codebook (sequentially $O(k)$), which is enabled with fine-grained parallelism.

It is intuitive to separate the functionalities of the aforementioned subprocedures and implement them into independent CUDA kernels with different configurations (i.e., `blockDim` and `gridDim`). Based on our profiling results on NVIDIA V100 GPU, however, launching a CUDA kernel usually takes about 60 microseconds (μs) (about 200 μs for the three kernels of canonization) measured by 11 kernels launched in total. Moreover, any two consecutive subprocedures require an additional expensive synchronization (i.e., `cudaDeviceSynchronize`). However, our experiment indicates that canonizing codebook is sufficiently fast ; thus, we integrate all the three subprocedures in one single kernel.

We note that this single kernel must be launched with more threads than the that is limited for a single thread block (i.e. 1024) for two reasons. On the one hand, a high scalability is required for the parallel reads/writes to match the problem size in subprocedures (1) and (3). On the other hand, unlike histogramming that saves only the $\Theta(k)$ frequencies in shared memory, this kernel requires saving both the codebook and its footprint, which may exceed the maximum allowable capacity of shared memory in a single thread block (e.g., 96 KB for a V100). Since shared memory is only visible to its designated thread block, shuffling codewords in shared memory across different thread blocks is semantically prohibitive. In addition, there is few intermediate data reuse, thus, we use global memory instead of shared memory to save the codebook. Hence, we employ the state-of-the-art CUDA API—Cooperative Groups [31]—to achieve in-grid operation. Specifically, we launch the same number of threads as the codeword in the base codebook. We select one thread to perform the RAW-restricted sequential subprocedure when needed (see Table 1). Note that it takes only 32 μs on a V100 to launch this Cooperative Groups enabled kernel, which significantly reduces the overhead compared to launching multiple kernels. Moreover, compared to two inter-kernel barriers with more than 2×60 μs, two in-grid barriers have relatively low overheads,

[2]Value-range-based relative error bound (denoted by `valrel`) is the error bound relative to the value range of the dataset.

and eventually result in, for instance, about 200 μs kernel time with $k = 1024$.

3.2.4 Encoding and Deflating. We design an efficient solution to perform the encoding by GPU threads in parallel. Encoding involves looking up a symbol in the codebook and performing a memory copy. After we adaptively select a 32-/64-bit unsigned integer to represent a Huffman code with its bitwidth, the encoding step is *massively* parallelized. To generate the *dense* bitstream of Huffman codes within each data block, we conduct *deflating* in order to concatenate the Huffman codes and remove the unnecessary zero bits according to the saved bitwidths.

Since the deflated code is organized sequentially, we apply the coarse-grained chunkwise parallelization technique discussed in §3.1.1. In particular, a data chunk for compression and decompression is mapped to one GPU thread. Note that the chunk size for deflating is not necessarily the same as the chunk size for dual-quant, and it does not rely on the dimensionality. We optimize the deflating chunk size by evaluating the performance with different sizes (will be showed in §4.2.1). We also employ memory reuse technique to reduce the GPU memory footprint in deflating. Specifically, we reuse the memory space of Huffman codes for the deflated bitstream because the latter uses significantly less memory space and does not have any conflict when writing the deflated bitstream to the designated location.

3.3 Decompression

cuSZ's decompression consists of two steps: Huffman decoding (or inflating the densely concatenated Huffman bitstream) and reversed dual-quant. In inflating, we first use the previously built reverse codebook to retrieve the quantization codes from the deflated Huffman bitstream. Then, based on the retrieved quantization codes, we reconstruct the floating-point data values. Note that only coarse-grained chunking can be applied to decompression, and its chunk size is determined in compression. The reason is that the two steps both have a RAW dependency issue. In fact, retrieving the variable-length codes has the same pattern as loop-carried RAW dependency. For the reversed dual-quantization procedure, each data point cannot be decompressed until its preceding values are fully reconstructed.

4 EXPERIMENTAL EVALUATION

In this section, we present our experimental setup (including platform, baselines, and datasets) and our evaluation results.

4.1 Experimental Setup

Evaluation Platform. We conduct our experimental evaluation using PantaRhei cluster [32]. We perform the experiments on an NVIDIA V100 GPU [13] from the cluster and compare with lossy compressors on two 20-core Intel Xeon Gold 6148 CPUs from the cluster. The GPU is connected to the host via 16-lane PCIe 3.0 interconnect. We use NVIDIA CUDA 9.2 and its default profiler to measure the kernel time.

Comparison Baselines. We compare our cuSZ with two baselines: SZ-1.4.13.5 and cuZFP [15]. For SZ-1.4, we adopt the default setting: 16 bits for linear-scaling quantization (i.e., 1,024 quantization bins),

DATASETS	TYPE	DATUM SIZE DIMENSIONS	#FIELDS EXAMPLE(S)
COSMOLOGY HACC	fp32	1,071.75 MB 280,953,867	6 in total x, vx
CLIMATE CESM-ATM	fp32	24.72 MB 1,800×3,600	79 in total CLDHGH, CLDLOW
CLIMATE Hurricane	fp32	95.37 MB 100×500×500	20 in total CLOUDf48, Uf48
COSMOLOGY Nyx	fp32	512.00 MB 512×512×512	6 in total baryon_density
QUANTUM QMCPACK	fp32	601.52 MB 288×115×69×69	2 formats in total einspline

Table 2: Real-world datasets used in evaluation.

#QUANT.	128	256	512	1024	2048	4096	8192
build tree	0.48	0.77	1.80	2.13	6.46	12.68	25.06
get codebook	0.20	1.14	2.36	2.69	7.09	14.43	25.65
total	0.68	2.16	4.16	4.81	13.55	27.10	50.71

Table 3: Breakdown time (in ms) of constructing a codebook, including building a Huffman tree and creating a codebook according to the tree based on the Hurricane Isabel dataset.

`best_compression` mode, and `best_speed` mode for `gzip`, which lead to a good tradeoff between compression ratio and performance.

Test Datasets. We conduct our evaluation and comparison based on five typical real-world HPC simulation datasets of each dimensionality from the Scientific Data Reduction Benchmarks suite [16]: ① 1D `HACC` cosmology particle simulation [1], ② 2D `CESM-ATM` climate simulation [33], ③ 3D `Hurricane ISABEL` simulation [34], ④ 3D `Nyx` cosmology simulation [35], and ⑤ 4D `QMCPACK` quantum Monte Carlo simulation [36]. They have been widely used in prior works [3, 11, 19, 37, 38] and are good representatives of production-level simulation datasets. Table 2 shows all 112 fields [3] across these datasets. The data sizes for the five datasets are 6.3 GB, 2.0 GB, 1.9 GB, 3.0 GB, and 1.2 GB, respectively. Note that our evaluated HACC dataset is consistent with real-world scenarios that generate petabytes of data. For example, according to [1], a typical large-scale HACC simulation for cosmological surveys runs on 16,384 nodes each with 128 million particles and generates 5 PB over the whole simulation. The simulation contains 100 individual snapshots of roughly 3 GB per node. We evaluate a single snapshot for each dataset instead of all the snapshots, because the compressibility of most of the snapshots usually has strong similarity. Moreover, when the field is too large to fit in a single GPU's memory, cuSZ divides it into blocks and then compresses them block by block.

4.2 Evaluation Results and Analysis

In this section, we evaluate the compression performance and quality of cuSZ and compare it with CPU-SZ and cuZFP.

4.2.1 Compression Performance. We first evaluate the performance of dual-quant of cuSZ. The average throughput of the dual-quant step on each tested dataset is shown in Table 7. Compared with the original serial CPU-SZ, the predict-quant throughput is improved by more than 1000× via our proposed dual-quant on the GPU. This improvement is because dual-quant entirely eliminates the RAW dependency and leads to fine-grained (per-point) parallel computation, which is significantly accelerated on the GPU.

We then evaluate the performance of our implemented Huffman coding step by step. First, we conduct the experiment of Huffman

[3] The `QMCPACK` dataset includes only one field but with two representations.

histogram computation and show its throughput performance[4]. Efficiently computing a histogram on a GPU is an open challenging problem, because of the way that multiple threads need to write to the same memory locations simultaneously. Here, we present a method that, while a bottleneck in the Huffman process, is a 2× improvement from a serial implementation.

Next, we perform the experiment of constructing codebook with different numbers of quantization bins, as shown in Table 3. We note that the execution times of building a Huffman tree and creating a codebook are consistent with our time complexity analyses in §3.2.2. We use 1,024 quantization bins by default. Since the time overhead of constructing a codebook depends only on the number of quantization bins, it is almost fixed—for example, 4.81 ms—for the remaining experiments. We also note that a larger data size lowers the relative performance overhead of constructing a codebook, thus leading to higher overall performance.

		1071 MB HACC	25 MB CESM-ATM	95 MB HURRICANE	512 MB NYX	602 MB QMCPACK
enc.64	μs	4,274.3	97.1	385.8	2,044.7	2,401.4
	GB/s	250.9	255.1	251.7	251.1	251.1
enc.32	μs	2,839.3	64.1	255.8	1,358.6	1,595.6
	GB/s	377.7	386.6	379.6	377.9	377.9

Table 4: Performance of encoding and deflating based on the constructed codebook (averaged based on all fields for each set).

We also evaluate the performance of encoding and decoding based on the canonical codebook. To increase the memory bandwidth utilization, we adapt online selection of Huffman codeword representation between a `uint32_t` and a `uint64_t`. Table 4 illustrates that our encoding achieves about 250 GB/s for `uint64_t` and about 380 GB/s [5] for `uint32_t`, based on the test with all 111 fields under the error bound of 1e-4. Hence, we conclude that using a uint32_t enables significantly higher performance than using a uint64_t. Because of the coarse-grained chunk-wise parallelization, the performance of deflating is about 60 GB/s, which is lower than the encoding throughput of 380 GB/s. Consequently, the Huffman coding performance is bounded mainly by the deflating throughput.

To improve the deflating and inflating performance, we further evaluate different chunk sizes and identify the appropriate sizes for both deflating and inflating on the tested datasets, as shown in Table 6. Specifically, we evaluate chunk sizes ranging from 2^6 to 2^{16}, due to different field sizes. We observe that using a total of around 2e4 concurrent threads consistently achieves the optimal throughput. Note that inflating must follow exactly the same data chunking strategy as deflating; thus we need to select the same chunk size. Even under this constraint, our selected chunk sizes still achieve throughputs close to the peak ones, as illustrated in Table 6. Therefore, we conclude that the overall optimal performance can be achieved by setting up a total of 2e4 concurrent threads in practice.

Next, we evaluate the overall compression and decompression performance of cuSZ, as shown in Table 7. We compare cuSZ with cuZFP in terms of the kernel performance and the overall performance that includes the GPU-to-CPU communication cost. Note that the performance of cuZFP is highly related to its user-set fixed bitrate according to the previous study [39], whereas the

[4]All throughputs shown are measured based on the original data size and time.
[5]NVIDIA V100 GPU has a theoretical peak memory bandwidth of 900 GB/s.

	cuSZ bitrate	cuSZ CR	cuSZ PSNR	cuZFP bitrate	cuZFP CR	cuZFP PSNR
CESM-ATM	3.08 bits	10.4	85.3 dB	12 bits	2.7	88.7 dB
HURRICANE	3.45 bits	9.3	87.0 dB	12 bits	2.7	81.9 dB
NYX	2.49 bits	12.8	86.0 dB	6 bits	5.3	85.1 dB
QMCPACK	3.38 bits	9.5	85.0 dB	8 bits	4.0	84.0 dB

Table 5: Bitrate comparison at PSNR of about 85 dB (cuSZ's PSNRs are no lower than cuZFP's). CR stands for compression ratio.

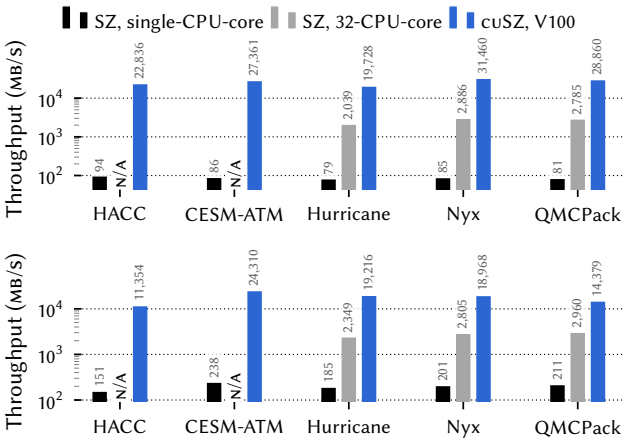

Figure 5: Compression (top) and decompression (bottom) throughput of cuSZ and CPU-SZ on tested datasets.

performance of cuSZ is hardly affected by the user-set error bound. Therefore, we choose the acceptable fixed bitrate for cuZFP, which generates data distortion (i.e., PSNR of about 85 dB) similar to that of cuSZ, as shown in Table 5. Also, note that we exclude cuZFP for HACC in Table 7, because cuZFP generates fairly low compression quality on 1D HACC. In particular, even when the bitrate is as high as 16, the PSNR is only about 20 dB, which is not usable. The throughput in Table 7 is calculated based on the original data size rather than the size of the data transferred between the GPU and CPU. Table 7 shows that cuZFP has a higher kernel throughput but lower GPU-to-CPU throughput than does cuSZ. The reason is that cuSZ provides a much higher compression ratio than does cuZFP with the same data distortion.

We note that the overall throughputs of cuSZ and cuZFP are close to each other with respect to the CPU-GPU interconnect (16-lane PCIe 3.0) bandwidth in our evaluation. Generally speaking, many applications in GPU-based HPC systems generate the data on GPUs, so the compression needs to be directly performed on the data in the GPU memory, and the compressed data currently must be transferred from GPUs to disks through CPUs. Current state-of-the-art CPU-GPU interconnect technologies such as NVLink [40] can typically provide a theoretical transfer rate of 50 GB/s over two links, while our cuSZ's compression kernel can provide comparable throughput of about 40 GB/s. Although cuZFP's compression kernel achieves about 70 GB/s, its overall throughput is limited by the CPU-GPU bandwidth of 50 GB/s. So, the data transfer between CPU and GPU is still the bottleneck for high-throughput compression kernels (e.g., not higher than 50 GB/s). Moreover, the decompression throughput of cuSZ is lower than its compression throughput and that of cuZFP. This is because only coarse-grained chunking

	HACC			CESM			HURRICANE			NYX			QMCPACK		
CHUNK	1071.8 MB	280,953,867 f32		24.7 MB	6,480,000 f32		95.4 MB	25,000,000 f32		512 MB	134,217,728 f32		601.5 MB	157,684,320 f32	
SIZE	#THREAD	DEFLATE	INFLATE	#THREAD	DEFLATE	INFLATE	#THREAD	DEFLATE	INFLATE	#THREAD	DEFLATE	INFLATE	#THREAD	DEFLATE	INFLATE
2^6	.	.	.	1.0e5	11.3	25.0	.	.	.	.	.	.	.	.	.
2^7	.	.	.	5.1e4	15.5	37.8	.	.	.	.	.	.	.	.	.
2^8	.	.	.	**2.5e4**	**67.1**	**41.6**	9.8e4	5.1	11.0	.	.	.	.	.	.
2^9	.	.	.	1.3e4	55.6	30.7	4.9e4	10.2	9.4	.	.	.	.	.	.
2^{10}	.	.	.	6.3e3	48.2	19.6	**2.4e4**	**64.6**	**34.2**	1.3e5	4.7	5.9	1.5e5	4.7	5.1
2^{11}	1.4e5	4.6	2.8	.	.	.	1.2e4	57.3	27.7	6.6e4	5.7	6.3	7.7e4	5.2	6.2
2^{12}	6.9e4	5.1	5.1	.	.	.	6.1e3	50.7	17.8	3.3e4	25.1	16.1	3.8e4	12.9	11.1
2^{13}	3.4e4	13.6	12.1	.	.	.	.	.	.	**1.6e4**	**69.7**	**52.4**	**1.9e4**	**72.7**	**40.3**
2^{14}	**1.7e4**	**63.1**	**35.0**	.	.	.	.	.	.	8.2e3	72.4	42.6	9.6e3	75.9	29.0
2^{15}	8.6e3	65.8	28.1	.	.	.	.	.	.	4.1e3	50.0	23.1	4.8e3	56.0	16.1
2^{16}	4.3e3	45.9	14.3	.	.	.	.	.	.	.	.	.	.	.	.

Table 6: Throughputs (in GB/s) versus different numbers of threads launched on V100. The optimal thread number in terms of inflating and deflating throughput is shown in bold.

		PREDICT. (P) + QUANT. (Q)	HUFFMAN		KERNEL COMPRESSION	GPU-TO-CPU VALREL@10^{-4}	OVERALL COMPRESSION	HUFFMAN DECODING	REVERSED (P+Q)	KERNEL DECOMPRESSION
		MB/S	MB/S				MB/S	MB/S	MB/S	MB/S
CPU-SZ	HACC	137.7	328.6		-	-	94.1	196.0	659.3	151.1
	CESM-ATM	105.0	459.1		-	-	85.5	502.2	451.9	237.9
	HURRICANE	93.8	504.0		-	-	78.5	524.5	306.8	185.0
	NYX	98.5	648.7		-	-	84.7	670.4	300.5	201.8
	QMCPACK	97.5	396.2		-	-	80.8	660.3	313.4	211.1
			HISTOGRAM	CODEBOOK	CODING			CANONICAL		
		GB/S	GB/S	MS	GB/S	GB/S	GB/S	DEC. GB/S	GB/S	GB/S
cuSZ	HACC	207.7	602.8	5.16	54.1	40.0	53.2	22.8	35.0	16.8	11.4
	CESM-ATM	252.1	345.3	4.33	57.2	41.1	81.9	27.4	41.6	58.5	24.3
	HURRICANE	175.8	418.0	4.81	55.2	38.2	40.8	19.7	34.2	43.9	19.2
	NYX	200.2	427.6	3.84	58.8	41.1	134.1	31.6	52.4	29.7	19.0
	QMCPACK	189.6	346.1	4.09	61.0	40.7	99.2	28.9	40.3	22.4	14.4
cuZFP	HACC	-	-	-	-	-	-	-	-	-	-
	CESM-ATM	-	-	-	-	47.6	27.7	17.5	-	-	113.1
	HURRICANE	-	-	-	-	83.7	27.7	20.8	-	-	102.2
	NYX	-	-	-	-	71.3	56.3	31.7	-	-	103.1
	QMCPACK	-	-	-	-	72.6	42.5	26.8	-	-	115.5

Table 7: Breakdown comparison of kernel performance among CPU-SZ, cuSZ, and cuZFP. Here "-" represents for N/A.

can be applied to decompression, as mentioned in §3.3. Here we argue that the compression throughput is more important than the decompression throughput, because users use the CPU-SZ mainly to decompress the data for postanalysis and visualization instead of the GPU after the compressed data is transferred and stored to parallel file systems [11, 39].

We note that cuSZ on the `CESM-ATM` dataset exhibits much lower performance than on other datasets. This is due to the fact that each field of the `CESM-ATM` dataset is fairly small (~25 MB), such that the codebook construction cost turns out to be relatively high compared with other steps for this dataset. In fact, the codebook construction would not be a bottleneck for a relatively large dataset (such as hundreds of MBs per field), which is more common in practice (e.g., `HACC`, `Nyx`, `QMCPACK`).

We also compare the performance of cuSZ with that of the production version of SZ running on a single CPU core and multiple CPU cores. The parallelization of OpenMP-SZ is achieved by simply chunking the whole data without any further algorithmic optimization (such as our proposed DUAL-QUANT). In particular, each thread is assigned with a fixed-size block and runs the original sequential CPU-SZ code. The points on the border are handled similar to cuSZ (as shown in Figure 2). The main differences between OpenMP-SZ and cuSZ are fourfold: (1) In the proposed DUAL-QUANT, each point in cuSZ is assigned to a GPU thread, whereas OpenMP-SZ uses a CPU thread to handle a block of data points. (2) After POSTQUANT, the data are transformed into integers (units of error bound), and all the following arithmetic operations are performed on these integers. Hence cuSZ does not need to handle the errors that are introduced by floating-point operations (e.g., underflow). (3) OpenMP-SZ does not fully parallelize Huffman coding, whereas cuSZ provides an efficient parallel implementation of Huffman coding on GPU. (4) OpenMP-SZ supports only 3D datasets, so in our comparison we use 3D `Hurricane Isabel` and `Nyx` and mark N/A for non-3D datasets in Figure 5. It illustrate the compression and decompression throughput of cuSZ (considering the CPU-GPU communication overhead) and CPU-SZ. Compared with the serial SZ, the overall compression performance can improved by 242.9× to 370.1×. cuSZ also improves the overall performance by 11.0× to 13.1× over SZ running with OpenMP on 32 cores.

4.2.2 ***Compression Quality.*** We then present the compression quality of cuSZ compared with another advanced GPU-supported lossy compressor—cuZFP—based on the compression ratios and data distortions on the tested datasets. We use the *peak signal-to-noise ratio (PSNR)* [6] to evaluate the quality of the reconstructed data.

[6] PSNR is calculated as $\text{PSNR} = 20 \cdot \log_{10}\left[(d_{\max} - d_{\min})/\text{RMSE}\right]$, where N is the number of data points and $d_{\max}$/ $d_{\min}$ is the maximal/minimal value. *Root mean*

The larger the PSNR, the lower reconstructed distortion, hence the more accurate postanalysis.

We compare cuSZ and cuZFP only on two 3D datasets—`Hurricane Isabel` and `Nyx`—because the compression quality of cuZFP on the 1D/2D datasets is much lower than that on the 3D datasets . For a fair comparison, we plot the rate-distortion curves for both cuSZ and cuZFP on all the fields of the two datasets and compare their compression quality in PSNR at the same compression ratio.

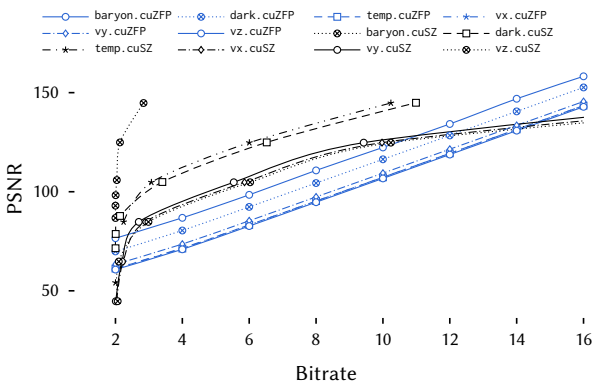

Figure 6: Comparison of rate-distortion between cuSZ (fixed `valrel`) and cuZFP (fixed rate) on `Nyx` dataset.

Figure 6 shows the rate-distortion curves of cuSZ and cuZFP on the `Nyx` dataset. We observe that cuSZ generally has a higher PSNR than does cuZFP with the same compression ratio on the `Nyx` dataset. In other words, cuSZ provides a much higher compression ratio compared with cuZFP given the same compression quality. The main reason is twofold: (1) ZFP has better compression quality with the absolute error bound (fix-accuracy) mode than with the fixed-rate mode (as indicated by the ZFP developer [41]); and (2) the ℓ-predictor of cuSZ has a higher decorrelation efficiency than does the block transform of cuZFP, especially on the field with a large value range and concentrated distribution, such as `baryon_density`.

Similar results for cuSZ and cuZFP are observed on the `Hurricane Isabel` dataset, as shown in Figure 7. We note that the rate-distortion curves for cuSZ—namely, `QCLOUD`, `QICE`, `CLOUD`—notably increase when the compression ratio decreases. This is because there are areas full of zeros, causing the compression ratio to change very slowly when the error bound is smaller than a certain value. In other words, most of the nonzeros are unpredictable, and the zeros are always predictable.

We also illustrate the overall rate-distortion curves of cuSZ and cuZFP on the `Hurricane` and `Nyx` dataset, as shown in Figure 8. For example, cuSZ provides a 2.41× (2.49 vs. 6) lower bitrate over cuZFP on the `Nyx` dataset and a 3.48× (3.45 vs. 12) lower bitrate over cuZFP on the `Hurricane Isabel` dataset, with reasonable PSNRs, as shown in Table 5.

The reason is that, according to §3.1.1, cuSZ sets all the values in the padding layer to `0` and uses these zeros to predict the top-left data points, resulting in better prediction on the tested datasets,

squared error (RMSE) is obtained by $\mathrm{sqrt}\left[\frac{1}{N}\sum_{i=1}^{N}(d_i - d_i^{\bullet})^2\right]$, where d_i and $d_i^{\bullet}$ refer to the original and decompressed values, respectively.

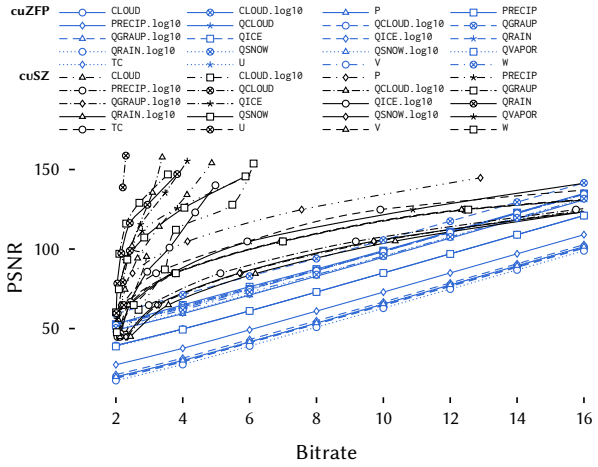

Figure 7: Comparison of rate-distortion between cuSZ (fixed `valrel`) and cuZFP (fixed rate) on `Hurricane Isabel` dataset.

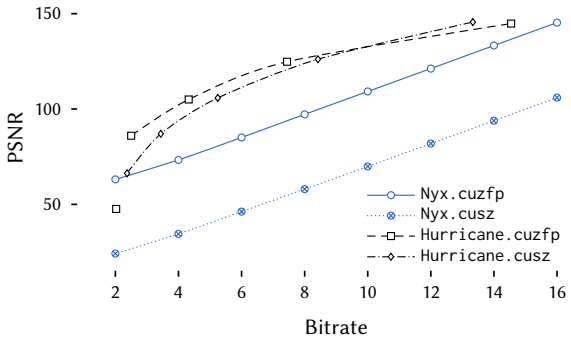

Figure 8: Comparison of overall rate distortion between cuSZ (fixed `valrel`) and cuZFP (fixed rate) on `Hurricane` and `Nyx` datasets (averaged based on all fields).

FIELD	SZ-1.4	cuSZ	FIELD	SZ-1.4	cuSZ
CLOUDf48	84.99	94.18	QSNOWf48	84.31	93.36
CLOUDf48.log10	84.51	87.17	QSNOWf48.log10	84.87	84.93
Pf48	84.79	84.79	QVAPORf48	84.79	84.80
PRECIPf48	85.35	92.86	TCf48	84.79	84.79
PRECIPf48.log10	84.82	84.77	Uf48	84.79	84.79
QCLOUDf48	85.03	98.91	Vf48	84.79	84.79
QCLOUDf48.log10	85.22	95.21	Wf48	84.79	84.79
QGRAUPf48	88.21	97.02	baryon_density	89.71	98.25
QGRAUPf48.log10	84.90	84.82	dark_matter_density	86.57	87.77
QICEf48	84.61	95.51	temperature	84.77	84.77
QICEf48.log10	85.56	85.77	velocity_x	84.77	84.77
QRAINf48	85.36	97.37	velocity_y	84.77	84.77
QRAINf48.log10	84.93	84.56	velocity_z	84.77	84.77
Hurricane avg.	85.01	86.96	Nyx avg.	85.58	85.98

Table 8: Comparison of PSNR between cuSZ and SZ-1.4 on `Hurricane` (FIRST 20) and `Nyx` (LAST 6) under `valrel` = 10^{-4}.

especially for the fields with large value ranges and a large majority of values close to zero (such as `CLOUDf48`, `QSNOWf48`, and `baryon_density` as shown in Table 9). However, SZ-1.4's prediction highly depends on the first data point's value, so it may cause low prediction accuracy when the first data point deviates largely from most of the other points. Therefore, cuSZ and SZ-1.4 have similar PSNRs on the datasets represented by the logarithmic scale.

CLOUDf48

min	1%	25%	50%	75%	99%	max	range
0.00e+0	0.00e+0	0.00e+0	0.00e+0	0.00e+0	2.53e-4	2.05e-3	2.05e-3
eb =2.05e-7	89.20% in $[-eb, eb]$, and 89.20% in [min, min + eb]						
$\frac{1}{10}eb$ =2.05e-8	88.50% in $[-\frac{1}{10}eb, \frac{1}{10}eb]$, and 88.50% in [min, min + $\frac{1}{10}eb$]						

QSNOWf48

min	1%	25%	50%	75%	99%	max	range
0.00e+0	0.00e+0	1.11e-10	1.96e-9	6.34e-9	6.01e-5	8.56e-4	8.56e-4
eb =8.56e-8	88.90% in $[-eb, eb]$, and 88.90% in [min, min + eb]						
$\frac{1}{10}eb$ =8.56e-9	80.90% in $[-\frac{1}{10}eb, \frac{1}{10}eb]$, and 80.90% in [min, min + eb]						

baryon density

min	1%	25%	50%	75%	99%	max	range
5.80e-2	1.37e-1	3.22e-1	5.06e-1	8.75e-1	7.42e+0	1.16e+5	1.16e+5
eb =1.16e+1	99.50% in $[-eb, eb]$, and 99.50% in [min, min + eb]						
$\frac{1}{10}eb$ =1.16e+0	83.30% in $[-\frac{1}{10}eb, \frac{1}{10}eb]$, and 84.40% in [min, min + $\frac{1}{10}eb$]						

Table 9: Statistical information (percentile) of example fields having high PSNR under `vrrel` = 10^{-4}. The range of eb or even $\frac{1}{10}eb$ at `0` or min value cover a majority of data in the fields.

5 RELATED WORK

5.1 GPU-Accelerated Scientific Compression

Scientific data compression has been studied for many years for reducing storage and I/O overhead. It includes two main categories: lossless compression and lossy compression. Lossless compressors for scientific datasets such as FPC [42] and FPZIP [43] ensure that the decompressed data is unchanged, but they provide only a limited compression ratio because of the significant randomness of the ending mantissa bit of HPC floating-point data. According to a recent study [6], the compression ratio of lossless compressors for scientific datasets is generally up to 2:1, which is much lower than the user-desired ratio for HPC applications.

Error-bounded lossy compression significantly reduces the size of scientific data while maintaining desired data characteristics. Traditional lossy compressors (such as JPEG [44]) are designed for image and visualization purposes; however, they are difficult to be applied to scientific datasets because of scientists' specific data fidelity requirement. Recently, error-bounded lossy compressors (such as SZ [8] and ZFP [45]) have been developed for scientific datasets. Such compressors provide strict error controls according to user requirements. Both SZ and ZFP, for example, provide an absolute error bound in their CPU version.

Different from SZ's prediction-based compression algorithm, ZFP's algorithm is based on a block transform. It first splits the whole dataset into many small blocks. It then compresses the data in each block separately in four main steps: exponent alignment, customized near-orthogonal transform, fixed-point integer conversion, and bit-plane-based embedded coding. A truncation is performed based on the user-set bitrate. Recently, the ZFP team released their CUDA version, called cuZFP [15]. cuZFP provides much higher throughputs for compression and decompression compared with the CPU version [39]. However, the current cuZFP only supports fixed-rate mode, which significantly limit its adoption in practice.

5.2 Huffman Coding on GPU

During the Huffman coding process, a specific method is used to determine the bit representation for each symbol, which results in variable length prefix codes. The set of these prefix codes make up the codebook, with each prefix code based on the symbols frequency in the data. This codebook is then used to replace each input symbol with its corresponding prefix code. Previous studies have shown that Huffman coding achieves better performance in parallel on a GPU than in serial on a CPU. In general, parallel Huffman coding obtains each codeword from a lookup table (generated by a Huffman tree) and concatenates codewords together with other codewords. However, a severe performance issue arises when different threads write codewords with different lengths, which results in warp divergence on GPU [46]. The most deviation between methods occurs in concatenating codewords.

Fuentes-Alventosa et al. [47] proposed a GPU implementation of Huffman coding using CUDA with a given table of variable-length codes, which improves the performance by more than 20× compared with a serial CPU implementation. Rahmani et al. [48] proposed a CUDA implementation of Huffman coding based on serially constructing the Huffman codeword tree and parallel generating the byte stream, which can achieve up to 22× speedups compared with a serial CPU implementation without any constraint on the maximum codeword length or data entropy. Lal et al. [49] proposed a Huffman-coding-based memory compression for GPUs (called E^2MC) based on a probability estimation of symbols. It uses an intermediate buffer to reduce the required memory bandwidth. In order to place the codeword into the correct memory location, E^2MC extends the codeword to the size of the buffer length and uses a barrel shifter to write the codeword to the correct location. Once shifted, the codeword is bitwise ORed with the intermediate buffer, and the write location is increased by the codeword length.

6 CONCLUSION AND FUTURE WORK

In this work, we propose cuSZ, a high-performance GPU-based lossy compressor for NVIDIA GPU architectures that effectively improves the compression throughput for SZ compared with the production version on CPUs. We propose a dual-quantization scheme to completely remove the strong data dependency in SZ's prediction-quantization step and implement an efficient customized Huffman coding. We also propose a series of techniques to optimize the performance of cuSZ, including fine-tuning the chunk size, adaptively selecting Huffman code representation, and reusing memory. Experiments on five real-world HPC simulation datasets show that our proposed cuSZ improves the compression throughput by 242.9× to 370.1× over the serial CPU version and 11.0× to 13.1× over the parallel CPU version. Compared with another state-of-the-art GPU-supported lossy compressor, cuSZ improves the compression ratio by 2.41× to 3.48× with reasonable data distortion on the tested datasets. We plan to further optimize the performance of decompression, implement other data prediction methods such as linear-regression-based predictor, and evaluate the performance improvements of parallel I/O with cuSZ.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations - the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nationâĂŹs exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This work was also supported by the National Science Foundation under Grants CCF-1619253, OAC-2003709, OAC-1948447/2034169, and OAC-2003624/202042084. We would like to thank The University of Alabama for providing the startup funding for this work.

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 (Big Data)*, Seattle, WA, USA: IEEE, 2018, pp. 438–447.

[4] X. Liang, S. Di, S. Li, D. Tao, Z. Chen, and F. Cappello, "Exploring best lossy compression strategy by combining SZ with spatiotemporal decimation," in *The 4th International Workshop on Data Reduction for Big Scientific Data*, Dallas, TX, USA: IEEE, 2018.

[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] A. H. Baker, H. Xu, J. M. Dennis, M. N. Levy, D. Nychka, S. A. Mickelson, J. Edwards, M. Vertenstein, and A. Wegener, "A methodology for evaluating the impact of data compression on climate simulation data," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, Vancouver, BC, Canada: ACM, 2014, pp. 203–214.

[8] 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.

[9] 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.

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

[11] 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.

[12] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "Wavesz: A hardware-algorithm co-design of efficient lossy compression for scientific data," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, USA, 2020, pp. 74–88.

[13] N. V.T. C. GPU, https://www.nvidia.com/en-us/data-center/v100/, Online, 2020.

[14] 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.

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

[16] Scientific Data Reduction Benchmarks, https://sdrbench.github.io/, Online, 2019.

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

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

[19] 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 (CLUSTER)*, Belfast, UK: IEEE, 2018, pp. 179–189.

[20] I. Foster, M. Ainsworth, B. Allen, J. Bessac, F. Cappello, J. Y. Choi, E. Constantinescu, P. E. Davis, S. Di, W. Di, *et al.*, "Computing just what you need: Online data analysis and reduction at extreme scales," in *European Conference on Parallel Processing*, Santiago de Compostela, Spain: Springer, 2017, pp. 3–19.

[21] 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.

[22] A. M. Gok, S. Di, A. Yuri, D. Tao, V. Mironov, X. Liang, and F. Cappello, "PaSTRI: A novel data compression algorithm for two-electron integrals in quantum chemistry," in *IEEE International Conference on Cluster Computing (CLUSTER)*, Belfast, UK: IEEE, 2018, pp. 1–11.

[23] 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 (IPDPS)*, Vancouver, BC, Canada: IEEE, 2018, pp. 348–357.

[24] S. Jin, S. Di, X. Liang, J. Tian, D. Tao, and F. Cappello, "Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, Phoenix, AZ, USA: ACM, 2019, pp. 159–170.

[25] X. Liang, S. Di, S. Li, D. Tao, B. Nicolae, Z. Chen, and F. Cappello, "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*, Denver, CO, USA: ACM, 2019, p. 33.

[26] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, Z. Chen, and F. Cappello, "Significantly improving lossy compression for hpc datasets with second-order prediction and parameter optimization," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, Stockholm, Sweden: ACM, 2020, pp. 89–100.

[27] N. Zhang, Y.-s. Chen, and J.-l. Wang, "Image parallel processing based on GPU," in *2010 2nd International Conference on Advanced Computer Control*, vol. 3, Shenyang, China: IEEE, 2010, pp. 367–370.

[28] J. Gómez-Luna, J. M. González-Linares, J. I. B. Benítez, and N. G. Mata, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications*, vol. 24, pp. 899–908, 2012.

[29] Y. Abu-Mostafa and R. McEliece, "Maximal codeword lengths in Huffman codes," *Computers & Mathematics with Applications*, vol. 39, no. 11, pp. 129–134, 2000.

[30] M. L. Barnett, *Canonical huffman encoded data decompression algorithm*, US Patent 6,657,569, 2003.

[31] M Harris and K Perelygin, *Cooperative groups: Flexible cuda thread programming*, 2017.

[32] PantaRhei cluster, https://www.dingwentao.com/experimental-system, Online, 2019.

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

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

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

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

[37] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Optimizing lossy compression rate-distortion from automatic online selection between SZ and ZFP," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1857–1871, 2019.

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

[39] 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*, New Orleans, LA, USA: IEEE, 2020, pp. 105–115.

[40] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[41] zfp Compression Ratio and Quality, https://computing.llnl.gov/projects/floating-point-compression/zfp-compression-ratio-and-quality, Online, 2019.

[42] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2008.

[43] 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.

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

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

[46] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in GPUs: Characterization, impact, and mitigation," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Orlando, FL, USA: IEEE, 2014, pp. 284–295.

[47] A. Fuentes-Alventosa, J. Gómez-Luna, J. M. González-Linares, and N. Guil, "Cuvle: Variable-length encoding on cuda," in *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, Madrid, Spain: IEEE, 2014, pp. 1–6.

[48] H. Rahmani, C. Topal, and C. Akinlar, "A parallel huffman coder on the CUDA architecture," in *2014 IEEE Visual Communications and Image Processing Conference*, Valletta, Malta: IEEE, 2014, pp. 311–314.

[49] S. Lal, J. Lucas, and B. Juurlink, "Eˆ 2MC: Entropy encoding based memory compression for GPUs," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Orlando, FL, USA: IEEE, 2017, pp. 1119–1128.