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
sd11@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 GPU 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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PACT ’20, October 3–7, 2020, Virtual Event, GA, USA
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.1145/3410463.3414624

CCS CONCEPTS
• Computing methodologies → Massively parallel algorithms.

KEYWORDS
Lossy Compression; Scientific Data; GPU; CUDA; Performance

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\textsuperscript{1} is the first strictly error-bounded lossy compressor on GPU for scientific data. Our contributions are summarized as follows.

- 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.

\section{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 $|d - d^*| < eb$, where $d$ and $d^*$ 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) \textbf{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) \textbf{Data Prediction} SZ predicts the value of each data point by a data-fitting predictor, e.g., a Lorenzo predictor [14] (abbreviated as $\ell$-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) \textbf{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.

\textsuperscript{1}The code is available at https://github.com/hipdac-lab/cuSZ.
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.

| TABLE 1: Parallelism implemented for cUSZ’s subprocedures (kernels) in compression and decompression. |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| compression     | decompression   |
| sequential      | dual-quantization |
| quantize        | inflate          |
| histogram        | Huffman decode   |
| build Huffman tree | reversed         |
| canonize codebook | dual-quantization |

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 \( \ell \)-prediction to the outer layer instead, such that every point in the block is consistently processed based on the \( \ell \)-predictor, avoiding thread/warp divergence. Moreover, we initialize the padding layer with 0s; 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\times16 for 2D data, and 8\times8\times8 for 3D data.

![Figure 2: Data chunking and padding in cuSZ.](image)

3.1.2 Dual-Quantization Scheme. In the following discussion, we use circle \( \circ \) and bullet \( \bullet \) to denote the compression and decompression procedure, respectively. We use star \( \star \) to denote all the values related to the data reconstruction in compression. The subscript \( k \) represents the \( k \)th 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 \( e^p_{k-1} \) (i.e., \( d^p_{k-1} - p^p_{k-1} \)) 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 \( e^r_{k-1} \) and the reconstructed value \( d^r_{k-1} \) are generated by using \( q_{k-1} \), \( eb \), and \( p^p_{k-1} \). After that, \( d^r_{k-1} \) is written back to replace \( d_{k-1} \). This procedure ensures that \( d^r_{k-1} \) is equivalent to the reconstructed \( d^r_{k-1} \) during decompression (as shown in Figure 3); however, the \( k \)th 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 \( d^r_{k-1} \) 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.](image)

**Algorithm 1: Original SZ of predict-quant**

```plaintext
for d ∈ D do
  p^p ← (d_{in})
  e^p ← p^p - d
  if e^p / eb < CAP (in-CAP) then
    e^o ← INTEGERIZE(e^p/(2×eb))
    \( \text{REHEARSAL} \leftarrow p^p + 2 \cdot e^o \cdot eb \)
    \( \text{WATCHDOC(REHEARSAL - d < eb, fallback: OUTLIER)} \)
    else
      OUTLIER: e^o ← 0 and record the verbatim \( x \) \( \leftarrow d \)
    end if
  end if
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 \( D^\circ \) with \( d^\circ \in D^\circ \). We write this as \( \text{PREQUANTIZATION} \) (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, $|d - d^*| · eb < eb$.

After the PREQUANTIZATION, we can calculate each predicted value based on its surrounding values (denoted by $d_{\text{3b}}$) and the $\ell$-predictor as

$$P^0 = \{ p^0 | p^0 = \ell(d_{3b}) \}.$$  

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 $\delta$ instead of $e$ to denote this difference:

$$\Delta^0 = \{ \delta^0 | \delta^0 = d^0 - p^0, \quad d^0 \in D^0, \quad p^0 \in P^0 \}.$$  

Then, the quantization code $q^*$ is generated based on $\delta$. Note that $q^*$ is quantitatively equivalent to $\delta^*$, represented differently: $\delta^*$ 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 $d^*$) based on losslessly decoded $q^* \equiv q^*$ (hence exactly $\delta^*$) and predicted $p^0$, 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 $\delta^*$ and $d^*$ during the compression, as shown in Figure 3. In fact, for $(k-1)$th iteration, $\delta^*_{k-1}$ is strictly equal to $\delta^0_{k-1}$, because casting quantization code $q^0_{k-1}$ to $\delta^*_{k-1}$ is an exact reversed procedure of casting $\delta^0_{k-1}$ to $q^0_{k-1}$.

Similarly, $d^*_{k-1}$ and $d^0_{k-1}$ are also strictly equivalent. Consequently, unlike the original SZ that must write $d^*_{k-1}$ back to update $d^0_{k-1}$ before the kth iteration, $d^*_{k-1} \equiv d^0_{k-1}$ always holds in our proposed DUAL-QUANT approach. As illustrated in Figure 3, after PREQUANT, all $d^0$ 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 $\ell$-predictor is given by

$$\sum_{0 \leq k_1, \ldots, k_\ell \leq n} \left( \prod_{j=1}^{\ell} (-1)^{j+1} \binom{n}{k_j} \right) \cdot d_{x_1-k_1, \ldots, x_\ell-k_\ell},$$

where $\sum_{0 \leq k_1, \ldots, k_\ell \leq n} \left( \prod_{j=1}^{\ell} (-1)^{j+1} \binom{n}{k_j} \right) = 1$ and $d \in D$. For example, 1D 1-order $\ell$-predictor is $p^\circ_{x} = d^0_{x-1}$, and 2D 1-order $\ell$-predictor is $p^\circ_{x,y} = \ell(d^0_{x,y}) = d^0_{x-1,y} + d^0_{x,y-1} - d^0_{x-1,y-1}$, as illustrated in Figure 1. We note that all the coefficients in the formula of the $\ell$-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 $\epsilon$, however, the original

<table>
<thead>
<tr>
<th>Algorithm 2: cuSZ of DUAL-QUANT</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $d \in D$ concurrently do</td>
</tr>
<tr>
<td>$d^0 \leftarrow d/(2\times eb)$</td>
</tr>
<tr>
<td>(FP representation) PREQUANT</td>
</tr>
<tr>
<td>$d \leftarrow d^0$</td>
</tr>
<tr>
<td>BARRIER</td>
</tr>
<tr>
<td>$p^0 \leftarrow \ell(d^0_{\text{3b}})$</td>
</tr>
<tr>
<td>$\delta^0 \leftarrow d^0 - p^0$</td>
</tr>
<tr>
<td>if $\delta^0 &lt; \text{CAP/2 (IN-CAP)}$ then</td>
</tr>
<tr>
<td>$\delta^0_{\text{ postqant }} \leftarrow \text{CAST-FLOAT2INT}(\delta^0)$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>OUTLIER: $\delta^0 \leftarrow 0$ and record the verbatim $x \leftarrow d^0$</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>for $d^* \in D^*$ to reconstruct cascadingly do</td>
</tr>
<tr>
<td>$p^* \leftarrow \ell(d^0_{\text{3b}})$</td>
</tr>
<tr>
<td>$d^* \leftarrow (p^* + \delta^0_{\text{ postqant }}) \cdot (2 \times eb)$ if IN-CAP else verbatim $x$</td>
</tr>
<tr>
<td>end for</td>
</tr>
</tbody>
</table>

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 $eb$.

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.](image)

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 codeword-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: 1️⃣ linear scanning of the base codebook (sequentially $O(k)$), which is parallelized at fine granularity with atomic operations; 2️⃣ loosely radix-sorting of the codewords by bitwidth (sequentially $O(k)$), which cannot be parallelized because of the intrinsic RAW dependency; and 3️⃣ 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 ($\mu s$) 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 1️⃣ and 3️⃣. 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 $\mu$s 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 2x60 $\mu$s, two in-grid barriers have relatively low overheads,
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 deflating is not necessarily the same as the chunk size for dual-qant, 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-qant. 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), 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 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.

<table>
<thead>
<tr>
<th>DATASETS</th>
<th>TYPE</th>
<th>DATUM SIZE</th>
<th>EXAMPLES</th>
<th>PERFIN (PB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSMOLOGY</td>
<td>HACC</td>
<td>fp32 1,071,75 MB</td>
<td>6 in total</td>
<td>x, vx</td>
</tr>
<tr>
<td>CLIMATE</td>
<td>CESM-ATM</td>
<td>fp32 280,953,867 MB</td>
<td>70 in total</td>
<td>CLDHGH, CLDLOM</td>
</tr>
<tr>
<td>CLIMATE</td>
<td>Hurricane</td>
<td>fp32 1,800,000 MB</td>
<td>70 in total</td>
<td>CLOUDf48, Uf48</td>
</tr>
<tr>
<td>CLIMATE</td>
<td>Nyx</td>
<td>fp32 512,000 MB</td>
<td>6 in total</td>
<td>baryon_density</td>
</tr>
<tr>
<td>CLIMATE</td>
<td>QMCPACK</td>
<td>fp32 601,32 MB</td>
<td>2 formats in total</td>
<td>einspine</td>
</tr>
</tbody>
</table>

#### 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-qant of cuSZ. The average throughput of the dual-qant 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 1,000 x via our proposed dual-qant on the GPU. This improvement is because dual-qant 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...
histogram computation and show its throughput performance. 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.

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 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 to 2^16, due to different field sizes. We observe that using a total of 2×4 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 2×4 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

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.

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 to 2^16, due to different field sizes. We observe that using a total of 2×4 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 2×4 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

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

<table>
<thead>
<tr>
<th>Bitrate</th>
<th>CR</th>
<th>PSNR</th>
<th>Bitrate</th>
<th>CR</th>
<th>PSNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>HACC</td>
<td>1024 MB</td>
<td>1024 MB</td>
<td>25 MB</td>
<td>25 MB</td>
<td>95 MB</td>
</tr>
<tr>
<td>enc.64</td>
<td>1024 MB</td>
<td>1024 MB</td>
<td>256 GB/s</td>
<td>256 GB/s</td>
<td>256 GB/s</td>
</tr>
<tr>
<td>enc.32</td>
<td>1024 MB</td>
<td>1024 MB</td>
<td>384 GB/s</td>
<td>384 GB/s</td>
<td>384 GB/s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>CR</th>
<th>PSNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024 MB</td>
<td>1024 MB</td>
</tr>
<tr>
<td>enc.64</td>
<td>1024 MB</td>
</tr>
<tr>
<td>enc.32</td>
<td>1024 MB</td>
</tr>
</tbody>
</table>

Figure 5: Compression (top) and decompression (bottom) throughput of cuSZ and CPU-SZ on tested datasets.
OpenMP-SZ supports only 3D datasets, so in the proposed dual-qant, OpenMP-SZ does not fully parallelize Huffman coding, whereas cuSZ is similar to sequential CPU-SZ code. The points on the border are handled similarly 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 compared with other steps for this dataset. In fact, the codebook construction costs a 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-qant). 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-qant, 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 1 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.

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

\[ \text{squared error (RMSE)} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (d_i - d_i^*)^2} \]

where $d_i$ and $d_i^*$ refer to the original and decompressed values, respectively.

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

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

![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).](image)
1.16e+5 8.56e-4

<table>
<thead>
<tr>
<th>min</th>
<th>1%</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>99%</th>
<th>max</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{eb}{eb} = 2.05e-7 )</td>
<td>89.20% in ([-eb, eb]), and 89.20% in ([min, min + eb])</td>
<td>( \frac{eb}{eb} = 2.05e-8 )</td>
<td>88.50% in ([-eb, eb]), and 88.50% in ([min, min + eb])</td>
<td>( \frac{eb}{eb} = 8.56e-9 )</td>
<td>80.90% in ([-eb, eb]), and 80.90% in ([min, min + eb])</td>
<td>( \frac{eb}{eb} = 8.56e-9 )</td>
<td>( 10^{eb} )</td>
</tr>
</tbody>
</table>

Table 9: Statistical information (percentile) of example fields having high PSNR under valrel

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 20x 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 22x 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\(^2\)MC) 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\(^2\)MC 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 242.9x to 370.1x over the serial CPU version and 11.0x to 13.1x over the parallel CPU version. Compared with another state-of-the-art GPU-supported lossy compressor, cuSZ improves the compression ratio by 2.41x to 3.48x 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 nation’s 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-2005624/2042084. We would like to thank The University of Alabama for providing the startup funding for this work.
REFERENCES


