Optimizing Error-Bounded Lossy Compression for Scientific Data on GPUs

Jiannan Tian*, Sheng Di†, Xiaodong Yu†, Cody Rivera§, Kai Zhao‡, Sian Jin*,
Yunhe Feng¶, Xin Liang‖, Dingwen Tao*, Franck Cappello†
*Washington State University, Pullman, WA, USA
†Argonne National Laboratory, Lemont, IL, USA
‡University of California, Riverside, Riverside, CA, USA
§The University of Alabama, Tuscaloosa, AL, USA
¶University of Washington, Seattle, WA, USA
‖Missouri University of Science and Technology, Rolla, MO, USA

2021 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-7281-9666-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/Cluster48925.2021.00047

***Abstract*—Error-bounded lossy compression is a critical technique for significantly reducing scientific data volumes. With ever-emerging heterogeneous high-performance computing (HPC) architecture, GPU-accelerated error-bounded compressors (such as cuSZ and cuZFP) have been developed. However, they suffer from either low performance or low compression ratios. To this end, we propose cuSZ+ to target both high compression ratios and throughputs. We identify that data sparsity and data smoothness are key factors for high compression throughputs. Our key contributions in this work are fourfold: (1) We propose an efficient compression workflow to adaptively perform run-length encoding and/or variable-length encoding. (2) We derive Lorenzo reconstruction in decompression as multidimensional partial-sum computation and propose a fine-grained Lorenzo reconstruction algorithm for GPU architectures. (3) We carefully optimize each of cuSZ kernels by leveraging state-of-the-art CUDA parallel primitives. (4) We evaluate cuSZ+ using seven real-world HPC application datasets on V100 and A100 GPUs. Experiments show cuSZ+ improves the compression throughputs and ratios by up to 18.4× and 5.3×, respectively, over cuSZ on the tested datasets.**

I. Introduction

Large-scale scientific applications for advanced instruments produce vast volumes of data every day for post hoc analysis. For instance, Hardware/Hybrid Accelerated Cosmology Code (HACC) [1, 2] may produce petabytes of data in hundreds of snapshots when simulating 1 trillion particles. It could be very inefficient to store such a large amount of data, especially in situations with relatively low I/O bandwidth on the parallel file system (PFS) [3, 4].

Data reduction is becoming an effective method to resolve the big-data issue for scientific research. Although traditional lossless data reduction methods such as data deduplication and lossless compression can guarantee no information loss, they suffer from limited compression ratios on scientific datasets. Specifically, deduplication usually reduces the scientific data size by only 20% to 30% [5], and lossless compression achieves a compression ratio of up to ∼2:1 [6]. The 2:1 is far lower than scientists' desired compression ratios (e.g., 10:1 [7]).

Error-bounded lossy compressors have been developed for years to address the issue of low compression ratio for scientific data: they can not only get very high compression ratios (such as over 100×) [3, 8, 9, 10], but strictly control the data distortion regarding the user-set error bound. Notably, a qualified lossy compressor designed for scientific data reduction should address three primary concerns simultaneously: (1) high fidelity preservation, (2) high compression ratio, and (3) high throughput. Most of the existing error-bounded lossy compressors (such as SZ [8, 9], FPZIP [11], ZFP [10]), however, are mainly designed for CPU architectures, which cannot adapt to the high throughput requirement. For example, LCLS-II laser [12], X-ray imaging data generated on advanced instruments, can result in a data acquisition rate at 250 GB/s [7]. As such, high compression throughput is critical for storing a tremendous amount of data efficiently for scientific projects.

Currently, several GPU-based error-controlled lossy compressors (such as cuSZ [13] and cuZFP [14]) have been developed, but they suffer from either sub-optimal compression throughputs or low compression ratios. For instance, cuSZ can achieve much higher compression ratios than cuZFP. Still, its performance is substantially limited by the Huffman encoding and dictionary encoding stages when compared with the up-to-date work [15]. However, the high compression ratios of SZ/cuSZ significantly depend on Huffman encoding and dictionary encoding because the output of the prediction-and-quantization step in SZ/cuSZ is often composed of many repeated symbols.

In this paper, we propose an efficient compression framework (called cuSZ+) based on the cuSZ framework, which can get both high compression ratios and high throughputs on GPUs. The notation "+" in cuSZ+ indicates that this new compression method is specifically optimized for high performance in compression and decompression on the latest GPU architecture (i.e., NVIDIA's Ampere architecture).

It is challenging to develop an efficient GPU-based error-bounded lossy compressor that can achieve high compression ratios and high throughputs at the same time. On the one hand, to develop efficient GPU code, one must maximize the parallelism from GPU threads. Moreover, the architecture/characteristics of GPU accelerators (such as coherence, divergence issues, bank conflicts, use of shared memory, use of regis-

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

978-1-7281-9666-4/21/$31.00 ©2021 IEEE
DOI 10.1109/Cluster48925.2021.00047

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

ters) must be coped with very carefully to get the optimal performance. On the other hand, state-of-the-art error-bounded lossy compressors (such as SZ [3, 8, 9]) often rely on Huffman encoding and dictionary encoding, which are procedures that contain substantial data dependencies. These dependencies make them very hard to parallelize on GPUs efficiently. For example, a Huffman tree must be built based on a code-frequency histogram before performing Huffman encoding, which has a significant data dependency inside. The CUSZ code just used one single GPU thread to do this work for simplicity. Moreover, it is fairly non-trivial to design an efficient parallel code for the dictionary encoding because of the intrinsic dependency in its repeated sequence search. CUSZ leaves this part to CPU, which may suffer from significant overhead. Our key contributions proposed particularly in CUSZ+ are summarized as follows.

- We design an adaptive compression workflow to perform run-length encoding and/or variable-length encoding (i.e., Huffman encoding) on GPUs. We exploit a sufficient condition to determine when the run-length encoding should be applied for improving compression ratio, i.e., when the average Huffman bit-length is no greater than 1.09.
- We identify and prove that the first-order Lorenzo reconstruction in decompression is equivalent to a multidimensional partial-sum computation. We propose a fine-grained Lorenzo reconstruction algorithm based on a multidimensional partial-sum and a modified quantization scheme. Such a design can fully parallelize the decompression operation with workload tuning of GPU thread, improving the overall decompression throughput significantly.
- We develop some optimization strategies to boost compression performance and scalability. For instance, we carefully optimize each kernel in compression considering CUDA architecture (e.g., reducing global memory accesses) to improve the compression throughput. We also leverage the state-of-the-art `NVIDIA::cub` parallel primitives [16] to enhance the decompression scalability and throughput.
- We evaluate CUSZ+ with seven real-world HPC application datasets from public *Scientific Data Reduction Benchmarks* [17] on two state-of-the-art GPUs–V100 and A100. Experiments show that CUSZ+ improves the compression throughputs and ratios by up to 18.4× and 5.3×, respectively, over CUSZ on the tested datasets.
- We conclude that with the advancement of GPU architecture, CUSZ+ can benefit more from the improvement of memory bandwidth than that of peak FLOPS and provide valuable insights for software and application R&D toward the exascale computing era.

II. Background and Research Motivation

In this section, we introduce the background of CUSZ (the CUDA version of SZ) [13] and our research motivation.

A. Background of CUSZ

Unlike CPU-based SZ that has only four steps (prediction, quantization, Huffman encoding, and dictionary encoding), CUSZ involves nine steps to adapt to the GPU architecture. Specifically, Step-1 splits the whole dataset into multiple blocks, each of which will be compressed independently. This design favors coarse-grained decompression. Upon splitting blocks, CUSZ's compression adopts a dual-quantization scheme (including prequantization[1], prediction, and postquantization), which can entirely remove the data dependency for the Lorenzo prediction. Then, Step-5 adopts parallel histograming to compute the frequencies of the quant-codes. Step-6 builds a canonical Huffman codebook [13] based on the histogram/frequency vector. Step-7 performs the Huffman encoding over the quant-codes. Step-8 concatenates all the Huffman codes (called deflating) on GPUs, which feeds a dictionary encoder (Zstd [18]) for further compression on CPUs in Step-9. The decompression is the reversed procedure of the compression. We refer readers to the CUSZ paper [13] for more details.

For compression, Step-6 and -9 are the main bottlenecks because Step-6 has to be executed sequentially with a single GPU thread and designing an efficient multi-thread GPU algorithm for dictionary encoding is non-trivial.

For decompression, the first step (i.e., the reversed dual-quantization) is the main bottleneck since the decompression cannot use the massive parallelism as the prequantization step does. Specifically, in the decompression stage, the data values must be reconstructed one by one, according to the Lorenzo predictor[2]. To address this issue, CUSZ adopts a coarse-grained parallel method instead: letting one GPU thread handle one independent data block in parallel. However, such a design suffers from low performance due to the underuse of massive parallelism on GPUs. In addition to Step-1, Step-9 is another significant bottleneck because the dictionary decoding is also very hard to parallelize on GPUs because of its intrinsic data dependency.

B. Research Motivations

B.1) Limitation of CUSZ*'s Compression Ratio* Full-fledged CPU-based compressors may utilize multifold techniques to boost the compression ratio: pattern-finding (e.g., BWT, LZ77), dictionary (e.g., LZ77), and variable-length encoding ("VLE"). An exemplary combination is DEFLATE (LZ77 and VLE), whose famous implementation is `gzip` (used by CPU-SZ). In comparison, CUSZ, the GPU-based lossy compressor, only leverages Huffman coding to compress prediction-error correction codes (i.e., quant-codes). Even though Huffman coding is a VLE that is optimal in bit-length (i.e., with minimal discrepancy from the entropy), no less than one bit represents a data element. Therefore, CUSZ can achieve up to 32× or 64× of compression ratios. CUSZ disregards the repeated pattern that may exist in the symbol sequence. Hence, there is a gap between CUSZ and CPU-SZ in compression ratio due to the lack of pattern-finding.

[1] All data items are quantized based on their original values before the data prediction step.

[2] Lorenzo predictor predicts the data values based on a high-order data approximation formula: e.g., $X_{[j,i]} \approx X_{[j-1,i]} + X_{[j,i-1]} - X_{[j-1,i-1]}$ for 2D dataset, where $X_{[j,i]}$ refers to the value of the data item $[j, i]$ in the dataset.

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

	HACC			Hurricane		
	qg	qh	qhg	qg	qh	qhg
1e-2	22.72 1.1×	20.33 1.0×	31.02 1.5×	43.67 1.8×	24.80 1.0×	58.76 2.4×
1e-3	7.58 0.8×	9.51 1.0×	10.01 1.1×	18.41 1.1×	17.04 1.0×	24.65 1.4×
1e-4	3.89 0.8×	4.82 1.0×	5.01 1.0×	10.31 1.1×	9.76 1.0×	12.99 1.3×
	CESM			**Nyx**		
	qg	qh	qhg	qg	qh	qhg
1e-2	61.21 2.5×	24.24 1.0×	75.50 3.1×	118.94 3.9×	30.24 1.0×	164.39 5.4×
1e-3	20.78 1.1×	18.38 1.0×	28.13 1.5×	28.25 1.2×	23.92 1.0×	40.17 1.7×
1e-4	9.98 1.0×	10.29 1.0×	12.50 1.2×	12.87 0.8×	15.27 1.0×	17.95 1.2×

TABLE I: Averaged compression ratios (per dataset) of different compression schemes on 109 fields of 4 datasets with 3 error bounds of 10^{-2}, 10^{-3}, 10^{-4} (relative to value range). *q* denotes quant-code as starting point, *h* denotes customized variable-length encoding (multi-byte-symbol Huffman coding), *g* denotes gzip-featured scheme. "*ab*" denotes scheme *a* precedes scheme *b*.

TABLE I shows the compression ratio variances by applying the CUSZ workflow followed by `gzip`. *q*, *h*, *g* denote *prediction-**q**uantization*, *multi-byte **H**uffman coding*, `gzip`, respectively, and the letter sequence indicates the order of processes (e.g., *qh* indicates that *h* comes after *q*). However, this additional `gzip` so far does not exist in CUSZ but can demonstrate the potentially achievable compression ratio by exploiting the repeated symbol pattern. More specifically, by changing the error bound from 10^{-4} to 10^{-2}, Lorenzo predictor generates the intermediate quant-codes that exhibit stronger repeated patterns, as indicated in *qh*. For example, when changing from *qh* to *qhg*, HACC data shows only a 1.04× improvement in compression ratio under the error bound of 10^{-4}, while the compression ratio is improved by 1.52× under the error bound of 10^{-2}. We use the compression ratio of this *qhg* as a reference in the following discussion.

B.2) Limitation of CUSZ*'s Decompression Performance* CUSZ follows CPU-SZ's scheme to sequentially reconstruct the prediction values in decompression (per data chunk). Specifically, the reconstructed value of one item must rely on its preceding values that are fully reconstructed. As a result, this scheme naturally has a sequential implementation because of data dependency. Moreover, since both CPU-SZ and CUSZ store the unpredicted data and the prequantized data separately, it introduces an extra handling step involving if-branch in decompression, which impedes fine-grained data parallelization. In addition, compared to CUSZ's compression kernel, its decompression kernel's throughput is relatively low [13]. In particular, the Lorenzo construction kernel can achieve the same order of magnitude of throughput as memory copy [13], while the Lorenzo reconstruction kernel has one order of magnitude less in throughput. All the above prevent CUSZ from broader use scenarios such as in-situ compression.

B.3) Importance of Lorenzo Predictor The modular design of SZ enables adaptively adopting various predictors for different scientific uses. Among all predictors, the first-order Lorenzo predictor plays an essential role in the SZ framework and is the default predictor since it achieves relatively low prediction error in most cases, as proven in prior works [3, 9, 19].

Overall, in this work, we endeavor to significantly boost the compression ratio and (de)compression performance of CUSZ (e.g., Lorenzo reconstruction kernel) by developing a series of optimization techniques to address the above issues.

III. Compressibility-Aware Framework on GPU

In this paper, we propose a compressibility-aware framework that can significantly improve compression ratios. In CUSZ [13], all the computations are executed on GPU for high-performance purposes, leading to compression ratios no greater than 32 (for single-precision, or 64 for double-precision). Such an upper bound is due to the lack of dictionary coding or other pattern-finding-based coding methods. We note that if "*data being smooth enough*" is satisfied, we can apply the alternative run-length encoding (RLE) technique to achieve a higher compression ratio while maintaining 1) the same data quality and 2) a comparable throughput. In the following text, we use *Workflow-Huffman* to denote the default "Lorenzo & multi-byte VLE" and *Workflow-RLE* to denote "Lorenzo & RLE with optional VLE".

In the following discussion, we first overview our compressibility-aware design and then give details of our optimization strategies. Fig. 1 presents an overview comparison between our novel compression framework, CUSZ+ and the previous CUSZ. The adaptivity of CUSZ+ is reflected in two workflow paths.

A. Compressibility

A.1) Source of Compressibility The rationale for the SZ framework to achieve high compression is twofold. First, integer data are easier to compress than IEEE-754 floating-point data. A non-special `float`-type number (e.g., non-zero) requires full 32 bits to represent (`double` requires 64 bits). SZ's prediction-quantization step transforms prediction errors into quant-codes in integer and eliminates the randomness in terms of floating-point mantissa. Second, the quant-codes within a predefined range would be compressed further in a lossless manner (i.e., Huffman encoding in our case), while the out-of-range prediction errors are *outliers*. Then, we enumerate the in-range quant-codes as bin numbers of the histogram and later as symbols in Huffman codebook. Without context, generic lossless compressions interpret the input as a stream of bytes. In contrast, the enumeration in a power-of-two n_{enum} can exceed 256 and thus overflow a byte. So, we use at least $\lceil n_{\text{enum}}/8 \rceil$ bytes to represent symbols, which can be single-byte or multi-byte. The byte interpretation ensures that the enumeration reflects the distribution of quant-codes such that in Huffman coding, frequent symbols are encoded with fewer bits.

A.2) Reference Compression Ratio TABLE I enumerates the *possible* lossless compression techniques that are from CPU-SZ and CUSZand show consequential compression ratio with different error bounds quantitatively. In the table, *qg* serves to demonstrate a *presumed* suboptimal scenario; the single-byte interpretation (by a generic lossless compressor) does not indicate the most likely quant-code and hence hurt the

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

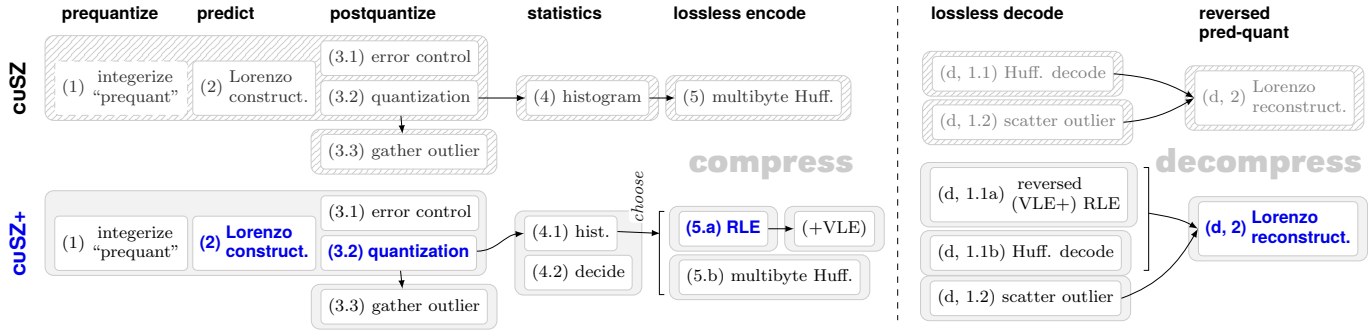


Fig. 1: Compression (left) and decompression (right) workflows of the original cuSZ (top, line patterned) and our cuSZ+ (bottom). We design an adaptive solution toward better throughput and compression ratio, featuring 2 workflow paths. The white block indicates functionality; the parenthesized number marks executing order, where the additional letters "a" and "b" mark the two paths to choose from; the gray enclosure indicates GPU kernel; the arrow indicates memory copy. The changes from cuSZ to cuSZ+ are marked with blue boldface.

compressibility. *h* indicates the Huffman coding of multi-byte symbols used in both CPU-SZ and cuSZ. *qh* indicates cuSZ compression schemes that are done on GPU entirely, while *qhg* adds *gzip* to exhibit the highest possible compression ratio, which is archived by CPU-SZ. We will use the compression ratio from *qhg* as a reference in the following discussion.

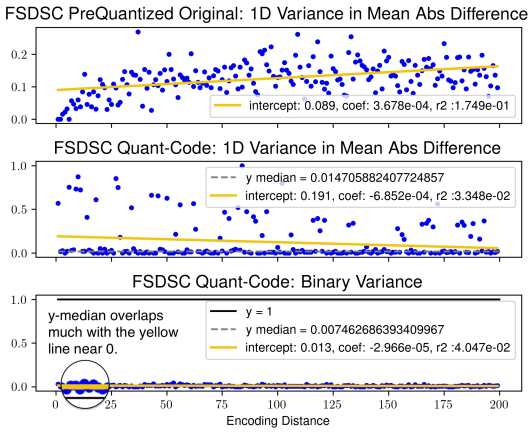
A.3) Data Feature Awareness Even though it is possible to achieve optimal compression ratio by appending another pattern-exploiting stage to cuSZ, it affects the throughput severely since `gzip` takes place on *host*. This motivates us to visit the data features that can infer compressibility. On the other hand, utilizing the repeated pattern is non-trivial because pattern-finding is usually implemented in dictionary coding with irregular accesses and has been reported as low in throughput: for instance, LZ4 features relatively high throughput but still can be significant in latency as one appended stage [20].

In this work, we propose a solution to exploit the repeated data pattern by leveraging the indication of data smoothness. The prediction would generate two types of data: zeros and non-zeros. The zero represents the cases in which the prediction error is no greater than one unit of the error bound (eb) regarding the original value. And the non-zeros represent the otherwise, which are expected scarce and scattered across the data. With such dichotomy, we consider that the quant-codes are smooth when they are *locally* continuous with zeros and propose to use run-length encoding to utilize the data patterns that may exist.

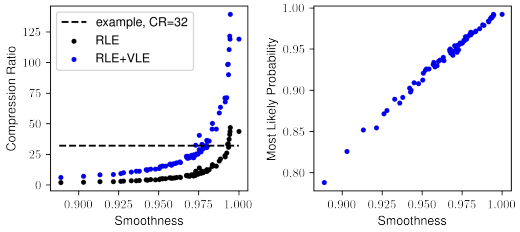
Optionally, we can append another stage of Huffman encoding, which can further bring a steady 2× to 3× ratio gain beyond RLE. Our design goal is to push the compression ratio beyond the original 32× for `float` (or 64× for `double`); considering the throughput, compressing the metadata of RLE output is optional and by default disabled in GPU processing. We further expand the criteria and the uses of RLE in §III-B.

B. Smoothness and Run-Length Encoding

RLE, first introduced in [21], is a form of lossless compression, in which sequences of consecutive same-value data elements are stored as value-count tuples. Such kinds of sequences are called *runs* of data. For example, "aabcccccaa" is stored as "(a,2)(b,1)(c,5)(a,2)". RLE's continuity in the same values can be seen as a simplistic pattern-finding method; its regular access



(a) Smoothness against encoding distance (`CESM FSDSC` at 1e-2). The yellow line denotes linear regression of variances at distances.



(b) Smoothness-Probability of the most likely symbol relationship.

Fig. 2: Smoothness of prequantized data and quant-code, and smoothness-probability of the most likely symbol relationship can help determine when to use RLE. For example, a threshold compression ratio can be set to 32 to find the desired smoothness, and the smoothness can be directed to the probability of the most likely symbol.

when checking the following values can contribute to the high throughput on GPU. Without the prefix property of Huffman coding, the length of runs must be recorded as metadata, introducing overhead. The overhead immediately raises questions of (1) how to model the compressibility and (2) when to use RLE. We identify that madogram, a variogram [22] variant, and histogram can help make the decision—madogram suffices to reveal the reason RLE can perform well, and histograming is easy to conduct and coincide can converge to indicate the chances of performing RLE.

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

B.1) Estimation from Histogram We first discuss the data features that motivate VLE, based on the histogram. The entropy value of the histogram is calculated as $H(X) = -\sum p_i \log_2 p_i$, where p_i is the probability of the i-th symbol. We use $\langle b \rangle$ to denote the average bit-length of Huffman codeword, and $\langle b \rangle_{\text{RLE}}$ to denote that of RLE. We use R (redundancy) to denote the discrepancy between $\langle b \rangle$ and entropy $H(X)$ (i.e., $R = \langle b \rangle - H(X)$). Denote the probability of the most likely symbol by p_1. With p_1, we can estimate the upper and lower bounds of R, R^+ and R^-, respectively. When $p_1 > 0.4$, R^- is given by $1 - H(p_1, 1-p_1)$, where $H(p_1, 1-p_1) = p_1 \log_2 \frac{1}{p_1} + (1-p_1) \log_2 \frac{1}{1-p_1}$ [23]. The upper bound is given by $R^+ = p_1 + 0.086$ (no restriction) [24]. Therefore, without building Huffman tree, we can estimate the upper and lower bound of $\langle b \rangle$ with R^+ and R^-, respectively. A lower bit-length leads to a higher compression ratio. We expect to use RLE when $\langle b \rangle_{\text{RLE}} \leq \langle b \rangle$. And we also use the upper bound of $\langle b \rangle$ to estimate the lowest gain of an additional Huffman coding after RLE (see TABLE IV).

B.2) Modeling RLE Compressibility Due to the obvious overhead from storing the metadata, RLE would do when the penalty caused by value change is sufficiently low. And the low penalty can be translated to the data being *smooth enough*. However, the histogram cannot reflect the smoothness because a locally smooth datum can have a similar histogram to a rougher one.

The method of variogram [22] inspires us to derive a new scheme to measure the smoothness. Variogram (i.e., general-purpose multidimensional data variance) is a very effective metrics to reveal a variance-distance relationship in spatial data based on sampling. Its theoretical form is

$$2\gamma(\mathbf{s}_1, \mathbf{s}_2) = \text{var}\big(Z(\mathbf{s}_1) - Z(\mathbf{s}_2)\big) = E\left[\big(Z(\mathbf{s}_1) - Z(\mathbf{s}_2)\big)^2\right],$$

where $Z(\mathbf{s})$ is a spatial random field. Considering the encoding iteration is unidimensional, we substitute the power description $\big(Z(\mathbf{s}_1) - Z(\mathbf{s}_2)\big)^2$ with the absolute difference $|Z(\mathbf{s}_1) - Z(\mathbf{s}_2)|$ to form *madogram*. Also note that an *RLE run* discontinues when the value differs from the current one (v_{this}), we further adjust the absolute difference to *binary* difference, defined as

$$\text{binary variance} = \begin{cases} 0 & v_{\text{this}} = v_{\text{next}} \\ 1 & v_{\text{this}} \neq v_{\text{next}} \end{cases},$$

regardless of the distance. And the expected value (defined below) is interpreted as *RLE roughness*; then, *smoothness* is naturally (1 − roughness). Given the $\mathcal{O}(n^2)$ nature of enumerating pairwise variances, the empirical madogram method would do with an offline sampling scheme. More specifically, given a sufficiently large number sampling number N and a maximum distance of measurement $D_{\text{max}} = 200$, we form the pair $(a, a+d)$, where a is randomly selected from the whole data field, and $d = \text{rand}(1, 200)$ (suppose $(a+d)$ is in the data range). The summed variance of each distance is averaged by its corresponding count. The averaged binary variance $v(d)$ remarks the roughness, and $1 - v(d)$ the smoothness.

We first show the madogram of the prequantized original data and quant-code in absolute difference and the madogram of quant-code using binary variance in Fig.2. Fig.2a shows that quant-code is smoother with less variance than the prequantized original data. That is to say, the prediction-quantization scheme uses much less information to represent the data change and therefore helps achieve a high rate of data reduction. The third part of Fig.2a indicates that quant-code can forward-encode from a fixed starting point with almost equal roughness at an arbitrary distance from the starting point. Hence, at a stable rate of roughness, it is worth performing RLE. The next step is to determine the threshold. Fig.2b shows that with binary variance we can relate (1) data continuity/smoothness and the compression ratio (CR) and (2) data smoothness and the probability of the most likely symbol (p_1). With (1), for example, a simplistic case is to set a CR threshold at 32 and look up the smoothness for RLE or RLE+VLE. With (2), p_1 can determine compression ratios of both the proposed RLE workflow and VLE in CUSZ and be used to select from the two workflows. For example, FSDSC has an RLE-CR above 25 while CUSZ-VLE-CR is 26× to 29×. Note that 1) there is an overhead of chunkwise metadata from CUSZ-VLE (the final CR is 23.88) and 2) additional VLE after RLE can provide steady 3× more CR in general (estimated from the corresponding histogram), making the accumulated CR above 70× in this case.

We can use the mapping to determine when to use RLE. For example, we can set a threshold of 32×, the highest possible compression ratio obtained from Huffman coding, and find the empirical smoothness hence the proper p_1. Or, more conveniently, we can give a practical conclusion: when Huffman is likely to achieve an average bit-length lower than 1.09, we can use RLE.

IV. Performance Optimization

In this section, we present our optimization strategies, featuring (1) performance improvements on *each* kernel in compression, and (2) a new high-performance Lorenzo reconstruction kernel in decompression.

A. Compression Optimization

A.1) Dual-Quantization In Original SZ, in situ data reconstruction is required during compression; such reconstruction is precisely the same as decompression-time one. Specifically, a data item d is reconstructed from the known previous items, based on their known predecessors recursively. Such process 1) makes a compression-time reconstructed item in place of the original, iteratively, and therefore 2) causes loop-carried *read-after-write* dependency. And quant-code that controls the error compensation is one outcome of this process, which would be encoded further. To get quant-code q and use it to reconstruct data d in an arbitrary iteration, SZ needs to go through the following data transformation. I) the prediction error is from $e^\circ = d - p^\circ$, where p° denotes predicted value. II) with respect to the user-input error bound eb, e° is integerized to quant-code q° with rounding. III) $e^{\circ\star}$ is transformed from q° and serves as *error compensation* to p° such that $d^{\circ\star} = e^{\circ\star} + p^\circ$

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

approximates d with a loss that is no greater than $1 \times eb$, ensuring error-boundness.

A.1.a Generality cuSZ work [13] resolved the tight RAW dependency by foregoing integerization. Its essential technique is two-phase *dual-quant*, including

prequant With integerization $d^\circ = \text{round}\langle d/(2 \cdot eb)\rangle$, where d is the original, the compression error $|d - d^\circ \cdot 2eb| < eb$ is guaranteed.

postquant The difference between the prediction p° and the target integer value d° is rendered as $\delta^\circ = d^\circ - p^\circ$. The quant-code q° is equivalent to but typecasted from δ°.

Note that δ is the counterpart of error compensation e; the mark δ is deliberately chosen for there is no further error introduced after prequant. Unlike error compensation $e^\circ \neq e^{\circ\star}$ in Original SZ, $d^\circ \to \delta^\circ$ and $\delta \equiv q$ guarantees the reconstructed $d^{\circ\star} = p^\star + \delta = d^\circ$. Therefore, it obviates the calculation after q°. And d°, equivalently the known reconstructed data, is ready after prequantization, eliminating the loop-carried RAW dependency parallelizing prediction-quantization.

A.1.b Computational Efficiency Computation-wise, it is worth noting that cuSZ's dual-quant method continues working for cuSZ+ with Lorenzo predictor and a modified quantization scheme. According to Tao et al. [9], the general-form Lorenzo predictor is given by $\sum_{0\le k_{1\dots m}\le n}^{k_{1\dots d}\neq \mathbf{0}} \left\langle \prod_{j=1}^{m} (-1)^{k_j+1} \binom{n}{k_j} \right\rangle \cdot d_{x_1-k_1,\cdots,x_d-k_d}$, where $\sum_{0\le k_{1\dots m}\le n}^{k_{1\dots d}\neq \mathbf{0}} \left\langle \prod_{j=1}^{m} (-1)^{k_j+1} \binom{n}{k_j} \right\rangle = 1$, that is, throughout the prediction, coefficients sum to 1. Thanks to *dual-quant*, the integer coefficient is a set $C = \{c \mid c \in \mathbb{N}\}$ that is closed under addition, subtraction, and multiplication (i.e., no division involved). Moreover, integer-based data reconstruction is precise and robust with respect to machine ϵ. In addition, integer summation is considered as commutative, i.e., $a \oplus b = b \oplus a$ and $\oplus$ becomes integer addition. Thus, given arbitrary numbers of integers, adjusting the addend order results in no difference in sum. This property guarantees that our proposed fine-grained Lorenzo reconstruction (will be discussed in §IV-B) can reorder the prediction computation.

A.2) Compression Kernel Enhancement We mainly focus on optimizing two kernels: Lorenzo construction and the Huffman encoding. Besides coalescing interaction with DRAM/shared memory, we propose to adopt two primary strategies to increase the number of thread blocks (or warps) that can run concurrently within one SM (streaming multiprocessor). I) We coarsen the granularity by assigning more data items to one thread. For example, a 16×16 2D data chunk is equally split into two groups, each traversed in consecutive 8 items along y-direction. Note that the data-thread mapping my differ from the coalescing load & store. Then, it is possible to launch more warps per SM toward higher occupancy. II) According to the extrapolative prediction form, neighboring data items are reused, with the index difference being 1. We perform in-warp shuffle to exchange data. This strategy can decrease the shared memory use to launch more warps in the same SM. The comparison between the kernels of cuSZ and our cuSZ+ is shown in TABLE VI.

Algorithm 1: Lorenzo construction and reconstruction. Yellow-highlight marks the modified quantization scheme. Blue-highlight marks partial-sum based reconstruction.

```
1  (for all fp-prepresented data item d)                    ▷ compression
2  d° ← (d).divided_by(2×eb)                                ▷ prequant, barrier
3  p° ← ℓ(d°_SR),   δ° ← p° − d°
4  if δ° < cap/2 ≡ radius r then                            ▷ postquant
5      q° ← (δ°).to_int() + r                               ▷ captured, to lossless-compress
6  else
7      outlier ← δ°                                         ▷ remaining fp presence
8  end if
   pΣ to denote inclusive partial-sum                       ▷ decompression
   (for all q• ≡ q°)
9  q′ ← (q• ⊕ outlier) − r                                  ▷ fuse quant. and outlier
10 d• ← pΣ_x q′ only if dim. x exists                       ▷ barrier
11 d• ← pΣ_y (pΣ_x q′) only if dim. x, y exist              ▷ barrier
12 d• ← pΣ_z (pΣ_y (pΣ_x q′)) only if dim. x, y, z exist    ▷ barrier
13 output ← d• · (2×eb)
```

B. Decompression Optimization

B.1) The Modified Quantization Scheme We first modify the quantization scheme of compression to eliminate the divergence in the reconstruction procedure, enabling fine-grained parallelism on GPU architectures. In cuSZ, if the error compensation δ° at the prequantized d° is out-of-range, d° is otherwise stored as an `outlier`, with 0 stored as the placeholder. In cuSZ+, if the compensation δ° is out-of-range, δ° is stored as an `outlier` (line 7 in Algorithm 1), while the quant-code remains stored in the same way (line 5 in Algorithm 1). Then, the outlier and the quant-code are further processed, i.e., stored directly and compressed in a lossless manner, respectively. During the decompression, the outlier and the quant-code are extracted from the compression archive as it is and from lossless decoding, respectively.

During decompression, the original cuSZ accesses `outlier` when hitting 0 (the placeholder). However, cuSZ's coarse-grained reconstruction only exploits the parallelism of multithreading but rarely considers dependency, memory access pattern, and computational efficiency. In comparison, by modifying quantization, cuSZ+ fuses the quant-code and the outlier before reconstruction (line 12 in Algorithm 1). In such a manner, we conduct the reconstruction from the error compensation $\delta^\bullet$ without any stall, hence eliminating the dependency that exists in cuSZ's coarse-grained reconstruction.

B.2) Partial-Sum Lorenzo Reconstruction The default 1D to 3D first-order Lorenzo predictors are put as follows,

$$
\begin{aligned}
p_{[x]} &= +\, d_{[x-1]} \\
p_{[y,x]} &= -\, d_{[y-1,x-1]} + d_{[y-1,x]} + d_{[y,x-1]} \\
p_{[z,y,x]} &= +\, d_{[z-1,y-1,x-1]} - d_{[z-1,y-1,x]} - d_{[z,y-1,x-1]} + d_{[z,y-1,x]} \\
&\quad - d_{[z-1,y\ \ ,x-1]} + d_{[z-1,y\ \ ,x]} + d_{[z,y\ \ ,x-1]}
\end{aligned}
$$

In the following text, we use 2D form to demonstrate the expression. Let r denote quantization radius. In decompression, $d^\bullet = p^\bullet + q^\bullet - r$; with $q' = q^\bullet - r$, it becomes $d^\bullet = p^\bullet + q'$. Considering that we initially predict from zeros, the first pre-

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

dicted item is $d^{\bullet}_{[0,0]} = q'_{[0,0]}$. We then observe that an arbitrary item $[y, x]$ is predicted as $\sum_{j=0}^{y}\sum_{i=0}^{x} q'_{[j,i]}$. We give a proof by induction on $[y, x]$, as $d^{\bullet}_{[y+1,x+1]}$ equals to

$$-\sum_{j=0}^{y}\sum_{i=0}^{x} q'_{[j,i]} + \sum_{j=0}^{y}\sum_{i=0}^{x+1} q'_{[j,i]} + \sum_{j=0}^{y+1}\sum_{i=0}^{x} q'_{[j,i]} + q'_{[y+1,x+1]}$$
$$= \sum_{i=0}^{y} q'_{[j,x+1]} + q'_{[y+1,x+1]} + \sum_{j=0}^{y+1}\sum_{i=0}^{x} q'_{[j,i]} = \sum_{j=0}^{y+1}\sum_{i=0}^{x+1} q'_{[j,i]}.$$

An intuitive demonstration for 2D case is shown in Fig.3a, with canceling the joint summation. Similarly, the computation for N-D case can be done by N-D partial-sum as follows.

B.2.a Computation We define N-D partial-sum of x till index $[k_N, \ldots, k_2, k_1] \in \mathbb{N}^N$ as

$$p\Sigma(x; k_N, \ldots, k_2, k_1) = \sum_{i_N=0}^{k_N} \cdots \sum_{i_2=0}^{k_2}\sum_{i_1=0}^{k_1} x_{[i_N,\ldots,i_2,i_1]},$$

where $p\Sigma$ is a variadic operator for any N. We can decompose it to N-pass 1-D partial-sums, as

$$p\Sigma(x; k_N, \ldots, k_2, k_1) = p\Sigma(p\Sigma\big(x; k_{N-1}, \ldots, k_2, k_1\big); k_N)$$
$$= p\Sigma(p\Sigma\big(\cdots p\Sigma\Big(p\Sigma\Big(x; k_1\Big); k_2\Big)\cdots; k_{N-1}\big); k_N).$$

That is, the output of a partial-sum on x_m-direction is the input of that on $x_{(m+1)}$-direction. Given the problem size $(X_N, \ldots, X_2, X_1)$, where $k_{(\cdot)} \leq X_{(\cdot)}$, a pass along $x_{(\cdot)}$ features the degree of independence (hence the maximum possible parallelism) equal to $\prod_{i\neq(\cdot)} X_i$.

We give an example for 2D case of size b_y-by-b_x. The first partial-sum along x is performed through indices $[y, 0 \ldots b_x]$, given any y; the partial-sum at $[y, x]$ is $p\Sigma\left(q'|_y; x\right) = \sum_{i=0}^{x} q'_{[y,i]}$. The second partial-sum along y is performed through indices $[0 \ldots b_y, x]$, given any x; the partial-sum at $[y, x]$ is $p\Sigma(q'|_{y,x}; y, x) = p\Sigma\left(p\Sigma\left(q'|_y; x\right)|_x; y\right)$. Their parallelism degrees are b_y and b_x, respectively. An illustration of this parallelized computation is given in Fig. 3b.

B.3) Implementation Detail We conduct partial-sum in a chunk-wide manner, as the compression is the same way: no inter-chunk dependency. To illustrate the effectiveness of this proposed solution, we first have a proof-of-concept implementation using shared memory and assign 1 item to 1 thread. Compared to the coarse-grained implementation in CUSZ, this one improves in performance notably (see "naïve" against "CUSZ" in TABLE II). Also, note that the new reconstruction kernel has high performance, similar to the fine-grained construction kernel in CUSZ (see "CUSZ" column). Our proposed fine-grained solution for CUSZ+ exhibits higher resource utilization than the coarse-grained parallelization does in CUSZ.

It is worth noting that the arithmetic intensity of the Lorenzo reconstruction kernel is relatively low with linear algorithmic time complexity, which tends to be memory-bound. Thus, we consider the two optimizations over the naïve implementation:

1) We tune the access pattern to ensure the coalesced load/store from/to global memory.
2) We increase the *sequentiality* to each thread to balance load/store and computation.

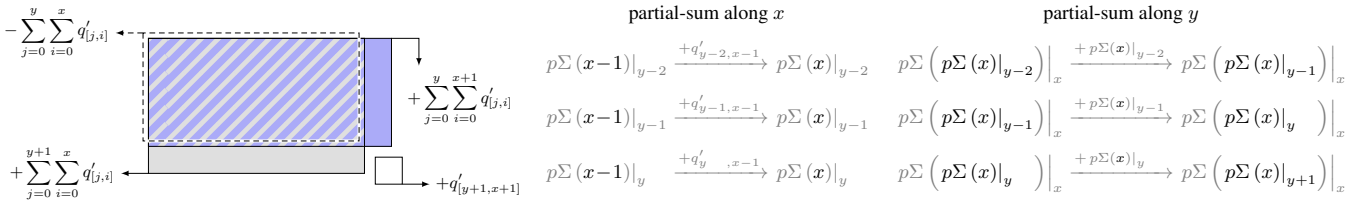
With these optimizations, we form a different thread block management, considering that there is no canonical way to map the thread to the data throughout the dimensions; instead, the data-thread mapping is more bottom-up. Among all the abstraction levels from warp ("SIMDness") to grid, warp convergence is the first concern to address. On the other hand, our problem features more "streaming"-style memory access given the linear processing time; there is not much of locality that we can exploit. This is distinct from "persisting" style found in, e.g., `gemm`, whose algorithmic complexity is much above $\mathcal{O}(n)$. Therefore, the workflow path of *global-memory* → *register* → *shared-memory* is neither efficient nor necessary, given that the register file can hold a certain amount of data and perform in-warp operations.

B.3.a 1D Implementation We attribute the chunkwise partial-sum to 1D `BlockScan` and implement it using `NVIDIA::cub` library. Warp-striped load/store from/to global memory is used to ensure the coalesced read/write. We test different sequentialities of each thread, with warp-/block-wide scan-and-sync employed. We use 256 as 1D chunk size in CUSZ+ and the `vx` field in 1D HACC dataset as an example in TABLE II.

B.3.b 2D Implementation There is no such direct abstraction, however, for higher-dimensional partial-sum—the multidimensional use of `cub` is internally linearized to 1D. Thus, we handcraft the 2D reconstruction kernel. We use 16×16 as 2D chunk size in CUSZ+ (the same as CUSZ). The 1-to-1 thread-data binding and unconditional use of shared memory (the only explicit cache in GPUs) incurs both low computational utilization of each thread (and hence warp) and underuse of register file. In-warp operations such as `__shfl_up_sync` allow accessing the register in the same warp and hence data exchange without using shared memory. We then specify x-direction as warp-shuffling space while setting sequentiality to y-direction. Each thread holds an $n_{(2)}$-length thread-private array `tp[]`, in which a fragment of partial-sum is sequentially and trivially done. $\frac{16}{n_{(2)}}$ threads are needed along y-direction to complete a size-16 partial-sum. Of the same y, all threads except the last one propagate the last-element value in `tp[]` to all the elements in the next thread's private array, sequentially, using shared memory to exchange. We identify the sequentiality of 8 results in the optimal throughput under such thread block configuration—a $(16, 2, 1)$-block size comprises a warp—at 254.2 GB/s on V100 and 508.6 GB/s on A100 with testing on a sample CESM field. Note the throughputs are comparable to those from the high-throughput 1D kernel.

B.3.c 3D Implementation The 3D problem has an addition to the 2D implementation. Right after the same procedure in the 2D case, we append an x-z transposition of the x- and y-direction partial-sum result and repeat the previous x-direction partial-sum (with z-direction data). Based on trials, we identify that the $8\times$ sequentiality results in the best throughput. Yet, the 3D kernel, due to the longer computational process, does not achieve as high throughput as lower dimensionality. Further evaluation and analysis are listed in TABLE VII in §V.

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.



(a) Concept of reconstruction in terms of 2D partial-sum. (b) Exemplary 2-pass partial-sum computation for 2D data reconstruction.

Fig. 3: Example of 2D partial-sum computation for Lorenzo reconstruction in CUSZ+'s decompression.

throughput measured in GB/s		cuSZ [13]		ours (naïve)		ours (optim)	A100 adv. over V100
1D (HACC)	A100	-		219.8	+130%	504.5	1.64×
	V100	16.8	+1404%	252.6	+24%	313.1	
2D (CESM)	A100	-		182.1	+179%	508.6	2.00×
	V100	58.5	+239%	198.4	+28%	254.2	
3D (Nyx)	A100	-		147.9	+174%	405.1	1.70×
	V100	29.7	+492%	175.9	+35%	238.1	

TABLE II: Proof-of-concept throughput on V100 for {1, 2, 3}-D. The referenced throughput of CUSZ's Lorenzo reconstruction covers all the fields [13]. The table shows a single field (i.e., vx in HACC, CLDHGH in CESM, baryon-density in Nyx) for demonstration purpose.

V. Experimental Evaluation

This section presents our experimental setup (platforms, baselines, and datasets) and our evaluation results.

A. Experimental Setup

A.1) Evaluation Platform We conduct our experimental evaluation on two HPC systems equipped with NVIDIA Tesla V100 and A100 GPUs[3], including ThetaGPU [25] at Argonne Leadership Computing Facility and Longhorn [26] at Texas Advanced Computing Center. More details are listed below,

- **ALCF-ThetaGPU**: NVIDIA A100, SXM4 variant, CUDA 11.1
  - **DRAM** 40-GB HBM2e at 1555 GB/s (1.38× V100)
  - **compute** 19.49 FP32 TFLOPS (1.73× V100)
  - **host** AMD 7532 (32-core), 256 GB
- **TACC-Longhorn**: NVIDIA V100, SXM2 variant, CUDA 10.2
  - **DRAM** 16-GB HBM2 at 900 GB/s
  - **compute** 14.13 FP32 TFLOPS
  - **host** 2 IBM Power9 (40-core), 256 GB

A.2) Baselines We compare our CUSZ+ with multiple baselines. Specifically, 1) we compare CUSZ+ with CUSZ in terms of our optimized kernels (i.e., Lorenzo construction, Huffman encoding, Lorenzo reconstruction), 2) we compare CUSZ+ on A100 versus on V100, and 3) we compare CUSZ+'s Workflow-RLE with CUSZ's Workflow-Huffman.

A.3) Test Datasets We conduct our evaluation and comparison based on seven typical real-world HPC simulation datasets of each dimensionality, most of which are from the Scientific Data Reduction Benchmarks suite [17]. The datasets include

1) 1D HACC cosmology particle simulation [1],
2) 2D CESM-ATM climate simulation [27],
3) 3D Hurricane ISABEL simulation [28],
4) 3D Nyx cosmology simulation [29],
5) 3D seismic wave RTM data,
6) 3D hydrodynamics Miranda [30] [4], and
7) 3D Quantum Monte Carlo [31], reinterpreted from 4D.

They have been widely used in prior works [3, 4, 7, 32, 33] and are good representatives of production-level simulation datasets. TABLE III shows all 128 fields across these datasets. The data sizes for the seven datasets are 6.3 GB, 2.0 GB, 1.9 GB, 3.0 GB, 1.8 GB, 1.0 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 similarities. Moreover, when the field is too large to fit in a single GPU's memory, CUSZ+ divides it into blocks and then compresses by block.

datasets	datum size dimensions	#fields examples(s)
cosmology **HACC**	1,071.75 MB 280,953,867	6 in total x, vx
climate **CESM-ATM**	24.72 MB 1,800×3,600	77 in total CLDHGH, PHIS
climate **Hurricane**	95.37 MB 100×500×500	20 in total CLOUDf48, Uf48
cosmology **Nyx**	512 MB 512×512×512	6 in total baryon_desnity
seismic wave **RTM**	180.72 MB 449×449×235	10 in total snapshot28{0...9}0
hydrodynamics **Miranda**•	144 MB 256×384×384	7 in total density, pressure
Quantum Monte Carlo **QMCPACK**	601.52 MB 288x115x69x69	2 in total preconditioned

TABLE III: Real-world float-type datasets used in the evaluation. • Miranda (double-type) is converted to float-type for CUSZ's support. QMCPACK includes only one field but with two representations.

B. Evaluation on Compression Ratio

TABLE IV shows several cases that RLE performs better in compression ratio than CUSZ-VLE. Run-length encoding is implemented using thrust::reduce_by_key and achieves 100 GB/s throughput on V100 and slightly higher throughput on A100. The table demonstrates that RLE may replace the multi-byte VLE in the original workflow and maintain or achieve higher compression ratios; it can also be used as an additional

[3] We note that PCIe-A100 holds a marginal 30% less throughput than SXM4-A100. In this work, we use the SXM4 variant for evaluation.

[4] It is converted to float from double

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

	cuSZ+gzip (qhg) ref.	cuSZ (qh) VLE	ours RLE	ours gain	ours RLE+VLE	ours gain
AEROD_v	94.27	25.06	10.46	-	30.33	1.21×
FLNTC	56.95	23.66	8.87	-	25.35	1.07×
FLUTC	57.06	23.66	8.91	-	25.46	1.08×
FSDSC	58.30	23.88	26.10	1.09×	71.35	2.99×
FSDTOA	430.61	26.10	43.65	1.67×	119.17	4.57×
FSNSC	51.73	23.44	10.11	-	29.46	1.26×
FSNTC	60.35	23.88	12.33	-	35.50	1.49×
FSNTOAC	111.63	25.06	12.46	-	35.84	1.43×
ICEFRAC	159.18	25.31	16.57	-	50.39	1.99×
LANDFRAC	97.15	23.66	13.98	-	40.50	1.71×
OCNFRAC	89.55	23.88	11.23	-	32.55	1.36×
ODV_bcar1	189.28	25.83	37.28	1.44×	110.51	4.28×
ODV_bcar2	197.32	25.83	30.71	1.19×	89.98	3.48×
ODV_dust1	242.89	26.10	22.91	-	67.72	2.59×
ODV_dust2	319.55	26.37	24.02	-	70.98	2.69×
ODV_dust3	270.50	26.10	33.29	1.28×	98.22	3.76×
ODV_dust4	230.40	26.10	46.81	1.79×	139.27	5.34×
ODV_ocar1	65.81	24.11	41.17	1.71×	121.59	5.04×
ODV_ocar2	64.92	24.11	33.79	1.40×	98.63	4.09×
PHIS	98.86	25.06	9.51	-	28.87	1.15×
PRECSC	176.21	25.83	19.50	-	58.92	2.28×
PRECSL	142.23	25.57	15.39	-	45.69	1.79×
PSL	83.13	24.34	12.43	-	36.32	1.49×
PS	98.59	21.09	7.45	-	22.27	1.06×
SNOWHICE	144.74	25.31	15.14	-	45.53	1.80×
SNOWHLND	184.39	25.57	21.18	-	63.33	2.48×
SOLIN	430.62	26.10	43.65	1.67×	119.17	4.57×
TAUX	100.30	25.06	11.30	-	33.28	1.33×
TAUY	106.55	25.31	12.40	-	36.45	1.44×
TREFHT	82.50	24.58	8.75	-	25.12	1.02×
TREFMXAV	87.39	24.58	9.60	-	27.33	1.11×
TROP_P	93.78	24.82	11.19	-	31.40	1.27×
TROP_T	92.94	24.82	11.10	-	30.64	1.23×
TROP_Z	84.81	24.58	9.48	-	27.07	1.10×
TSMX	64.95	23.88	8.55	-	24.69	1.03×

TABLE IV: Data fields that CUSZ+ with Workflow-RLE has higher compression ratio than CUSZ with Workflow-Huffman under 10^{-2} error bound. "gain" is based on ours against (*qh*) VLE from CUSZ.

stage to VLE to get up to 5.3× compression ratio improvements over CUSZ on the tested datasets.

		V100 (GB/s) Huff/RLE	V100 (GB/s) overall	A100 (GB/s) Huff/RLE	A100 (GB/s) overall	CR
RTM #2800	ours	142.4	57.8	212.6	78.0	76.0×
	cuSZ	135.7	55.1	233.9	80.8	31.7×
CESM FSDSC	ours	104.8	47.7	162.4	57.8	26.1×
	cuSZ	146.3	54.8	146.4	55.5	23.0×
Nyx baryon	ours	159.1	64.1	214.5	91.2	122.7×
	cuSZ	130.8	58.9	234.2	94.8	31.0×

TABLE V: Throughputs (in GB/s) of CUSZ+ (based on RLE) and CUSZ (based on Huffman coding) on example RTM, CESM, and Nyx fields for a demonstration purpose.

TABLE V shows the throughputs of CUSZ+ using the Workflow-RLE on the RTM, CESM, and Nyx datasets. It demonstrates that the RLE-based workflow can not only improve the compression ratio, but also maintain a comparable compression throughput. Thus, CUSZ+ can provide users flexibility between high compression ratio and performance.

C. Evaluation on Performance and Scalability

In this section, we evaluate the compression performance of CUSZ+ and compare it with CUSZ.

C.1) Evaluation on Optimized Kernels We first evaluate the performance of the majorly changed kernels in CUSZ+ and CUSZ on V100, as shown in TABLE VI. The baseline is from the evaluation results shown in the CUSZ paper [13]. The table illustrates that the performance improvements of CUSZ+'s Lorenzo construction kernels are 1.48× for 1D data, 1.09× for 2D data, and 1.45× for 3D data on average over CUSZ.

	Lorenzo comp. cuSZ	Lorenzo comp. ours	Huffman Enc. cuSZ	Huffman Enc. ours	Lorenzo decomp. cuSZ	Lorenzo decomp. ours
HACC	207.7	307.4 1.48×	54.1	58.3 1.08×	16.8	313.1 18.64×
CESM	252.1	273.9 1.09×	57.2	107.7 1.88×	58.5	254.2 4.35×
Hurricane	175.8	229.9 1.31×	55.2	111.2 2.01×	43.9	218.4 4.97×
Nyx	200.2	296.0 1.48×	58.8	120.5 2.05×	29.7	238.1 8.02×
QMCPACK	189.6	298.6 1.57×	61.0	110.8 1.82×	22.4	255.5 11.41×

TABLE VI: Performance comparison of Lorenzo and Huffman encoding kernels in CUSZ+ and CUSZ on V100. The unit is in GB/s.

Moreover, we increase the lowest throughput from 175.8 GB/s to 229.9 GB/s (+30.7%) on the tested datasets.

For Huffman encoding kernel, despite it being more latency-bound, it also suffers from non-coalescing store. Because of the variable-length encoding and bit operation spanning multiple bytes, it is impossible to make threads work in a synchronized manner (otherwise, a high synchronization overhead would be imposed). Our optimization can decrease the number of DRAM store transactions to be inversely proportional to the compression ratio. In particular, we perform a DRAM store only when a new data unit needs to be written back, which helps us achieve 1.08× to 2.05× performance gain.

The table also shows that by using the fine-grained ND partial-sum computation, CUSZ+'s Lorenzo reconstruction kernel exhibits up to 18.4× performance improvements over CUSZ's coarse-grained kernel. The 2D and 3D kernels also exhibit 4.5× and 4.6× to 7.1× performance improvements, respectively. In addition, TABLE VII shows the speedup of our optimized kernels on A100 compared to on V100.

C.2) Evaluation on Default Compression Workflow The rest of TABLE VII shows the evaluation of CUSZ+ based on the default compression workflow on both V100 and A100 with the relative error bound of 10^{-4} (with PSNRs higher than 85 dB). It illustrates the performance of our optimized compression and decompression kernels. We note that the performance improvements of the histogram kernel and the gather-outlier kernel are relatively lower than those of other compression kernels (the performance is even degraded on some datasets such as CESM and RTM) from using A100 to using V100. The degradation may be because each field of CESM-ATM and RTM is fairly small (24.7 MB and 180 MB, respectively), such that the histogram kernel (using the algorithm from [34]) and the gather-outlier kernel (using the dense-to-sparse kernel from cuSPARSE) on A100 do not maintain the same work efficiency as on V100. But note that these two kernels would not be bottlenecks [5] for a relatively large dataset (e.g., hundreds of MBs per field), which is more common in practice (e.g., HACC and Nyx).

In addition, we observe that CUSZ+'s kernels with different parallelism have different scalabilities. Specifically, the sparsity-related operation in compression can be enhanced significantly by using A100 GPU, but other compression kernels such as Huffman encoding are scaled up marginally. As a result, the overall improvement of compression performance is limited when changing from using V100 to using A100. Similarly, for decompression, although the outlier scatter operation scales

[5]The Huffman encoding kernel is the main bottleneck in the compression workflow compared to other compression kernels.

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

	V100-ours, GB/s							A100-ours, GB/s (and advantage over V100)						
size in MB	1071.8 **HACC**	24.7 **CESM**	95.4 **Hurr**	512.0 **Nyx**	180.7 **RTM**	144.0 **Mira.**	601.5 **QMC**	1071.8 **HACC**	24.7 **CESM**	95.4 **Hurr**	512.0 **Nyx**	180.7 **RTM**	144.0 **Miranda**	601.5 **QMC**
Lorenzo construct	328.3	273.9	199.0	296.0	193.1	289.3	298.6	501.1 1.53×	466.8 1.70×	429.0 2.16×	481.3 1.63×	422.7 2.19×	480.7 1.66×	492.9 1.65×
gather outlier	221.4	160.6	251.1	238.0	249.7	228.6	261.2	324.8 1.47×	151.4 0.94×	284.2 1.13×	334.9 1.41×	221.6 0.89×	336.0 1.47×	266.2 1.02×
histogram	565.9	356.5	438.4	372.4	573.6	489.8	724.3	923.5 1.63×	409.8 1.15×	681.2 1.55×	870.2 2.34×	793.9 1.38×	714.9 1.46×	569.7 0.79×
Huffman encode	58.3	107.7	111.2	120.5	123.2	161.1	110.8	174.6 2.99×	121.6 1.13×	206.0 1.85×	217.2 1.80×	202.2 1.64×	201.6 1.25×	198.4 1.79×
overall, compress	37.8	44.8	49.3	53.9	52.5	62.2	56.9	84.1 2.27×	51.5 1.15×	82.2 1.67×	92.4 1.72×	76.4 1.46×	87.6 1.41×	79.5 1.40×
Huffman decode	42.1	37.9	45.8	66.8	48.9	42.7	44.6	48.5 1.15×	26.6 0.70×	51.8 1.13×	91.2 1.37×	56.0 1.15×	50.1 1.17×	49.0 1.10×
scatter outlier	225.0	334.8	628.1	359.7	440.2	679.1	347.1	658.4 2.93×	630.2 1.88×	918.3 1.46×	797.4 2.22×	906.6 2.06×	1066.8 1.57×	782.8 2.26×
Lorenzo reconstruct	308.7	267.0	200.1	251.7	201.3	245.3	255.5	504.4 1.63×	495.3 1.86×	345.5 1.73×	398.6 1.58×	335.6 1.67×	386.9 1.58×	384.0 1.50×
overall, decompress	31.8	30.2	35.2	46.0	36.1	34.5	34.2	41.4 1.30×	24.3 0.80×	43.0 1.22×	67.9 1.47×	45.6 1.26×	42.6 1.23×	41.2 1.20×

TABLE VII: Evaluation of cuSZ+ using default compression workflow (Lorenzo and multi-byte VLE) with relative error bound of 10^{-4} on V100 and A100: breakdown throughput of compression subprocedures.

naturally and shows a speedup of higher than 2×, the multi-byte Huffman decoding exhibits a stagnation in scaling up, resulting in a marginal improvement of the overall decompression performance.

VI. Related Work

Compression for scientific datasets has been studied for years to reduce the storage burden and I/O overhead. Scientific data compression techniques fall into two classes: lossless compression and lossy compression. The former includes the generic lossless compressors such as Zlib [35] and Zstd [18], as well as the specific algorithm designed for floating-point values such as FPZIP [36] and FPC [37]. The lossless compressors, however, all suffer from very low compression ratios (generally 2:1 or even lower [6]) because of the somewhat random ending mantissa bits in the floating-point representation.

Lossy compressors have been studied for decades. The traditional lossy compressors (such as JPEG [38] and JPEG2000 [39]) are designed for 2D images, which are not suitable for scientific datasets. The key reason is that the traditional lossy compressors focus on the visual quality while scientific applications care more about the post hoc analysis results beyond the simple visualization purpose.

To address the above-mentioned gap, error-bounded lossy compressors [8, 9, 10, 11, 40] have been proposed for years. Based on their design principle, they can be split into two categories - prediction-based model [3, 8, 9] and transform-based model [10, 40, 41]. The typical example in the former category is SZ, which supports different categories of errors to control data distortion, such as absolute error bound, relative error bound, and peak signal-to-noise ratio (PSNR). The typical example in the latter category is ZFP, which supports absolute error bound and precision mode[6].

However, all the above existing lossless and lossy compressors cannot run on GPUs directly. Recently, both the SZ team and the ZFP team released their CUDA versions, called cuSZ [13] and cuZFP [14], respectively. Both versions provide much higher throughputs for compression and decompression compared with their CPU versions. Compared with cuSZ, cuZFP provides slightly higher compression throughput, but it only supports fixed-rate mode, significantly limiting its adoption in practice. In comparison with the two existing GPU-supported compressors, our designed new compression method is aware of the compressibility of the datasets, such that it can adopt the run-length encoding (RLE) method to significantly improve the compression ratios when needed.

[6]In the precision mode, users can use an integer number to control the data distortion. Higher precision in value means lower data distortion.

VII. Conclusion and Future Work

In this work, we propose cuSZ+, a compressibility-aware GPU-based lossy compressor for NVIDIA GPU architectures, which can effectively improve the compression throughput over cuSZ. Specifically, we propose an efficient compression method to adaptively perform run-length encoding and/or Huffman encoding by considering data smoothness to improve the compression ratio over cuSZ. We prove that the Lorenzo reconstruction in decompression is equivalent to a multidimensional partial-sum computation and develop an efficient fine-grained Lorenzo reconstruction algorithm on GPUs. Moreover, we carefully optimize cuSZcompression kernels by leveraging different techniques for CUDA architectures. Finally, we evaluate cuSZ+ using seven real-world HPC application datasets on the most advanced GPUs (V100 and A100) and compare with cuSZ, the GPU-centric error-bounded lossy compressor. Experiments show that our cuSZ+ improves cuSZ's decompression kernel throughput by up to 1.6× with the same compression quality on state-of-the-art GPUs, including A100.

In the future, we plan to optimize the performance of decompression further, implement other data prediction methods such as linear-regression-based predictors 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 OAC-2042084, OAC-2034169, OAC-2003709, and CCF-1619253.

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] S. Habib *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 *et al.*, "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, 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.

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

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

[7] F. Cappello *et al.*, "Use cases of lossy compression for floating-point data in scientific data sets," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, 2019.

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

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

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

[11] 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.

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

[13] J. Tian *et al.*, "Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 3–15.

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

[15] J. Tian *et al.*, "Revisiting huffman coding: Toward extreme performance on modern gpu architectures," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Portland, OR, USA, May 17-21, 2021*, IEEE, 2021, pp. 881–891.

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

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

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

[19] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, IEEE, 2021, pp. 1643–1654.

[20] *Nvcomp*, https://github.com/NVIDIA/nvcomp, (Accessed on 05/19/2021).

[21] A. H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," *Proceedings of the IEEE*, vol. 55, no. 3, pp. 356–364, 1967. DOI: 10.1109/PROC.1967.5493.

[22] N. Cressie and D. M. Hawkins, "Robust estimation of the variogram: I," *Journal of the International Association for Mathematical Geology*, vol. 12, no. 2, pp. 115–125, 1980.

[23] O. Johnsen, "On the redundancy of binary huffman codes (corresp.)," *IEEE Transactions on Information Theory*, vol. 26, no. 2, pp. 220–222, 1980, ISSN: 1557-9654. DOI: 10.1109/TIT.1980.1056158.

[24] R. Gallager, "Variations on a theme by huffman," *IEEE Transactions on Information Theory*, vol. 24, no. 6, pp. 668–674, 1978. DOI: 10.1109/TIT.1978.1055959.

[25] *Theta/thetagpu - argonne leadership computing facility*, https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview, (Accessed on 05/21/2021).

[26] *Longhorn - texas advanced computing center*, https://www.tacc.utexas.edu/systems/longhorn, (Accessed on 03/15/2021).

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

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

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

[30] Miranda Radiation Hydrodynamics Data, https://wci.llnl.gov/simulation/computer-codes/miranda, Online, 2019.

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

[32] 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.

[33] X. Liang *et al.*, "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.

[34] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on gpu," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, 2013.

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

[36] 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.

[37] 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.

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

[39] D. Taubman and M. Marcellin, *JPEG2000 image compression fundamentals, standards and practice: image compression fundamentals, standards and practice*. Boston, MA, USA: Springer Science & Business Media, 2012, vol. 642.

[40] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart," in *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Hyderabad, India: IEEE, 2015, pp. 914–922.

[41] J. Clyne, P. Mininni, A. Norton, and M. Rast, "Interactive desktop analysis of high resolution simulations: Application to turbulent plume dynamics and current sheet formation," *New Journal of Physics*, vol. 9, no. 301, pp. 1–29, 2007.

Authorized licensed use limited to: Washington State University. Downloaded on October 18,2021 at 17:47:53 UTC from IEEE Xplore. Restrictions apply.