CEAZ: Accelerating Parallel I/O via Hardware-Algorithm Co-Designed Adaptive Lossy Compression

Chengming Zhang
Washington State University
Pullman, WA, USA
chengming.zhang@wsu.edu

Sian Jin
Washington State University
Pullman, WA, USA
sian.jin@wsu.edu

Tong Geng
Pacific Northwest National Laboratory
Richland, WA, USA
tong.geng@pnnl.gov

Jiannan Tian
Washington State University
Pullman, WA, USA
jiannan.tian@wsu.edu

Ang Li
Pacific Northwest National Laboratory
Richland, WA, USA
ang.li@pnnl.gov

Dingwen Tao*
Washington State University
Pullman, WA, USA
dingwen.tao@wsu.edu

ABSTRACT

As HPC systems continue to grow to exascale, the amount of data that needs to be saved or transmitted is exploding. To this end, many previous works have studied using error-bounded lossy compressors to reduce the data size and improve the I/O performance. However, little work has been done for effectively offloading lossy compression onto FPGA-based SmartNICs to reduce the compression overhead. In this paper, we propose a hardware-algorithm co-design for an efficient and adaptive lossy compressor for scientific data on FPGAs (called CEAZ), which is the first lossy compressor that can achieve high compression ratios and throughputs simultaneously. Specifically, we propose an efficient Huffman coding approach that can adaptively update Huffman codewords online based on codewords generated offline, from a variety of representative scientific datasets. Moreover, we derive a theoretical analysis to support a precise control of compression ratio under an error-bounded compression mode, enabling accurate offline Huffman codewords generation. This also helps us create a fixed-ratio compression mode for consistent throughput. In addition, we develop an efficient compression pipeline by adopting cuSZ's dual-quantization algorithm to our hardware use cases. Finally, we evaluate CEAZ on five real-world datasets with both a single FPGA board and 128 nodes (to accelerate parallel I/O). Experiments show that CEAZ outperforms the second-best FPGA-based lossy compressor by 2.3× of throughput and 3.0× of ratio. It also improves MPI_File_write and MPI_Gather throughputs by up to 28.9× and 37.8×, respectively.

CCS CONCEPTS

• **Computer systems organization → Reconfigurable computing**.

KEYWORDS

Lossy compression; parallel I/O; scientific data; co-design.

*Corresponding author: Dingwen Tao, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99163, USA.

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06.
https://doi.org/10.1145/3524059.3532362

ACM Reference Format:
Chengming Zhang, Sian Jin, Tong Geng, Jiannan Tian, Ang Li, and Dingwen Tao. 2022. CEAZ: Accelerating Parallel I/O via Hardware-Algorithm Co-Designed Adaptive Lossy Compression. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524059.3532362

1 INTRODUCTION

Today's HPC applications can generate large volumes of scientific data for post-hoc analysis and visualization. However, since the development of storage and networking hardware is much slower than that of computing power and memory capacity [7], the I/O and network bandwidth are becoming the main bottlenecks for HPC applications to achieve high performance on a large scale. I/O and communication costs can quickly overwhelm the overall performance as parallel computers grow towards exascale. For instance, a well-known cosmological simulation code Nyx [2] can generate up to 2.8 TB of data for a single snapshot under a simulation resolution of 4096^3, requiring to save a total of 2.8 PB data, when running the simulation for 5 times with 200 snapshots dumped per run.

Such a large amount of data is often generated in a parallel manner from a scaling number of ranks, on which each holds a proportion of the data and must introduce an extra collective communication to dump the entire snapshot to the file system. This process takes an unprecedented challenge to I/O bandwidths and storage systems on today's HPC systems [7, 30, 54, 55]. Therefore, it is urgent to develop effective data reduction methods to reduce the size of data movement between memories and storage systems such as parallel file systems.

One of the most effective ways to address this challenge is using data compression. The data partition in each rank is compressed before sending it to the storage system via an interconnected network. This can reduce both I/O overhead and storage consumption. However, traditional lossless compression can only provide a limited compression ratio to the scientific dataset by usually up to 2× [42]. Thus, error-bounded lossy compressors such as SZ [10, 30, 46], ZFP [31], and MGARD [1] have been developed to provide a much higher compression ratio while only introducing controllable distortion of data. Many prior studies have demonstrated the effectiveness of using those error-bounded lossy compressors for scientific data reduction [7, 10, 18, 24, 25, 30, 31, 34, 35, 46, 47] and improving I/O performance. [16, 39, 56].

While error-bounded lossy compressors on CPUs can provide a high compression ratio, their low throughputs unavoidably cause relatively high performance overheads to applications, which often offset the performance benefit from saving and loading compressed data of less sizes. Thus, we need to develop a high-throughput lossy compressor to effectively accelerate parallel I/O for HPC applications. Recently, all SZ, ZFP, and MGARD teams started to develop and release their GPU and/or FPGA implementations. On one hand, GPU's massive SIMT parallelism enables high throughput. However, during the lossless compression step of SZ algorithm, Huffman encoding and decoding [22] results in a random memory access pattern [49]. This causes serious divergence issues, inevitably leading to low GPU memory bandwidth utilization and performance. On the other hand, FPGAs offer many advantages, such as configurability, high energy efficiency, low latency, and low price [15], and have been a viable and popular option at scales for smart network interface cards (SmartNICs) [13, 40], which are being increasingly used in data centers to offload networking functions from host processors [21]. Thus, this makes FPGA-based SmartNICs ideal platforms to offload compression and accelerate I/O.

The state-of-the-art FPGA-based lossy compressor is only capable of about 8 GB/s throughput [44], which is still much lower than the throughput of PCIe3/4 and InfiniBand. This precludes their use in real application scenarios. There are several challenges to implementing a high-throughput lossy compression on FPGAs with a relatively high compression ratio: (1) Most of lossy compression algorithms involve multiple stages which have strong data dependency. (2) It is infeasible to simply add more compression pipelines on FPGAs with limited resources. Thus, we need to deeply optimize the algorithm to effectively utilize the hardware resources.

To address these challenges, in this work, we focus on designing an efficient lossy compression algorithm that is suitable for FPGA hardware, and offload it onto FPGA-based SmartNICs to accelerate parallel I/O. Specifically, we propose a hardware-algorithm co-designed efficient and adaptive lossy compressor (zip) (CEAZ)[1], which is the first lossy compressor to achieve high compression ratio and throughput simultaneously. CEAZ adopts a dual quantization strategy [49] to completely remove data dependency and instantiates multiple pipelines to process input data in parallel. Unlike cuSZ that implements dual-quantization using massive GPU threads, we implement the dual quantization in CEAZ using a pipelined manner, which is more suitable for FPGA architecture. Moreover, different from existing FPGA-based lossy compressors such as GhostSZ [57] and waveSZ [48] that statically build trees in Huffman coding for every data chunk, CEAZ dynamically determines whether to update codewords by building a new tree or use previous/offline codewords according to the distribution of current symbol frequencies, leading to a high throughput. Thus, CEAZ can efficiently and effectively reduce the data size and significantly increase the parallel I/O performance. In addition, to the best of our knowledge, CEAZ is also the first lossy compressor to enable fixed-rate compression for prediction-based compression workflow. Our contributions are summarized as follows:

- We propose an efficient Huffman coding approach that adaptively updates Huffman codewords online based on our offline Huffman codewords, which are generated from a variety of representative scientific datasets. It can reduce the data dependency in Huffman coding and dramatically improve the compression throughput on datasets of various sizes.
- We derive a theoretical analysis to support a precise control of compression ratio under the error-bounded compression mode, which can align quantization-code histograms of different datasets and enable an accurate generation of offline Huffman codewords. It also helps us develop a fixed-ratio compression mode, which is important to guarantee a consistent throughput in data transfer. Our work is the first prediction-based lossy compressor to enable fixed-rate compression.
- We develop an efficient compression pipeline by adapting the dual-quantization algorithm to our hardware use case.
- We evaluate CEAZ with five real-world scientific datasets in both serial and parallel processing. Experiments show that CEAZ outperforms the state-of-the-art solution by 2.3× in throughput and 3× in ratio on a single FPGA board. Moreover, CEAZ can improve the MPI_File_write and MPI_Gather throughputs by up to 28.92× and 37.8×, respectively, with 128 nodes from the Summit supercomputer.

The rest of this paper is organized as follows. In Section 2, we discuss the background and research challenges. In Section 3, we present the design of CEAZ. In Section 4, we evaluate CEAZ on six scientific datasets and present our results. In Section 5, we summarize our work and discuss future work.

2 BACKGROUND AND MOTIVATION

In this section, we first present background information about scientific data compression, FPGA-based lossy compression, and MPI communication and I/O. We then discuss the challenges and motivations for our research.

2.1 Floating-Point Data Compression

Floating-point data compression has been studied for decades. The data compressors can be split into two categories: lossless compression and lossy compression. In comparison to lossy compression, lossless compression such as FPZIP [32] and FPC [6] can only provide limited compression ratios (typically up to 2:1 for most scientific data) due to the significant randomness of the ending mantissa bits, especially for large scientific floating-point data [42].

Lossy compression, on the other hand, can compress data with little information loss in the reconstructed data. Compared to lossless compression, lossy compression can provide a much higher compression ratio while still maintaining useful information for scientific discoveries. Many lossy compressors supporting floating-point data were proposed and designed for visualization. Thus, many lossy compressors employ techniques directly inherited from lossy compression of images, such as variations of wavelet transforms, coefficient prioritization, and vector quantization. While such compressors may be adequate for visualization, they do not provide error controls on demand for scientific studies.

In recent years, a new generation of lossy compressors for scientific floating-point data has been proposed and developed, such as SZ [10, 30, 46, 49] and ZFP [9, 31]. Both lossy compressors can provide multiple compression modes, such as error-bounded mode

[1]The code of CEAZ is available at https://github.com/szcompressor/CEAZ.

and fixed-rate mode to introduce error control or compression ratio control. Error-bounded mode requires users to set an error bound, and fixed-rate mode means that users can set a target bitrate. Compared to ZFP which utilizes Discrete Fourier transform to manipulate data information, SZ predicts each data point's value by its neighboring data points in a multidimensional space with an adaptive predictor (mainly using a Lorenzo predictor [23]). Next, it performs an error-controlled linear-scaling quantization to convert all floating-point values to an array of integer numbers. Lastly, it performs a customized Huffman coding and lossless compression to shrink the data size significantly. This helps SZ provide a unified error distribution between original and reconstructed data within the user-set error-bound range, which fully utilizes the error tolerance space and provides a high compression ratio [10, 30, 34, 45, 46].

SZ was first developed for CPU architectures, and has released the CUDA implementation (called cuSZ) [49]. Compared to lossy compression on CPUs, GPU-based lossy compression can provide much higher (de)compression throughputs [24]. cuSZ [49] and cuZFP [9] are existing GPU-based implementations of SZ and ZFP, respectively, which are capable of achieving tens to hundreds of GB/s (de)compression throughputs. However, these approaches are not very efficient on FPGAs, which have much lower clock frequencies compared to GPUs. Moreover, the FPGA chip space limitations prevent fitting too many instances of compression pipelines on chip. But, FPGA implementations offer several advantages over GPU implementations: (1) FPGAs can inherently provide low latency as well as deterministic latency for real-time applications. (2) FPGAs provide a high degree of user customization, and their implementations are easier to be integrated into other systems.

2.2 FPGA-based Lossy Compression

Existing works have shown that significant performance speedups can be achieved by offloading lossy compression onto hardware. GhostSZ [57] is the first implementation of SZ-1.0 [11] on FPGAs. GhostSZ improves throughput by 10~85× over the SZ with a similar compression ratio and peak signal-to-noise ratio (PSNR). However, SZ-1.0 is a deprecated version which suffers from low prediction accuracy which results in low compression ratios [48].

waveSZ [48] is another hardware implementation of SZ lossy compression. It adopts a wavefront memory layout to fit into SZ algorithm to alleviate the data dependency during the prediction process. It improves the compression ratio and throughput over GhostSZ. However, waveSZ has several drawbacks: (1) It just alleviates the data dependency using wavefront memory but does not eliminate it. As a result, its throughput does not exceed 1 GB/s. (2) Its wavefront memory layout involves rearranging data before compression, and this overhead would be relatively high when processing a large amount of data. (3) It only focuses on accelerating the prediction stage without handling the high overhead of Huffman coding. This, however, is the main bottleneck after fully removing the data dependency in prediction.

BurstZ [44] is a variant of the one-dimensional ZFP algorithm and also implemented onto FPGAs. BurstZ can provide a high throughput (8 GB/s), but it suffers from a significantly lower compression ratio drop compared with the original ZFP algorithm. For example, the original ZFP algorithm achieves 21× compression ratio on the NWChem dataset [38] with an error bound of 0.001, whereas BurstZ only achieves 4.7× compression using the same error bound. Besides, the 8GB throughput is much smaller than the throughput of current PCIe3/4 and InfiniBand. Moreover, DE-ZFP [19] is an FPGA implementation of modified ZFP by replacing the embedded encoding with a dictionary-based encoding. It focuses on maximizing throughput while minimizing the compression ratio increase. In addition, ZHW [58] is another FPGA-based lossy compressor based on ZFP, but it is not error-bounded. So, we take BurstZ as the current state-of-the-art FPGA implementation of ZFP.

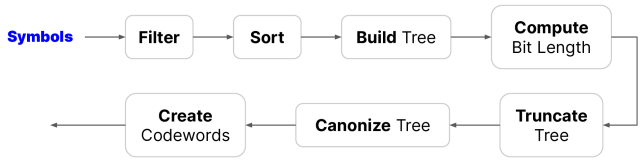


Figure 1: The process of canonical Huffman encoding.

2.3 MPI Collectives and MPI-I/O

Message Passing Interface (MPI) [17] contains two main types of operations related to parallel I/O, i.e., MPI-IO operations and collective operations. On the one hand, MPI-IO is a fundamental HPC middleware for parallel I/O. Many parallel I/O systems such as parallel HDF5 [14] and ADIOS [33] are built based on it. In MPI-IO, data is moved between files and processes by issuing read and write calls. The data access routines can be individual or collective. By using a collective routine, processes are coordinated with each other to optimize access to I/O devices. On the other hand, MPI collective operations such as scatter, gather and reduce play an important role in many HPC applications for communication. Considering that many applications use dedicated I/O nodes to periodically collect/distribute data from compute nodes and then write/read to the file system asynchronously, we also use MPI_Gather/Scatter in this paper to evaluate our compressor in this use scenario.

2.4 Research Challenges

High Overhead of Huffman Coding. Given a set of symbols, Huffman coding generates codewords based on the evidence that not all symbols have the same probability. Instead of using fixed-length codewords, Huffman coding uses variable-length codewords based on the relative frequency of different symbols. The principle is to use fewer bits to represent frequent symbols and more bits to represent infrequent symbols. Even though variable-length codewords can provide high compression ratio in our scenario, Huffman coding has high overhead in terms of latency, area, and power [29]. To achieve a high overall compression throughput, certain key challenges in Huffman coding need be addressed.

Challenge of Codewords Generation. The first challenge is to build a Huffman tree and generate codewords within limited hardware clock cycles to meet high-throughput requirements. Our goal is to accelerate MPI collective I/O in real time through compression. So we hope to reduce the compression latency as much as possible. In addition, generating codewords needs 7 steps: filter, sort, create tree, compute bit length, truncate tree, canonize tree, and create codewords, as shown in Figure 1. This procedure is a serial process

that is hard to be parallelized on FPGAs or GPUs. We make full use of the characteristics of the FPGAs to speed up this process by pipelining. However, the latency presented in Figure 4 is still large.

Challenge of Predefined Codewords. Inspired by [29], we will use predefined codewords at the beginning and update the codewords during the runtime. This method introduces the second challenge: how to generate suitable codewords to cover the features of all the scientific datasets?

3 DESIGN METHODOLOGY

In this section, we describe our proposed FPGA-based lossy compressor CEAZ and parallel I/O accelerator. Specifically, we will first overview the design of CEAZ and then describe our efficient and adaptive Huffman coder. Finally, we describe our proposed parallel I/O accelerator integrated with CEAZ.

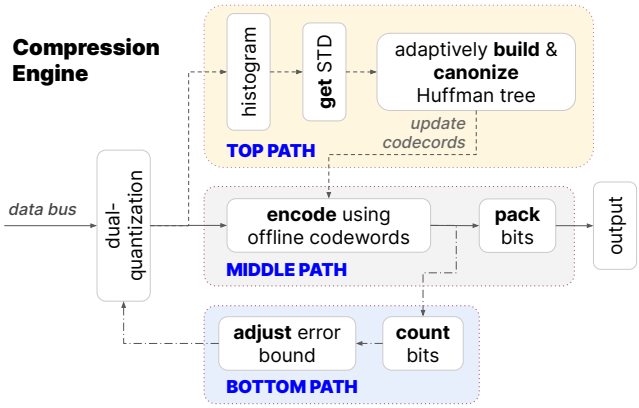


Figure 2: Design of our proposed lossy compression engine CEAZ.

3.1 Overview of CEAZ Engine

We show our proposed lossy compression engine in Figure 2. It has three main dataflow paths. On the top dataflow path, we preprocess float-point data using a dual quantization algorithm [49] (abbreviated as dual-quant), which generates integers as symbols (quantization codes) for the following Huffman coding. We collect frequencies of symbols using a histogram and calculate the standard deviation (STD) of frequencies. According to the STD value, we will decide whether to build a new Huffman tree based on the symbol frequencies or not (will be discussed in the next section).

Figure 3 shows our dual-quant pipeline design. Dual-quant is a novel two-phase prediction-quantization approach, which can completely eliminate the data dependency in the prediction and quantization steps. Our dual-quant consists of two steps: prequantization

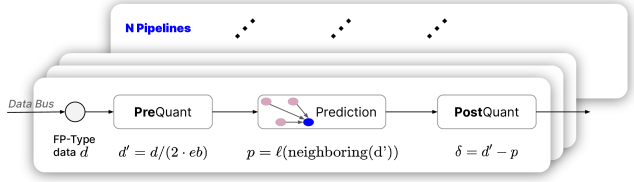


Figure 3: Design of our adopted dual-quantization pipeline.

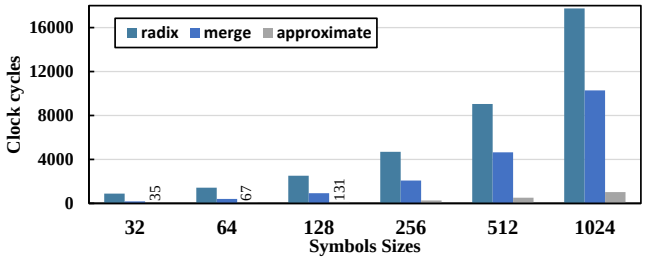


Figure 4: Latencies of different sorts with different symbol sizes.

and postquantization. Given a float point data d, we first quantize it based on the user-set error bound and convert it to an integer data d'. After the prequantization, we can calculate its predicted value based on its neighboring values (denoted neighboring(d')) using Lorenzo predictor (denoted ℓ), $p = \ell(\text{neighboring}(d'))$. The second step, called postquantization, computes the difference δ between the predicted value and the prequantized value. δ will be compressed by Huffman compression. Since the dual-quant part has no data dependency, we could instantiate N pipelines to process N float-point data in parallel. On the middle dataflow path, we directly encode symbols using existing codewords for seeking high throughput. The encoder can find the codeword corresponding to each symbol and output it. We then pack variable-length encoded (compressed) symbols to save the storage space. On the bottom dataflow path, we feed back total bits of encoded symbols to estimate compression ratio, and then adjust the error bound.

Our proposed compression engine has two working modes: fixed accuracy (i.e., error bounded) and fixed ratio (i.e., fixed bit-rate). The fixed-accuracy mode ensures information loss of compressed data is within the specific error bound, whereas the fixed-ratio mode ensures the transfer of compressed data has a consistent throughput. For the fixed-accuracy mode, we need to define an upper bound of errors that can be tolerated by applications and keep using this error bound through the whole compression process. For the fixed-ratio mode, we can set a suitable error bound to achieve a target compression ratio, as a higher error bound leads to a higher compression ratio. Specifically, we adjust the error bound as follows:

(1) We use $C = \frac{\texttt{TotalBits(original data)}}{\texttt{TotalBits(compressed data)}}$ to estimate the compression ratio, where $\texttt{TotalBits(original data)} = W * N$. Here W is the bit-rate of original data; for single/double floating-point data, W is 32/64 bits per value; N is the total number of data points that have been compressed.
(2) We calculate the compressed and target bit-rates by $B = \frac{W}{C}$ and $B_{\text{target}} = \frac{W}{C_{\text{target}}}$, respectively.
(3) We adjust the error bound by Equation (2) (will be discussed).

3.2 Efficient and Adaptive Huffman Coder

3.2.1 Fast approximate sort. In order to build the Huffman tree, we must sort the symbols based on their frequencies after we filter out symbols with a frequency of zero. Previous works [26, 51] use radix sort to reduce the utilization of hardware resources. However, we find that radix sort typically takes more than 30% of the total time when we break down the execution time of generating codewords. This is because the time complexity of radix sort is $O(d \times (n + b))$ (where b is the base to represent numbers and d is the maximum

Algorithm 1: Proposed fast sort based on Lorenzo predictor's feature.

Result: A approximately sorted array

S: defined structure contains two members: symbol, and its frequency
A: input array, its data type is S, len: length of input array, i: index of A
p: index of symbol 512 in A, m: index of the midpoint of A, l: index, h: index
O: output sorted array, j: index of O
t: loop count
$l = p - 1, h = p + 1, j = len - 2, O[len - 1] = A[p]$
if $p \leq m$ **then**
  $t = p$
else
  $t = len - p - 1$
end
for $i \leftarrow 1$ *to rows* **do**
  Count **if** $A[l].frequency \leq A[h].frequency$ **then**
    $O[j] = A[h]$
    $O[j-1] = A[l]$
  else
    $O[j] = A[l]$
    $O[j-1] = A[h]$
  end
  $l = l - 1, h = h + 1, j = j - 2$
end
CopyRemaining(A, O) /* copy remaining data from A to O */

number of digits) with $d = 32$ and $b = 10$ typically. We also adopt the non-recursive way of merge sort in our hardware implementation. However, we identify that Vitis HLS requires the size of the array to be sorted is a constant and a power of two [53], which prevents us from using the original merge-sort hardware implementation. This is because the non-zero frequencies to be sorted in our case are neither fixed nor the power of two.

We note that the frequencies of symbols that are generated by Lorenzo predictor and linear-scaling quantization [46] are symmetric, as shown in prior studies [25]. We verify this in our experiments with 1024 symbols, as shown in Figure 5 (symmetric with respect to symbol 512). This feature inspires us to use an approximate sort to improve the efficiency, since Huffman coding can accept the approximately sorted symbols, which would not notably degrade the compression ratio. Specifically, assuming A is an unsorted array with symbols, and O stores sorted symbols. We initialize two indexes l and h to represent the symbols to the left and right of the middle symbol. We compare the frequencies of the two symbols and store them in a correct order into O. We then decrease l by 1 and increase h by 1 and repeat this till all symbols are sorted.

We describe our proposed approximate sort algorithm in Algorithm 1 in detail. It has the time complexity of $O(\frac{n}{2})$, which is lower than the time complexity of radix sort and merge sort. We compare sort time of different sort algorithms with different symbol sizes. The results shown in Figure 4 illustrate that our approximate sort saves the sort time by 7.5× on average over the merge sort. More details about our evaluation platform will be shown in Section 4.

3.2.2 Offline Huffman codewords generation. On the premise of meeting the acceptable reduction in compression ratio, we propose to combine offline and online Huffman codewords generation strategies in order to improve the throughput as much as possible. As shown in figure 2, the symbols generated at the beginning by dual-quant will be encoded by offline codewords directly; at the same time, we also collect the frequencies of symbols. We will generate new Huffman codewords if the change of STD of symbol frequencies is greater than the threshold τ. τ is a hyper-parameter, and we will discuss it in the next section.

In order to make offline codewords representative, we generate corresponding offline codewords for various types of datasets that are currently available. The current types of offline codewords include Climate, Cosmology, Molecular, and Physics. When we target to compress a certain type of dataset, we will use the offline codewords corresponding to this type. However, when we encounter a new type of dataset, we will first use the average offline codewords. Then we will add the offline codewords of this new type into our offline codewords repository for future uses.

Specifically, we generate offline codewords based on the following four steps: (1) We set a suitable error bound to let our compressor have a similar compression ratio on different datasets under the same type. (2) We collect symbol frequencies on different datasets under the same type. (3) We calculate the average symbol frequencies from collected frequencies. The average symbol frequencies are used to generate offline codewords. (4) We store the offline codewords of this type into our offline codewords repository. In order to make offline codewords representative and promising for high compression ratio, we collect symbol frequencies based on all the real-world datasets from the Scientific Data Reduction Benchmarks (SDRBench) [41] and Datasets for Benchmarking Floating-Point Compressors [27]. Figure 9 shows the ratio degradation by using the offline codewords (will be showed in Section 4.4).

Moreover, to generate the average offline codewords, we use the following three steps: (1) We set a suitable error bound to let our compressor have a similar compression ratio on different datasets under different types. (2) We collect symbol frequencies on different datasets under different types. (3) We calculate the average symbol frequencies from collected frequencies. The average symbol frequencies are used to generate the average offline codewords.

Using different error bounds to compress the same dataset results in different histograms of symbols (i.e., different distributions of symbol frequencies). For example, using a larger error bound results in a tighter histogram of symbols compared to using a smaller error bound. In extreme cases where very large error bounds are used, there can be only a few symbols for Huffman coding. To make the offline codewords adaptive for a wide range of datasets, we must choose suitable error bounds for multiple scientific datasets, which can result in a similar histogram of symbols after employing the Lorenzo predictor. In other words, we must control the error bound for each dataset to provide a similar ratio. Instead of using a trial-and-error approach to search the suitable error bound for every dataset, we provide a theoretical analysis to predict the error bound given a target ratio based on one-time sampling.

A naive solution to align different datasets with a similar compression ratio is to use the same value-ranged-based relative error bound instead of the same absolute error bound. While using the same value-ranged-based relative error bound for different datasets can reduce the divergence of their symbols' histograms, it cannot guarantee that the compression ratio of different datasets is similar to each other. In our experiment, we identify the compression ratio range of 4~13× when using the same value-ranged-based relative error bound for multiple scientific datasets. Our proposed solution considers the efficiency of Huffman coding affected by error bound to accurately estimate the error bound for a target compression

ratio. We assume the bit-rate of symbol after Huffman encoding is:

$$\text{mean}(L) = \sum_{i=0}^{n} P(s_i)L(s_i) \approx \sum_{i=0}^{n} P(s_i)\log_2 P(s_i), \tag{1}$$

where n is the number of different Huffman code, P is the probability of given code s_i, L is the length of given code s_i. We further represent the Huffman code length based on its probability with binary base-2 numeral system. Note that in our case, 1024 symbols are used for Huffman coding and thus are sufficient for this simplification. Consider a given error bound eb can provide a bit-rate of B, when doubling the error bound to $2eb$, the symbols' histogram also shrinks accordingly where the total number of symbols is reduced by 2× and the possibility of each symbol is increased by 2×. In this case, the bit-rate should be:

$$B' = \sum_{i=0}^{n/2} P'(s_i)\log_2 P'(s_i) \approx \sum_{i=0}^{n/2} (P(s_{2i-1})\log_2 P'(s_{2i-1}) + P(s_{2i})\log_2 P'(s_{2i})) - 1 = B - 1 \tag{2}$$

Thus, we conclude that by doubling the error bound, the bit-rate should increase by 1. Furthermore, we can derive that if the compression bit-rate is B under the error bound eb, then under the new error bound Neb the predicted bit-rate is $B' = B - \log_2 N$. Note that the SZ algorithm uses previous data points' quantized values to predict the value of current point based on Lorenzo prediction, which means different error bounds would affect the shape of symbols' histogram. However, this only applies to very large error bounds and hence few quantization bins. In our case, we simplify this to a fixed symbols' histogram shape under different error bounds, yielding a precise 2× shrink when doubling the error bound.

With the above analysis, we can simply compress each scientific dataset once with the same value-ranged-based relative error bound eb and compute the optimized error bound eb' for the target bit-rate B_{target} based on the current bit-rate B by $eb' = 2^{B-B_{\text{target}}}eb$.

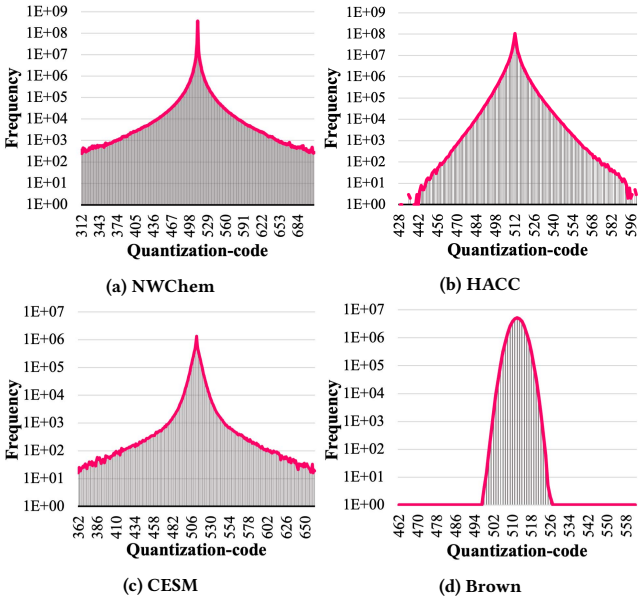


Figure 5: Distribution of symbol frequencies on four scientific datasets, i.e., NWChem [38], HACC [20], CESM [8], and Brown [5].

3.2.3 Adaptive online codewords updates. In general, on one hand, the more frequently we update the codewords, the closer we can get to the optimal codewords in terms of compression ratio. On the other hand, too frequently updating codewords may decrease the throughput, since the newly generated codewords need to be stored. Moreover, if we do not update the codewords, the compression ratio may decrease as well, because the old codewords are too outdated to reflect current distribution of symbol frequencies. In order to solve this problem, we use two effective metrics to determine when to generate new codewords: (1) the storage overhead of codewords and (2) the change of distribution of symbol frequencies.

For example, suppose we have S symbols and S codewords. Each codeword is B bits on average after canonization process. We have `size(codewords)` $= S \times B$. Our target compression ratio is C. The bit-rate of original data is W. For single or double floating-point data set, the bit width is 32 or 64 bits per value. The bit-rate R is $\frac{W}{C}$. Bit-rate R can also be regarded as an average bit length of compressed data. We have `size(compressed data)` $= R * N$, where N is the total number of original data. Assume the ratio of the codewords size to the compressed data size is O, if the codewords overhead is set to be less than 10%, $\frac{S\times B}{S\times B+R\times N} \leq 10\%$.

The symbols generated by dual-quant present a centralized and symmetric distribution, as shown in Figure 5. The generated codewords are highly related to the distribution of symbol frequencies [3]. These good characteristics inspire us to evaluate the similarity of two sets of symbol frequencies using STD. Specifically, assume σ_0 is the STD obtained from the previous data chunk, σ_1 is the STD obtained from the current data chunk, and $\chi = |\sigma_0 - \sigma_1|$. We define a set of thresholds τ_x and propose the following strategy:

- We will not generate new codewords if $\chi \leq \tau_0$ (two frequencies with similar distributions generate almost identical codewords) but keep using the old codewords;
- We will generate new codewords if $\tau_0 < \chi \leq \tau_1$;
- We will use the offline Huffman codewords if $\chi \geq \tau_1$.

The change of STD in a large dataset typically does not exceed τ_0, so we also update the codewords for every U chunks to better utilize the top dataflow path and increase the compression ratio. We are processing data that is completely different from the previous data if distribution changes drastically. We need to clear the histogram of compression engine and collect new symbol frequencies. We set τ_0 and τ_1 as 3.05 and 4.88, respectively, after comprehensive experiments (will be discussed in detail in Section 4.5).

Note that our design is different from other Huffman coding works in terms of adaptivity. For example, Tian *et al.* [50] proposed a reduction-based scheme for GPUs that iteratively merges the encoded symbols and adaptively determines the number of merge iterations. However, CEAZ only builds a new codebook for the data chunk when the change of its histogram exceeds a threshold in order to target FPGA with limited resources and low clock frequency.

3.3 Parallel I/O Accelerator

Figure 6 shows the overview of our system architecture integrated with CEAZ compression engine. Our system includes two parts: (1) The host partitions the input dataset and feed the chucked data through the PCIe. Raw data is buffered in high-bandwidth memory (HBM) with 460 GB/s bandwidth and converted into stream data by

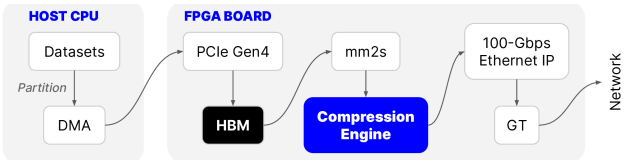


Figure 6: Overview of system architecture integrated with CEAZ.

memory to stream (mm2s) unit. (2) The compression engine compresses the stream data in real-time and output compressed data. Ethernet intellectual property (IP) packs the compressed data according to the network protocol. QSFP28 (fiber optical transceiver) gigabit transceiver (GT) finally outputs packed data into network.

Many scientific applications, such as cosmology simulations, need to periodically dump a huge amount of raw simulation data to the storage for post-hoc analysis and visualization after simulations. Data across all computing nodes needs to be aggregated to the storage node(s). Even though state-of-the-art supercomputers are using InifniBand interconnect (e.g., 200 Gb/s), it can take hours to complete the data aggregation and save (e.g., 1.5 TB/s of aggregated I/O bandwidth and 4.85 PB memory capacity in Fugaku supercomputer [36]). Therefore, we propose to apply CEAZ to the future HPC systems, as shown in Figure 7.

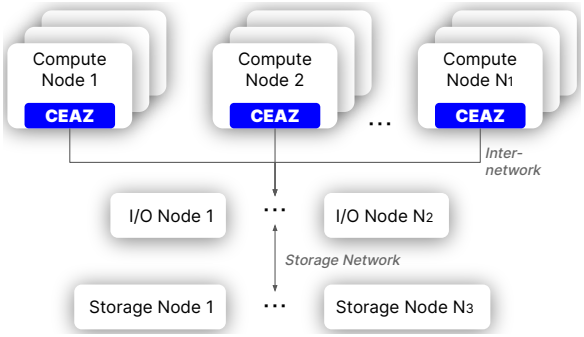


Figure 7: Overview of CEAZ-supported parallel I/O system.

Specifically, CEAZ is directly integrated into the FPGA-based SmartNIC in each computing node and used to compress the raw data before transmitting it to the storage system via interconnection network. There are two main scenarios to use the CEAZ-compressed data in the storage: (1) checkpoint/restart and (2) post analysis and visualization. Thus, there is no need to change the storage system to adapt to our design, since the data is compressed (decompressed) before (after) sending (receiving) to (from) the network adapter. Note that similar to the FPGA-based SmartNIC, the emerging Data Processing Unit (DPUs) [12]—a class of programmable processor—based SmartNIC can also offload and improve application performance for communications and storage. We will extend CEAZ to DPU-based systems in future work.

4 EXPERIMENTAL EVALUATION

In this section, we evaluate our proposed FPGA-based error-bounded lossy compressor CEAZ and demonstrate its effectiveness in two perspectives: (1) the effectiveness of the proposed adaptive compression algorithm and the performance (i.e., throughput and latency) of its accelerator implementation, and (2) the improvement of parallel I/O supported by CEAZ in different scales.

Table 1: Test datasets from Scientific Data Reduction Benchmarks.

type	name	# fields	precision	dimensions	size
Climate	CESM	77	`float`	1,800×3,600	1.86 GB
Cosmology	HACC	6	`float`	280,953,867	6.28 GB
	NYX	6	`float`	512×512×512	3.0 GB
Molecular	NWChem	3	`double`	1,617,048,176	12.05 GB
Physics	S3D	55	`double`	500×500×500	51.22 GB
Other	Brown	3	`double`	33,554,433	0.75 GB

4.1 Experimental Setup

Experimental Platform. We use two platforms as our testbed. The first platform is Xilinx Alveo U280 Data Center accelerator card, which is equipped with a PCIe Gen4x8 with CCIX to leverage the latest server interconnect infrastructure for high-bandwidth host processors, 8 GB HBM2 and 32 GB on-board DDR4 DRAM. CEAZ is implemented with Xilinx Vitis unified software platform (v.2020.2) [52]. The second platform (for parallel I/Os) is Summit [43], which is one of the most powerful supercomputers in the world. We perform GPU experiments on an NVIDIA Tesla V100 GPU.

Test Datasets. To conduct our evaluation and comparison under realistic scenarios, we use six real-world datasets from the Scientific Data Reduction Benchmarks [41]. Datasets belong to various domains: (1) 2D CESM-ATM climate simulation [8]. (2) 1D HACC cosmology particle simulation [20]. (3) 3D NYX adaptive mesh hydrodynamics and N-body cosmological simulation [20]. (4) 1D NWChem two-electron repulsion integrals computed over Gaussian-type orbital basis sets [38]. The sizes of the three fields are 102,953,248, 801,098,891, and 712,996,037, respectively. (5) 3D S3D Combustion simulation [28]. Each raw file needs to be split into 11 files to form the data size of $500 \times 500 \times 500$. (6) 1D Brown Samples synthetic and generated to specified regularity [5]. More details about the datasets can be found in Table 1.

4.2 Resource Utilization and Clock Frequency

Table 2 shows the breakdown of hardware resource utilization of CEAZ. To provide a fair comparison with BurstZ, we implement 32 pipelines for single-precision datasets and 16 pipelines for double-precision datasets. U280 has 2 HBM2 stacks and each stack has 16 channels. Each channel is 32-bits data width. 32 pipelines require a data width of exactly 1024. The BRAM_18K is dual-port RAM module instantiated into the FPGA for on-chip storage, and its size is 18k bits. DSP block is an arithmetic logic unit. FF represents for flip-flop. Look-up table (LUT) is the basic building block of an FPGA and is capable of implementing any logic function. The running frequency based on our measurement is slightly lower than the clock frequency we set (i.e., 300 MHz). Note that N/A means the utilization data is not provided in the BurstZ work [44].

4.3 Evaluation on Robustness

There are mainly two cases where the codewords change drastically. The first case is at the very beginning of the compression process. The offline codewords at the beginning may be largely

Table 2: Hardware resource utilization.

	board	BRAM_18K	DSP	FF	LUT	percent
BurstZ	VCU118	222	N/A	N/A	125000	<5%
CEAZ	U280	475	256	67623	302407	<28%

different from the actual codewords. The second case is during the compression process, the statistics of input data chunk suddenly change. Therefore, we need to evaluate the response time of our system to these cases, which is the duration that our system needs to recover to a reasonable compression ratio.

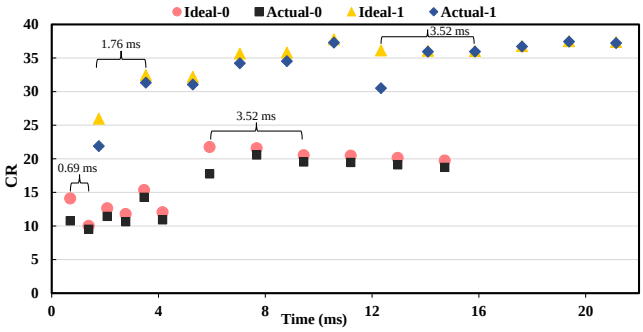


Figure 8: Response time when codewords change drastically.

To prove the robustness of CEAZ, we concatenate two different datasets with different types. Specifically, we concatenate CESM and NYX for the first dataset and concatenate NWChem and S3D for the second dataset. The reason that we concatenate different datasets is that codewords can change drastically both at the beginning and during the processing. Figure 8 shows the response time when current codewords are significantly different from the actual codewords. In the figure, "Actual-0" and "Actual-1" represent for the actual compression ratio of CEAZ on the two datasets based on offline codewords or previous chunk's codewords, while "Ideal-0" and "Ideal-1" represent for the ideal compression ratio that CEAZ can achieve on the two datasets using the codewords generated from current chunk. We note that CEAZ can recover to a reasonable compression ratio with only 3.52 *ms* with a strong robustness.

4.4 Evaluation on Offline Codewords and Codewords Update Frequency

We use predefined (offline) codewords at the beginning of the compression process. To prove the effectiveness of our predefined codewords, we evaluate four datasets (i.e., NWChem, HACC, CESM, and S3D) using these codewords with the same value-ranged-based relative error bound of 1e-4 and compare their compression ratios with the optimal ones, as shown in Figure 9. The orange bars represent the ideal compression ratio achieved by first building Huffman tree and then generating accurate codewords. The blue bars represent the compression ratio by directly using our offline codewords. The compression ratio drops on NWChem, CESM, and S3D are 5.1%~10.7%. The compression ratio degradation on HACC is more obvious (i.e., ~18.2%); this is because the Lorenzo predictor has low efficiency on HACC dataset, causing many outliers (unpredictable data points), and the distribution of quantization codes generated by the Lorenzo predictor is not statistically representative.

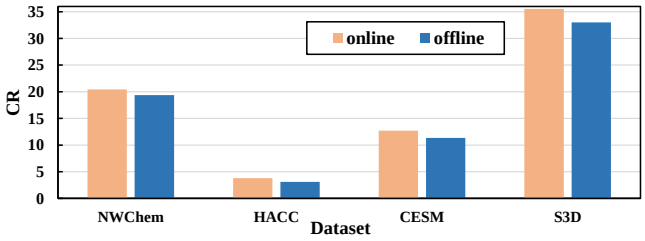


Figure 9: Comparison of compression ratio (CR) between offline codewords and online codewords.

We note that even though HACC's compression ratio drops more than 18.2%, the compression ratio over 3× can still relieve the communication pressure to a certain extent thanks to our high throughput (will be discussed in Section 4.8).

As aforementioned, frequently updating codewords will decrease the compression ratio due to the overhead of saving codewords. We evaluate the impact of update frequency on the final compression ratio. We perform the experiments on both CESM and NYX. We set the error bound to the value-range-based relative error bound of 1e-4. We choose to update the codewords every 1 MB, 2 MB, 4 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, and 512 MB. We find that the compression ratio is significantly reduced when the update size is smaller than 32, because the overhead of storing the codewords is relatively large. Moreover, we also observe that the compression ratio decreases when the update size is larger than 256 MB. This is because the codewords are outdated to reflect the current symbols frequencies. Thus, we choose 32 MB as our default update size.

4.5 Evaluation on Change of Standard Deviation of Symbol Frequencies

As aforementioned, we use the change of standard deviation of symbol frequencies (i.e., $\chi = \mid \sigma_0 - \sigma_1 \mid$) to determine when to generate new codewords, use old codewords, or use offline codewords. Using previous codewords under a large χ (a large difference between current and previous distribution) results in a notable drop of compression ratio, while generating new codewords under a small χ (a small difference between current and previous distribution) leads to a high overhead of Huffman coding. Thus, we use experiments to find the suitable thresholds τ_0 and τ_1. As shown in Figure 10, the drop of ratio is less than 10% when $\chi \leq 3.05$, while the drop is over 25% when $\chi \geq 4.88$. So, we set τ_0 and τ_1 to 3.05 and 4.88, respectively, to meet both requirements on ratio and performance.

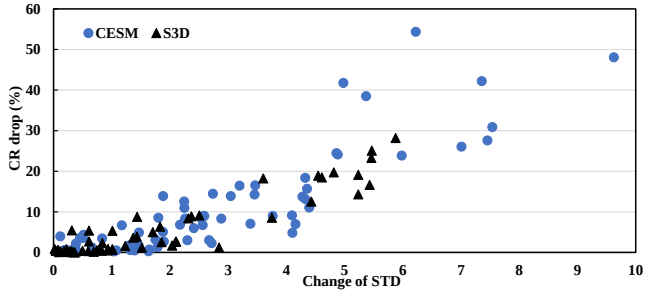


Figure 10: Compression ratio drops with different changes of STD.

4.6 Evaluation on Fixed-Ratio Mode

As discussed in Section 3, our novel compression engine has two working modes: fixed-accuracy mode (i.e. error-bounded mode) and fixed-ratio mode (i.e., fixed bit-rate mode). The fixed-ratio mode can allow the system have a consistent throughput for data transfer. To verify the effectiveness of our fixed-ratio mode, we set the target compression ratios of 10.5 and 21 for single and double floating-point data, respectively. Figure 11 shows the compression between the target ratio and the actual ratio. The difference is within 7.5%, which is acceptable in our use case.

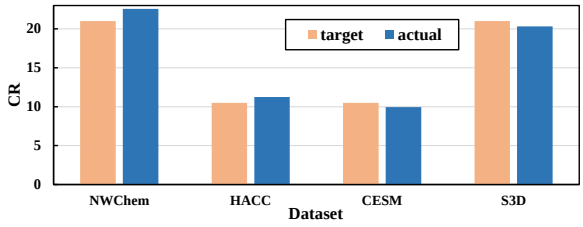


Figure 11: Comparison of target & actual ratio in fixed-ratio mode.

4.7 Evaluation on Ratio and Distortion

Compression ratio is defined as the ratio of original data size to the compressed data size. PSNR is a widely used indicator to assess the distortion of data after lossy compression, which is calculated as $\text{PSNR} = 20 \cdot \log_{10}\left[(d_{\max} - d_{\min})/\text{RMSE}\right]$. N is the number of data points and $d_{\max}$ and $d_{\min}$ are the maximal and minimal values, respectively. RMSE is the *root mean squared error*, i.e., $\text{sqrt}\left(\frac{1}{N}\sum_{i=1}^{N}\left(d_i - d_i^{\bullet}\right)^2\right)$, where d_i and $d_i^{\bullet}$ are the original and decompressed data values, respectively. The larger the PSNR, the lower the data distortion, hence more accurate post analysis.

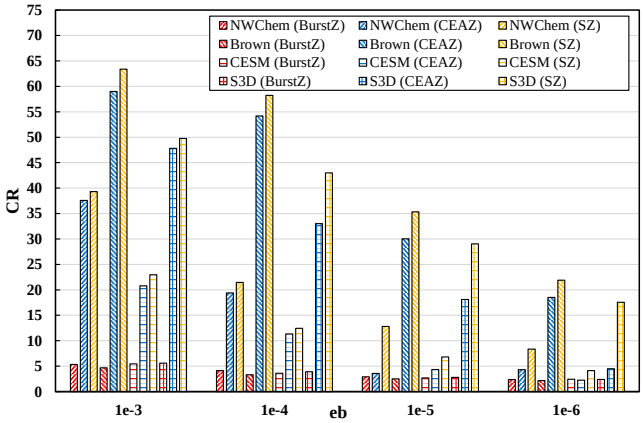


Figure 12: Ratio comparison among BurstZ, CEAZ, and SZ.

Figure 12 shows the comparison of compression ratio among BurstZ, CEAZ, and CPU-SZ on our test datasets with different value-ranged-based relative error bounds of 1e-3~1e-6. The compression ratio of our CEAZ is notably higher than that of BurstZ. CEAZ consistently provides 1.2×~16.4× higher compression ratio than BurstZ under the same error bound. Particularly, our CEAZ improves the compression ratio by up to 12× on the Brown dataset over BurstZ when the error bound is equal to 1e-3. Compared to the CPU-SZ, the degradation of compression ratio is within 23.3% on all test datasets and reasonable error bounds. This result demonstrates the effectiveness of our adaptive strategy and offline codewords.

We also compare the compression ratio and throughput among LZ4, Gzip, and CEAZ with the best compression mode for LZ4/Gzip. Table 3 illustrates that CEAZ is effective to reduce the scientific data size with high throughput.

In addition, Table 4 shows the comparison of distortion (PSNR) between CEAZ and CPU-SZ under different error bounds. The degradation of PSNR is within 4 dB under very high PSNRs (all higher than 60 dB).

Table 3: Comparison of compression ratio and averaged throughput (in GB/s) among LZ4, Gzip, CEAZ, and CPU-SZ on test datasets.

	eb	NWChem	Brown	CESM	S3D	throughput
LZ4	N/A	1.005	1.003	1.182	1.055	1.43
Gzip	N/A	1.056	1.442	1.361	1.181	1.14
CEAZ	1e-4	20.4	58.2	12.3	35.0	17.8
CPU-SZ	1e-4	21.5	58.2	12.5	43.0	0.31

Table 4: Distortion (i.e., PSNR) comparison between CEAZ and SZ.

eb	**NWChem**		**Brown**		**CESM**		**S3D**	
	SZ	CEAZ	SZ	CEAZ	SZ	CEAZ	SZ	CEAZ
1e-3	65.8	75.1	64.7	64.8	65.6	65.4	71.2	68
1e-4	85.8	90.4	84.8	84.8	85.4	84.8	88.8	84.9
1e-5	105.6	107.7	104.8	104.8	105.4	105.3	108.2	104.8
1e-6	125.0	126.0	124.8	124.8	125.5	125.3	127.7	124.8

4.8 Evaluation on Time, Throughput, Latency

Time. The compression time (excluding the file loading and dumping time) is measured as the period from the moment that FPGA receives the data through the moment that the whole compression is finished with output bytes. We show the comparison of compression time among BurstZ, CEAZ, and CPU-SZ in Table 5. The error bounds are 1e-4 and 1e-5. We observe that CEAZ reduces the compression time on average 55.8% compared with the second-best BurstZ on the same dataset.

Table 5: Compression time (in second) of different compressors.

	eb	NWChem	Brown	CESM	S3D
BurstZ	1e-4	1.65	0.09	0.23	6.40
BurstZ	1e-5	1.70	0.13	0.35	9.67
CEAZ	1e-4	0.67	0.04	0.11	2.93
CEAZ	1e-5	0.67	0.04	0.11	2.93
CPU-SZ	1e-4	27.6	1.58	12.28	141.4
CPU-SZ	1e-5	28.6	1.59	12.37	160.8

Throughput. The compression throughput is defined as the size of data being compressed per second. In order to compare the compression throughputs on CPU and GPU, we evaluate the throughputs of CPU-SZ, cuSZ, cuZFP, and CEAZ across three datasets, as

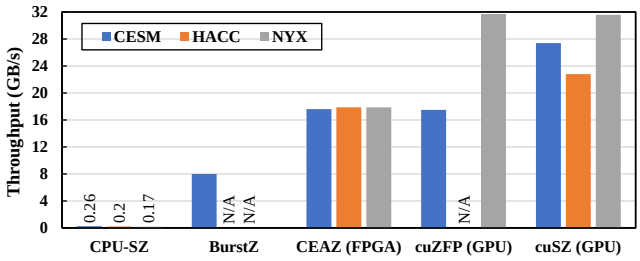


Figure 13: Throughput comparison among BurstZ, CEAZ, CPU-SZ.

shown in Figure 13. We set the error bound to 1e-4, which has the data distortion (i.e., PSNR) of about 85 dB.

Note that the throughput of cuZFP is highly related to its user-set fixed bitrate according to the previous study [24], whereas the throughputs of BurstZ, cuSZ, and CEAZ are almost unaffected by the user-set error bound. Therefore, we choose the acceptable fixed bitrate for cuZFP, which generates the data distortion similar to that of CEAZ. The figure illustrates that CEAZ can consistently provide about 17.8 GB/s throughput with different error bounds, which is 2.3× higher than BurstZ on average. Compared with the serial CPU-SZ, CEAZ improves the throughput by 67.4× on average.

It is worth noting that CEAZ can provide at least 3.0× higher compression ratio compared to BurstZ when error bounds are from 1e-3 to 1e-4. Due to such a high compression ratio (or high data reduction capability), the bandwidth of dumping compressed data (even with the smallest ratio) is still less than the bandwidth capacity of the Ethernet transceiver in our FPGA board (i.e., 100 Gb/s), thereby, the overall throughput has not been bounded by this capacity. Moreover, our clock frequency is around 300 MHz; thus, the throughput could be further improved by increasing the frequency.

Table 6: Hardware specifications of tested GPU and FPGA.

	Power	Tech.	BW	Freq.	Peak Perf.
V100	250W	12 nm	900 GB/s	1.25 GHz	15.7 TF/s
U280	225W	16 nm	460 GB/s	0.3 GHz	-

CEAZ stably provides a compression throughput of 17.8 GB/s, which is about 56%~78% of cuSZ/cuZFP's throughputs (note that cuZFP has a fairly low compression quality on 1D HACC). CEAZ and cuSZ/cuZFP are implemented on Xilinx Alveo U280 FPGA card and Nvidia Tesla V100 GPU, respectively. Table 6 lists the hardware specifications of V100 GPU and U280 FPGA. Note that V100 provides up to 900 GB/s bandwidth (1.96× higher than U280) and 1.25 GHz frequency (4.2× higher than U280). In addition, although we cannot find U280's theoretical peak performance, a similar FPGA, Intel S10 NX, has the peak performance of 3.96 TF/s (4.0× lower than V100) [4, 37]. Therefore, FPGA-based CEAZ is more efficient in resource utilization than GPU-based cuSZ/cuZFP.

We also note that there is a recent work (called DE-ZFP) [19] that develops an FPGA implementation of a modified ZFP algorithm. However, the paper only evaluates DE-ZFP on three datasets with two absolute error bounds. Thus, we select their tested data fields that have reasonable relative error bounds (under the absolute error bounds of 1e-3 and 1e-6) and perform a comparison. Our evaluation shows that CEAZ has 1.7× higher compression ratio and 11.1× higher compression throughput over DE-ZFP on average.

Table 7: Latency (μs) of different lossy compressors on small data.

	CPU-SZ	cuSZ (GPU)	cuZFP (GPU)	CEZA (FPGA)
1 KB	69	358.3	16.6	3.7
4 KB	114	416.6	19.4	3.8
16 KB	147	507.4	27.4	4.5
64 KB	458	546.6	46.7	7.0

Latency. We evaluate the latency of CPU-SZ, cuSZ, cuZFP, and CEAZ on small datasets to demonstrate the capability of using CEAZ in reducing the communication cost in future work, as shown in Table 7. The test small data are chunked from the CESM-ATM dataset. The table illustrates that CEAZ achieves up to 113.4× and 6.7× lower latency than cuSZ and cuZFP, respectively.

4.9 Parallel Performance Evaluation

We demonstrate the parallel performance in two ways. We first evaluate the throughput of CEAZ with multiple pipelines in the single FPGA board. We then evaluate CEAZ with multiple nodes.

4.9.1 Multi-pipeline Evaluation. We choose the CESM-ATM and NYX datasets and set the value-range-based error bound to 1e-4, which is commonly used in the CESM and NYX applications [39]. We increase the compression pipelines from 1 to 64. Figure 14 illustrates that the throughput (in log-scale) increases linearly as the number of pipelines increases. CEAZ can achieve this high scalability because (1) our compression engine reads data from HMB2 with a very high bandwidth of 460 GB/s, and (2) our compression engine adopts dual-quant to fully remove the data dependency so that we can process different chunks of the dataset in parallel.

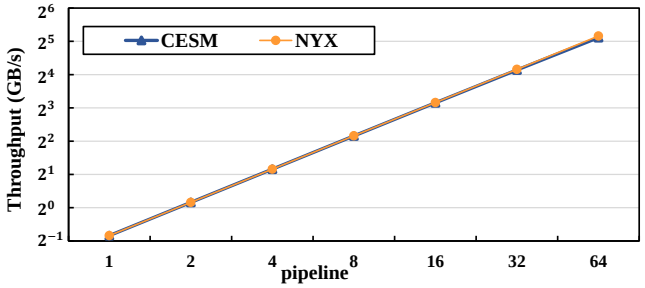
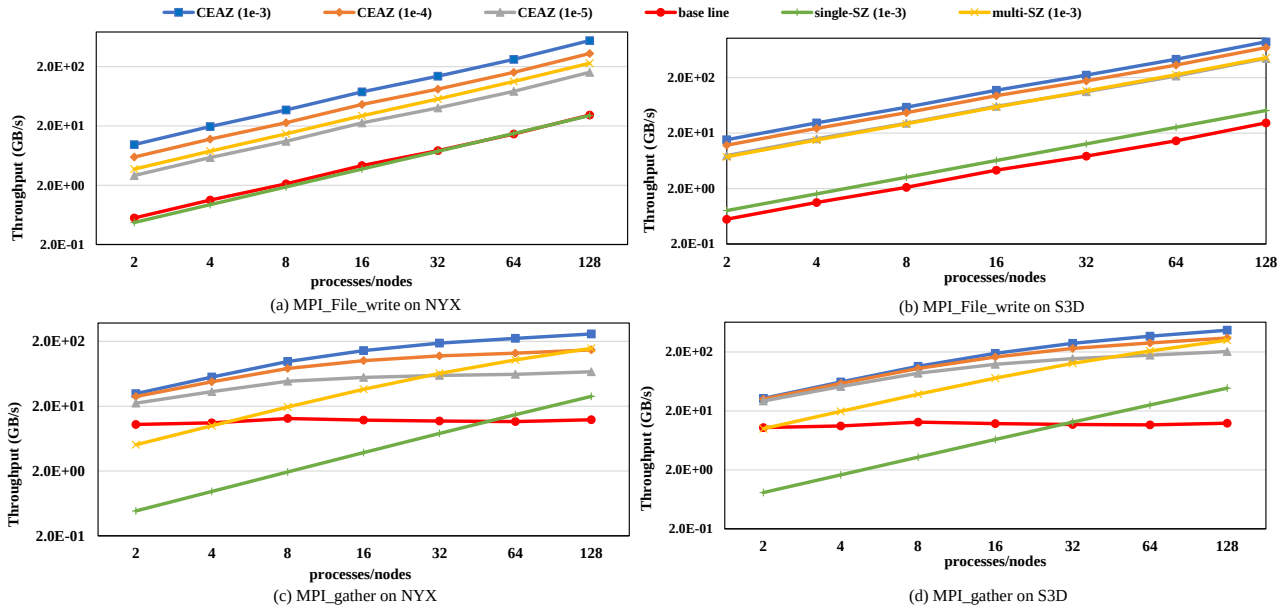


Figure 14: Compression throughputs with multiple pipelines.

4.9.2 Multi-node Evaluation. We evaluate the performance improvements of MPI-IO and MPI collective operations gained from CEAZ, i.e., MPI_File_write and MPI_Gather. We conduct our experiments with up to 128 nodes (one process per node). Each node holds a copy of datasets for compression and transmission, i.e., 3.0 GB of NYX and 10.2 GB of S3D per node. Thus, the overall data size for parallel I/O is up to 1.3 TB with 128 nodes. We evaluate CPU-SZ on both a single core and 32 cores with the error bound of 1e-3. Table 8 shows the compression ratio of CPU-SZ and CEAZ on NYX and S3D with different error bounds.

Figure 15 (a) and Figure 15 (b) show the MPI-IO throughputs on the NYX and S3D datasets with different approaches. The throughput of original MPI_File_write (without compression) increases as the number of nodes increases and can reach up to 30.5 GB/s with 128 nodes in Summit, as shown as the baselines in the figures. The single-core-SZ-supported MPI_File_write only achieves an overall



(a) MPI_File_write on NYX

(b) MPI_File_write on S3D

(c) MPI_gather on NYX

(d) MPI_gather on S3D

Figure 15: Throughput comparison of CEAZ-accelerated and original MPI_File_write and MPI_Gather on NYX and S3D with different numbers of processes/nodes. "single-SZ" denotes SZ with a single CPU core/node. "multi-SZ" denotes SZ with 32 CPU cores/node.

Table 8: Compression ratio of CPU-SZ and CEAZ on NYX and S3D.

	NYX		S3D	
	CPU-SZ	CEAZ	CPU-SZ	CEAZ
1e-3	26	23.7	49.8	47.8
1e-4	14.7	12.8	43.0	33
1e-5	7.7	5.7	29.1	18.1

throughput of 51.0 GB/s (including compression time and time to write compressed data) on S3D when using 128 nodes, which is 67.3% higher than the baseline. This is because the compression throughput of single-core CPU-SZ (i.e., about 0.41 GB/s) is not fast enough compared to the state-of-the-art interconnect such as InfiniBand HDR with a bandwidth of 200Gb/s. Note that unlike on S3D, the single-core-SZ-supported MPI_File_write on NYX is slower than the baseline, since the compression ratio of SZ on NYX (i.e., 26.0) is much lower than S3D (i.e., 49.8). The multi-core-SZ-supported MPI_File_write can provide an overall throughput of up to 456.56 GB/s when using 128 nodes, which is 15.0× higher than the baseline. In comparison, CEAZ-supported MPI_File_write can improve the overall throughput (including compression time and time to write compressed data) by 18.0× and 28.9× on NYX and S3D, respectively.

Figure 15 (c) and Figure 15 (d) show the MPI_Gather throughputs on the NYX and S3D datasets with different approaches. The throughput of original MPI_Gather reaches 12.4 GB/s with 128 nodes in Summit, as shown as the baselines in the figures. Similar to MPI_File_write, the single-core-SZ-supported MPI_Gather only achieves an overall throughput of up to 48.7 GB/s with 128 nodes, which is just 3.9× higher than the baseline; the multi-core-SZ-supported MPI_Gather can provide an overall throughput of up to 316.9 GB/s when using 128 nodes, which is 25.6× higher than the baseline. In comparison, CEAZ-supported MPI_Gather can improve the overall throughput by 21.0× and 37.8× on NYX and S3D, respectively, due to the high efficiency of CEAZ.

5 CONCLUSION AND FUTURE WORK

In this work, we propose CEAZ: a hardware-algorithm co-design of efficient and adaptive lossy compressor for scientific data. To achieve both high compression ratio and throughput, we propose an efficient Huffman coding approach that can adaptively update Huffman codewords online based on our offline generated representative Huffman codewords. We also derive a theoretical analysis to accurately control compression ratio under the error-bounded compression mode, enabling an accurate generation of offline Huffman codewords and a fixed-ratio compression mode. Our evaluation demonstrates that CEAZ outperforms the second-best FPGA-based error-bounded lossy compressor by 2.3× of throughput and 3.0× of compression ratio. CEAZ improves MPI_File_write and MPI_Gather by up to 28.9× and 37.8×, respectively, with 128 nodes in Summit.

In future work, we plan to deploy our system to FPGA-based clusters and extend CEAZ to DPU-based systems.

ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation OAC-2042084 and OAC-2104024. This work was also partially supported by the Compute-Flow-Architecture (CFA) project under PNNL's Data-Model-Convergence (DMC) LDRD Initiative. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] M Ainsworth, O Tugluk, B Whitney, and S Klasky. 2017. MGARD: A Multilevel Technique for Compression of Floating-Point Data. In *DRBSD-2 Workshop at Supercomputing*.

[2] Ann S Almgren, John B Bell, Mike J Lijewski, Zarija Lukić, and Ethan Van Andel. 2013. Nyx: A massively parallel amr code for computational cosmology. *The Astrophysical Journal* 765, 1 (2013), 39.

[3] Ian Blanes, Miguel Hernández-Cabronero, Joan Serra-Sagristà, and Michael W Marcellin. 2019. Lower bounds on the redundancy of Huffman codes with known and unknown probabilities. *IEEE Access* 7 (2019), 115857–115870.

[4] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C Hoe, Vaughn Betz, and Martin Langhammer. 2020. Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 10–19.

[5] Brown-Samples by Brown University. 2019. https://sdrbench.github.io/. Online.

[6] Martin Burtscher and Paruj Ratanaworabhan. 2008. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.* 58, 1 (2008), 18–31.

[7] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1201–1220.

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

[9] cuZFP. 2021. https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp.

[10] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, 730–739.

[11] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, Chicago, IL, USA, 730–739.

[12] DPU. 2021. https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu-data-processing-unit/. Online.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.

[14] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. 36–47.

[15] Tong Geng, Tianqi Wang, Chunshu Wu, Chen Yang, Wei Wu, Ang Li, and Martin C. Herbordt. 2019. O3BNN: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, Denver, CO, USA, 461–472.

[16] Ali Murat Gok, Sheng Di, Yuri Alexeev, Dingwen Tao, Vladimir Mironov, Xin Liang, and Franck Cappello. 2018. Pastri: Error-bounded lossy compression for two-electron integrals in quantum chemistry. In *2018 IEEE international conference on cluster computing (CLUSTER)*. IEEE, IEEE, 1–11.

[17] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.

[18] Pascal Grosset, Christopher Biwer, Jesus Pulido, Arvind Mohan, Ayan Biswas, John Patchett, Terece Turton, David Rogers, Daniel Livescu, and James Ahrens. 2020. Foresight: analysis that matters for data reduction. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, IEEE, 1171–1185.

[19] Mahmoud Habboush, Aiman H El-Maleh, Muhammad ES Elrabaa, and Saleh AlSaleh. 2022. DE-ZFP: An FPGA implementation of a modified ZFP compression/decompression algorithm. *Microprocessors and Microsystems* (2022), 104453.

[20] HACC team (ECP EXASKY). 2019. https://sdrbench.github.io/. Online.

[21] Pouya Haghi, Anqi Guo, Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Justin T Broaddus, Ryan Marshall, Anthony Skjellum, and Martin C Herbordt. 2020. FPGAs in the Network and Novel Communicator Support Accelerate MPI Collectives. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, IEEE, 1–10.

[22] D. A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sep. 1952), 1098–1101.

[23] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, Vol. 22. Wiley Online Library, 343–348.

[24] Sian Jin, Pascal Grosset, Christopher M Biwer, Jesus Pulido, Jiannan Tian, Dingwen Tao, and James Ahrens. 2020. Understanding GPU-based lossy compression for extreme-scale cosmological simulations. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 105–115.

[25] Sian Jin, Jesus Pulido, Pascal Grosset, Jiannan Tian, Dingwen Tao, and James Ahrens. 2021. Adaptive Configuration of In Situ Lossy Compression for Cosmology Simulations via Fine-Grained Rate-Quality Modeling. *arXiv preprint arXiv:2104.00178* (2021).

[26] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel programming for FPGAs. *arXiv preprint arXiv:1805.03648* (2018).

[27] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2020. Datasets for Benchmarking Floating-Point Compressors. *arXiv preprint arXiv:2011.02849* (2020).

[28] Hemanth Kolla. 2019. https://sdrbench.github.io/. Online.

[29] Sohan Lal, Jan Lucas, and Ben Juurlink. 2017. Eˆ 2MC: Entropy Encoding Based Memory Compression for GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, 1119–1128.

[30] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. (2018).

[31] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[32] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 1245–1250.

[33] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. 15–24.

[34] Tao Lu, Qing Liu, Xubin He, Huizhang Luo, Eric Suchyta, Jong Choi, Norbert Podhorszki, Scott Klasky, Mathew Wolf, Tong Liu, et al. 2018. Understanding and modeling lossy compression schemes on HPC scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, 348–357.

[35] Huizhang Luo, Dan Huang, Qing Liu, Zhenbo Qiao, Hong Jiang, Jing Bi, Haitao Yuan, Mengchu Zhou, Jinzhen Wang, and Zhenlu Qin. 2019. Identifying latent reduced models to precondition lossy compression. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, 293–302.

[36] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhisa Sato. 2020. Performance Evaluation of Supercomputer Fugaku using Breadth-First Search Benchmark in Graph500. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, IEEE, 408–409.

[37] Tan Nguyen, Samuel Williams, Marco Siracusa, Colin MacLean, Douglas Doerfler, and Nicholas J Wright. 2020. The performance and energy efficiency potential of fpgas in scientific computing. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 8–19.

[38] NWChem - Open Source High-Performance Computational Chemistry. 2019. https://nwchemgit.github.io/. Online.

[39] Andrew Poppick, Joseph Nardi, Noah Feldman, Allison H Baker, Alexander Pinard, and Dorit M Hammerling. 2020. A statistical analysis of lossily compressed climate model data. *Computers & Geosciences* 145 (2020), 104599.

[40] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

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

[42] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. Data compression for the exascale computing era-survey. *Supercomputing Frontiers and Innovations* 1, 2 (2014), 76–88.

[43] Summit. 2021. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. Online.

[44] Gongjin Sun, Seongyoung Kang, and Sang-Woo Jun. 2020. BurstZ: a bandwidth-efficient scientific computing accelerator platform for large-scale data. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[45] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. In-depth exploration of single-snapshot lossy compression techniques for N-body simulations. In *2017 IEEE International Conference on Big Data*. IEEE, IEEE, 486–493.

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

[47] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. 2019. Optimizing Lossy Compression Rate-Distortion from Automatic Online Selection between SZ and ZFP. *IEEE Transactions on Parallel and Distributed Systems* (2019).

[48] Jiannan Tian, Sheng Di, Chengming Zhang, Xin Liang, Sian Jin, Dazhao Cheng, Dingwen Tao, and Franck Cappello. 2020. Wavesz: A hardware-algorithm co-design of efficient lossy compression for scientific data. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 74–88.

[49] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. (2020), 3–15.

[50] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. 2021. 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, 881–891.

[51] Vitis Libraries. 2020. https://github.com/Xilinx/Vitis_Libraries/. Online.

[52] Vitis Unified Software Platform. 2020. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html. Online.

[53] Vivado-ug. 2021. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf. Online.

[54] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. 2017. Analysis and modeling of the end-to-end i/o performance on olcf's titan supercomputer. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, IEEE, 1–9.

[55] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. 2017. Comprehensive measurement and analysis of the user-perceived I/O performance in a production leadership-class storage system. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, IEEE, 1022–1031.

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

[57] Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Anthony Skjellum, and Martin C Herbordt. 2019. Ghostsz: A transparent fpga-accelerated lossy compression framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, IEEE, 258–266.

[58] ZHW - ZFP Hardware Implementation. 2021. https://github.com/LLNL/zhw/. Online.