Lightweight Huffman Coding for Efficient GPU Compression

Milan Shah
North Carolina State University
Argonne National Laboratory
Raleigh, NC, USA
mkshah5@ncsu.edu

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

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

Michela Becchi
North Carolina State University
Raleigh, NC, USA
mbecchi@ncsu.edu

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

ABSTRACT

Lossy compression is often deployed in scientific applications to reduce data footprint and improve data transfers and I/O performance. Especially for applications requiring on-the-flight compression, it is essential to minimize compression's runtime. In this paper, we design a scheme to improve the performance of cuSZ, a GPU-based lossy compressor. We observe that Huffman coding - used by cuSZ to compress metadata generated during compression - incurs a performance overhead that can be significant, especially for smaller datasets. Our work seeks to reduce the Huffman coding runtime with minimal-to-no impact on cuSZ's compression efficiency.

Our contributions are as follows. First, we examine a variety of probability distributions to determine which distributions closely model the input to cuSZ's Huffman coding stage. From these distributions, we create a dictionary of pre-computed codebooks such that during compression, a codebook is selected from the dictionary instead of computing a custom codebook. Second, we explore three codebook selection criteria to be applied at runtime. Finally, we evaluate our scheme on real-world datasets and in the context of two important application use cases, HDF5 and MPI, using an NVIDIA A100 GPU. Our evaluation shows that our method can reduce the Huffman coding penalty by a factor of 78-92×, translating to a total speedup of up to 5× over baseline cuSZ. Smaller HDF5 chunk sizes enjoy over an 8× speedup in compression and MPI messages on the scale of tens of MB have a 1.4-30.5× speedup in communication time.

*Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA

ICS '23, June 21–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593736

CCS CONCEPTS

• **Information systems** → **Data compression**; • **Computing methodologies** → *Parallel algorithms.*

KEYWORDS

compression, Huffman coding, GPU

1 INTRODUCTION

Scientific applications running on high-performance computing clusters generate large amounts of floating-point data and require efficient I/O and data transfers to meet the demands of large-scale simulations. Large amounts of floating-point data present significant costs in terms of storage and power. In addition, efficient data transfers are instrumental to the scalability of parallel scientific applications since large communication overhead inhibits application speedup as data size and parallelism increase. To address these two requirements, compression can be integrated into data pipelines to reduce data footprint and improve performance of costly data transfers and I/O operations.

Lossless compressors [3, 4, 6, 24] ensure that data that have been compressed can be accurately restored after decompression, but struggle to achieve high compression ratios on floating point data [21]. Many scientific applications can tolerate some degree of inaccuracy, allowing lossy compression to shine as a high compression ratio and high throughput alternative to lossless compression. Lossy compressors often allow users to supply an error bound, such that data

distortion due to compression does not greatly impact subsequent data analysis. cuSZ [17] is a GPU-accelerated lossy compressor, and it consists of three compression stages: prediction, quantization, and Huffman coding. Huffman coding, a lossless compression algorithm, is performed on metadata generated in the second stage, called *quantization codes*, to further improve the compression ratio of cuSZ.

Huffman coding [8] compresses data by assigning variable-length encodings to each symbol in the input data, based on the input symbols' frequency. The encoding assignment involves building a Huffman tree, where each symbol is represented by a leaf node and each path from root to leaf corresponds to the symbol's encoding. The resulting codebook mapping symbols to variable-length codes is used to encode the input data. While there exist parallel implementations of Huffman coding [12][18], pre-computed codebooks have been proposed as a means of offloading codebook generation [1][20]. However, these prior pre-computed codebook designs either have not focused on limiting the compression ratio degradation, or have not targeted the specific requirements of compressors such as cuSZ.

In this work, we design a pre-computed Huffman codebook scheme for cuSZ. Our design seeks to greatly reduce the overhead of Huffman codebook generation through maintaining a dictionary of pre-computed codebooks, while simultaneously limiting compression ratio degradation from using a custom codebook.

Our contributions are:

- We use maximum likelihood estimation with a variety of probability distributions to closely model quantization code distributions, such that pre-computed codebooks are tailored to cuSZ metadata. The resulting codebooks can be used in a dictionary accessed during compression, enabling compression ratios comparable to a custom codebook (i.e., a codebook specific to the considered input data).
- We evaluate various selection criteria to access the dictionary at runtime in an effort to find a pre-computed codebook that best fits the input data. The selection criteria are calculated during compression and require significantly less time to compute than a custom Huffman codebook.
- We perform design space exploration for sizing the dictionary and targeting GPU, quantifying the effects of using our design on GPU resources.
- We evaluate our design using an NVIDIA A100 GPU, studying the effect of our pre-computed codebook dictionary on compression ratio across four scientific datasets. We additionally explore the compression performance of using our design and the performance impact in two real-world use cases: HDF5 and MPI.

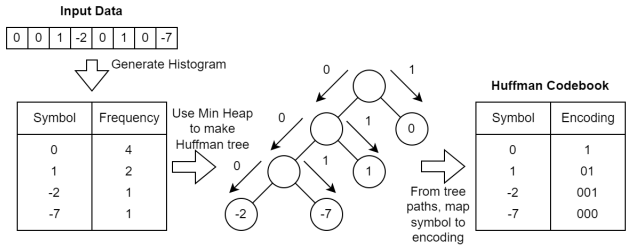


Figure 1: Huffman codebook generation algorithm

Our evaluation indicates that our method improves the performance of cuSZ's Huffman coding step by a factor 78-92× while limiting compression ratio degradation to less than 4%. HDF5 chunking with compression enjoys up to 8× speedup for smaller chunk sizes compared to baseline cuSZ and MPI communication time is sped up by a factor 1.4-30.5×.

2 BACKGROUND

2.1 Compression with cuSZ

Lossy compression is often used in scientific applications to reduce floating-point data footprint. In contrast with lossless compression, which perfectly reconstructs data when decompressed, lossy compression introduces distortions to data such that decompressed data may not be exactly equal to the data prior to compression. While lossy compression can reduce data quality, it can significantly increase compression ratios for floating-point data compared to lossless counterparts [21]. Scientific floating-point data often have a tolerance for error, enabling the use of a lossy compressor to improve performance and compression ratio.

cuSZ [17] is a state-of-the-art lossy compressor, developed for GPU. Compared to other GPU-based lossy compressors, such as cuZFP, cuSZ achieves 2.41-3.48× higher compression ratios and is the first error-bounded GPU lossy compressor with user-specified error [17]. Based on the SZ lossy compressor [10], cuSZ has three main stages: (1) data prediction, (2) linear-scale quantization based on a user-specified error bound, and (3) encoding quantization codes with Huffman coding, a lossless compression technique. The first stage involves predicting data points based on previous data points. SZ implements various predictors depending on the data dimensionality; the results from these predictors influence the effectiveness of the second and third stages. In the second stage, predicted data points are quantized with respect to both the given error bound and the original data points. For each data point, the quantization code maps the data point's prediction to a region of values within the error bound of the original point. This stage yields quantization codes that must be compressed further. The third stage performs lossless compression on the quantization codes. Lossless compression

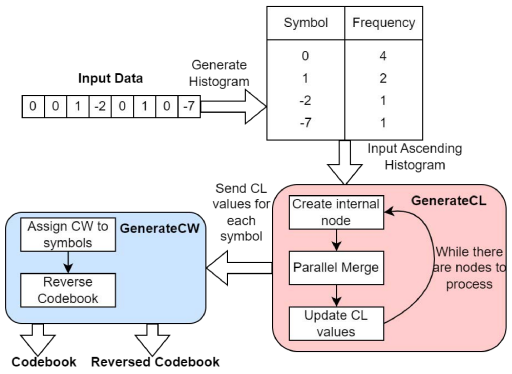


Figure 2: Huffman codebook generation algorithm for GPU

is required because the quantization codes themselves are metadata that have no tolerance for error. If lossy compression was used at this stage, decompressed data points may not respect the user-specified error bound since their quantization code may not map to a value within the error bound. It is at this stage that Huffman encoding is used. cuSZ is configured to have a user-specified quantization code range. If a data point's quantization code is not within this range, it is treated as an outlier and the data point is stored instead of being compressed with Huffman coding. During decompression, cuSZ first uses Huffman decoding to retrieve the quantization codes.

2.2 Huffman Coding

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to an input set of symbols. It is used in a variety of compressors, both lossless (ie. Zstd [6]) and lossy (MGARD [2]). In order to achieve compression, Huffman coding aims to assign shorter codes to more frequently occurring symbols. In the case of cuSZ, the input symbols are the quantization codes generated in the second step of the compressor. The encoding process begins with the creation of a Huffman tree based on the frequency of occurrence of the input symbols in the data being compressed. An Huffman tree is a binary tree that stores input symbols as leaf nodes with more frequent symbols residing higher in the tree (Fig. 1). The edges along the path from the root node to a symbol's leaf node correspond to the symbol's encoding. Typically, traversing to the left child of node indicates a "0" bit element of the code while traversing to the right child indicates a "1" bit. Since more frequent symbols have lower depth, their traversal path is shorter, yielding a smaller encoding length. Once the Huffman tree is generated, a codebook associating an encoding to each input symbol can be derived from it. That codebook will be used to compress the input.

Fig. 1 illustrates a serial implementation of Huffman coding. The algorithm consists of three steps: (1) input histogram generation, (2) Huffman tree's construction using Min Heap, and (3) codebook generation through Huffman tree traversal. Stage (2)'s Min Heap algorithm first places all the symbols in a priority queue, with priority inversely related to frequency. The first two queue elements are popped and an internal node is created as a parent of the two symbols' nodes. The internal node stores the sum of the two children's frequencies and is inserted back into the priority queue. This process is repeated until there is only one node in the queue: the root node. For n input symbols, this algorithm has $O(n \log n)$ time complexity. Ostadzadeh et al. [12] proposed a parallel Huffman codebook creation algorithm, later ported to GPU by Tian et al. [18] and incorporated in cuSZ. This parallel implementation, illustrated in Fig. 2, consists of two primary stages (each implemented in a GPU kernel): *GenerateCL* and *GenerateCW*. First, *GenerateCL* calculates the encoding length of each symbol, and then *GenerateCW* generates the actual encoding (i.e., the codebook) based on the lengths calculated in the *GenerateCL* step. This implementation includes two additional steps. First, the generated codebook is canonized [15], enabling Huffman decoding with only a reversed codebook while maintaining the same compression ratio. Secondly, the reversed codebook itself is generated, with a key-value pairing of (encoding, symbol). For H bits for the longest codeword, this GPU implementation has a practical time complexity of $O(H \log \frac{n}{H})$. This means that codebook generation time increases with the number of symbols and the depth of the Huffman tree.

3 MOTIVATION

The runtime cost of Huffman codebook's generation can affect the performance of cuSZ significantly, especially for small input datasets. Table 2 shows the fraction of cuSZ's compression time spent in Huffman codebook's creation for five datasets from SDRBench [22] and [14]. The size of these datasets and their application domain are summarized in Table 1. As can be seen, for these datasets codebook generation accounts for 29.6-81.7% of the compression time.

Table 1: Dataset Attributes

Dataset	Dimensionality	Size (MB)	Domain
CESM	1800x3600x1	24.7	Climate Simulation
NWChem	102953248x1x1	392.7	Electron Repulsion
Miranda	256x384x384	36.0	Hydrodynamics
QTensor	67108864x1x1	256.0	Quantum Computing
Nyx	512x512x512	512.0	Cosmology

Recall that, since cuSZ performs Huffman coding on the quantization codes, the codebook generation overhead does not depend on cuSZ's input data size, but on the quantization

Table 2: Fraction of cuSZ compression time spent in the codebook generation algorithm for quantization code range of (-512,512) (1023 symbols)

Dataset	Total Time (ms)	Codebook Gen. Time (%)
CESM	2.39	81.7
NWChem	5.90	33.2
Miranda	3.63	55.5
QTensor	4.91	46.2
Nyx	7.09	29.6

code set size. Therefore, if the range of quantization codes is fixed, the cost to generate the codebook remains the same independent of the input data size. While creating the quantization code histogram requires a scan through the input data, histogram generation is a high-throughput stage with negligible cost (and can be integrated in cuSZ's quantization stage). The number of quantization codes directly impacts the performance, but increasing the range of quantization codes (thus the set of input symbols for Huffman tree creation) can increase the compression ratio since there are fewer data points considered outliers in cuSZ.

Since the codebook generation overhead is non-negligible and accounts for a significant portion of cuSZ's runtime, our design seeks to avoid codebook generation altogether, opting for a dictionary of pre-computed codebooks with a fast codebook selection step. In this work, we examine the performance of our method on general scientific floating-point datasets and on applications relying on the HDF5 library and the MPI protocol. Many such applications can benefit from good compression performance in real time while operating on small-middle size datasets. While we note that the percent of time cuSZ spends in codebook generation becomes smaller as the input data size grows, compression for small- and medium-sized datasets on the scale of tens to hundreds of MB is a problem space that must be explored to yield high speeds for real-time applications.

3.1 Example applications

HDF5 is a data management and storage framework implemented as the combination of a model, a library, and a file format [7]. HDF5 data are organized in a hierarchical fashion where groups contain other groups or datasets, and datasets contain raw data. HDF5 chunking breaks a dataset into smaller partitions, or chunks, stored separately on disk. HDF5 chunking presents an attractive alternative to contiguous storage for applications that operate on only subsets of data at a time. When chunks are written to disk or loaded from disk, I/O filters, such as compression, can be implemented to operate on each individual chunk. Chunk compression reduces data footprint as well as disk I/O time [5, 9]. Since chunks have a maximum size of 4 GB and most often are on the scale of kilobytes to megabytes, compression filters must be able to perform well on small datasets. The cost of codebook creation in cuSZ can be prohibitive in the adoption of cuSZ as a compression filter for HDF5 since chunks are relatively small inputs for compression. Table 2 suggests that reducing the codebook generation time can significantly improve the performance of HDF5, leading to up to 80% faster compression and thus, less computationally-intensive HDF5 writes and reads from disk.

MPI applications' performance is often limited by communication costs. Data compression has been proven an efficient mechanism to limit data transfer overhead and improve the utilization of the interconnect bandwidth [5, 23]. Data sent via messages typically are in the range of kilobytes to megabytes in size, meaning that compressors must perform well on small data sizes. Compression introduces its own overhead in addition to communication, thus for scalable applications, minimizing communication overhead *and* compression overhead are crucial for high performance.

3.2 Problem Formulation

We formulate the research problem as follows. Given a dataset T, whose data points' original values are denoted as x_i, and whose set of quantization codes is denoted as $Q(T)$, our objective is to develop a fast pre-computed codebook scheme for $Q(T)$. This scheme should allow for quantization code compression ratios close to using an optimal Huffman codebook while improving the performance of cuSZ.

Compression ratio,*CR*, is defined as the ratio of the original raw data size to the compressed data size: $\frac{|T|}{|T'|}$, where T' denotes the decompressed data; $|T|$ and $|T'|$ represent the raw data size and compressed data size, respectively. The compression throughput is defined as $\frac{|T|}{\tau_C(T)}$, where $\tau_C(T)$ refers to the compression time of the raw dataset.

Specifically, our pre-computed codebook compressor should address two important objectives: minimizing the compression ratio difference between the optimal and pre-computed Huffman codebook and maximizing throughput, which are formulated as Formula (1) and Formula (2), respectively.

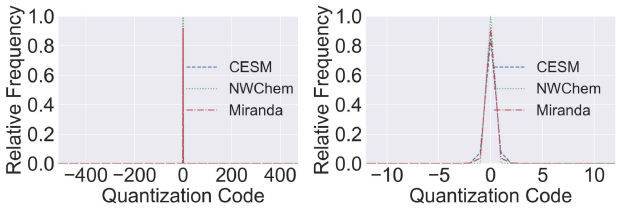
$$\begin{aligned} &\min |CR_{custom} - CR_{pre-computed}| \\ &s.t.\ \ |x_i - x_i'| \leq \epsilon, \forall x_i \in T, \text{WITH ERROR BOUND } \epsilon \end{aligned} \tag{1}$$

$$\begin{aligned} &\max \left(\frac{|T|}{\tau_C(T)}\right) \\ &s.t.\ \ |x_i - x_i'| \leq \epsilon, \forall x_i \in T, \text{WITH ERROR BOUND } \epsilon \end{aligned} \tag{2}$$

4 DESIGN

4.1 Overview

In order to avoid the runtime cost of Huffman codebook generation, we propose building a dictionary of pre-computed Huffman codebooks and selecting at compression time a "best-fit" codebook for the input data. The pre-computed



(a) Full Distribution **(b) Zoomed-In Distribution**

Figure 3: Distributions of quantization codes for CESM, NWChem, and Miranda; radius=512, R2R error bound of 0.01. QTensor and Nyx have similar distributions, thus are omitted.

Huffman codebooks are generated based on probability distributions that closely resemble common quantization code distributions. In the following subsections, we will discuss: 1) the pre-computation of Huffman trees that model various quantization code distributions, 2) criteria to select a codebook from the dictionary at compression time, 3) GPU-specific implementation details, and 4) design space exploration for sizing the dictionary.

4.2 Creating Huffman Trees

As discussed in Section 2, the input to Huffman tree/codebook generation is the histogram of quantization codes. Thus, if one could accurately predict the histogram of quantization codes for a given dataset, using a pre-computed Huffman codebook would not affect compression accuracy. Since we cannot perfectly predict the histograms of all datasets, we propose pre-computing Huffman codebooks from histograms that model representative distributions of quantization codes. If we divide the probability density function of some common distributions into n partitions, taking the integral for each partition can yield a relative frequency histogram of n bins. For cuSZ, this approach can be applied to distributions modeling quantization code frequency, yielding histograms to generate pre-computed Huffman codebooks. Fig. 3 shows the quantization code distributions for the CESM, NWChem, and Miranda datasets summarized in Table 1. QTensor and Nyx have similar distributions to these three, thus they are omitted. The three datasets have a user-specified relative-to-value-range (R2R) error of 0.01. R2R error r translates to an absolute error bound ϵ as $\epsilon = r(x_{max} - x_{min})$, where x_{max} and x_{min} are the maximum and minimum data points in the input dataset, respectively.

Table 3: Average PDF mean squared error of probability distributions w.r.t five datasets in Table 1. The top three performers are Cauchy, Laplace, and Gaussian distributions.

Distribution	Average PDF MSE	Standard Dev.
Cauchy	**4.2E-06**	3.7E-06
Laplace	**4.5E-06**	8.4E-05
Gaussian	**3.5E-05**	2.7E-05
Chi-square	7.7E+07	1.4E+08
Weibull	2.5E-01	4.9E-01
Exp. Weibull	1.1E+35	2.1E+35
Gamma	4.2E+38	8.4E+38

To determine probability distributions that model quantization codes well, we utilize a parameter-fitting method known as **Maximum Likelihood Estimation**, or **MLE**. MLE is a method of estimating the *parameters* of a statistical model given some observed data. Given a probability distribution and an input dataset, MLE returns the parameters for the distribution that yield a model closest to the input data's distribution. Repeating MLE on the same input data with varying distributions will yield the best-fit model for each distribution. The resulting models that can subsequently be compared against each other using other statistical metrics.

We use the SciPy library for Python [19] to perform MLE with 21 probability distributions on the quantization codes of the datasets in Table 1. Table 3 reports the average and standard deviation mean-squared error of the top seven performing distributions' probability density functions (PDFs) with respect to the datasets. Our results indicate that the three distributions modeling quantization code frequencies the best are the Cauchy, Laplace, and Gaussian distributions.

The standardized form of the Cauchy, Laplace, and Gaussian distributions' PDFs are shown in Eq. 3, 4, and 5, respectively.

$$f_{cauchy}(x) = \frac{1}{\pi(1+x^2)} \tag{3}$$

$$f_{laplace}(x) = \frac{1}{2}\exp(-|x|) \tag{4}$$

$$f_{gaussian}(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}} \tag{5}$$

For a given PDF $f(x)$, the PDF can be shifted by L and scaled by S to yield a final PDF $F(x)$ as follows:

$$F(x) = \frac{f(\frac{x-L}{S})}{S} \tag{6}$$

Using Eq. 6, altering S can create a variety of Huffman codebooks from these three distributions. For this work, L is fixed at zero since all quantization code distributions are centered around zero. Increasing S will increase the spread of the distribution, meaning quantization codes further from zero will have an increase in relative frequency, resulting in a more balanced Huffman tree. Quantization codes further

from zero will benefit from shorter encoding lengths while codes closer to zero suffer from longer encoding lengths.

To generate a final Huffman codebook for the dictionary, we first select one of the three distributions, divide the PDF into as many regions as there are quantization codes (typically between 512-2048) to create a histogram, then feed the histogram as input to the Huffman coding algorithm to generate a codebook. This process is repeated for different values of S, resulting in a dictionary of codebooks.

4.3 Tree Selection Criterion

Given a set of codebooks, the compressor must select the one that fits best the quantization codes of the data being compressed. The selection criterion should have the following qualities: 1) it should identify a codebook that yields the optimal or close-to-optimal compression ratio for the input data, 2) it should rely on some data inherent to the pre-created codebooks, and 3) it should be faster to compute than the codebook generation algorithm. Here, we examine three selection criteria: *Shannon's entropy*, *cross entropy*, and *Kullback–Leibler (KL) divergence*.

Shannon's Entropy gives the optimal average number of bits to encode a random variable from a given probability distribution. For an n bin histogram and relative frequency p_i for each bin i, entropy H is defined in Eq. 7.

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{7}$$

Using Shannon's entropy as selection criterion requires computing, at compression, the entropy of the quantization codes' distribution for the input data. The result is then compared against the entropy of all pre-computed codebooks, and the codebook with the *closest entropy* is selected. Increasing the scale factor S when creating pre-computed codebooks increases the entropy of the distribution.

Cross Entropy is closely related to Shannon's entropy since it gives an average number of bits to encode a variable, with the key difference being that the coding scheme is optimized for a *different* distribution. That is, for an n bin histogram, relative frequency p_i for the dataset being encoded, and relative frequency q_i for the pre-computed codebook's distribution, cross entropy CE is defined in Eq. 8.

$$CE = -\sum_{i=1}^{n} p_i \log_2 q_i \tag{8}$$

This criterion accounts for the distribution of both the input data and pre-computed codebook, unlike entropy alone. As such, cross entropy can discriminate a "best-fit" codebook relative to the input data. When $q(x) = p(x)$, cross entropy becomes entropy, providing a lower bound for the average number of bits to encode the data.

To select a codebook using cross entropy, cross entropy is first calculated based on the histograms of the input data quantization codes and the codebook's input histogram. Then, the codebook yielding the *smallest cross entropy* is selected for Huffman encoding.

KL Divergence, also called *relative cross entropy*, is closely related to cross entropy since it measures the similarity between two distributions. If two distributions are identical, their KL divergence is zero. For an n bin histogram, relative frequency p_i for the dataset being encoded, and relative frequency q_i for the pre-computed codebook's distribution, KL divergence KL is defined in Eq. 9.

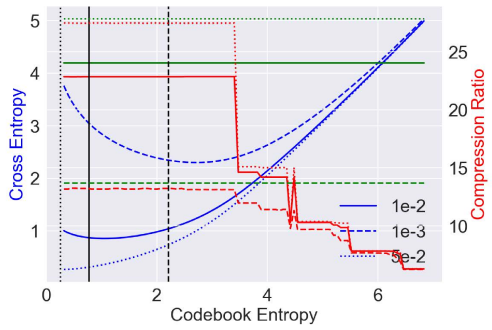
$$KL = -\sum_{i=1}^{n} p_i \log_2 \frac{q_i}{p_i} \tag{9}$$

KL divergence is often used in machine learning applications as a loss function and can be applied here as a measure of compressibility "loss". Note that KL divergence is not a metric due to its asymmetry and is dependent on the reference distribution.

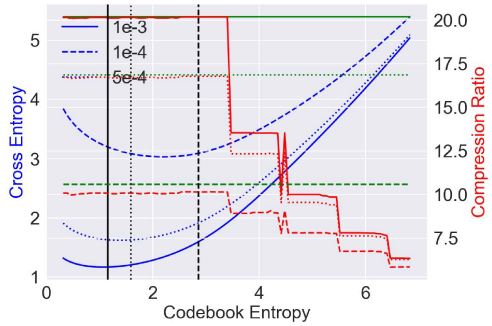
KL divergence yields a nearly identical shape to cross entropy when varying the entropy of the codebooks. As such, the minimum KL divergence corresponds to the same codebook as the minimum cross entropy. Both of these find a codebook that minimizes the reduction in compression ratio from using a custom Huffman codebook, but KL divergence requires one additional division and prevents the logarithm calculation beforehand. Using cross entropy, each dictionary key can hold the $\log_2 q_i$ terms instead of only the q_i terms for a codebook, further offloading runtime computation. Since cross entropy is less computationally costly, we use cross entropy over KL divergence as the selection criterion.

Example: Fig. 4 compares using entropy and cross entropy as selection criteria for CESM and NWChem. The other three datasets exhibit similar behavior and their plots are omitted for space. We generated 100 codebooks with varying S, thus varying the entropy from approximately 0 to 7. The optimal compression ratio when using a custom codebook is plotted in green while pre-computed codebook compression ratios are plotted in red. Cross entropy, plotted in blue, can yield a global minimum value that coincides with the high compression ratio regions of the plots.

If Shannon's entropy were the selection criteria, the codebook with the closest entropy to the quantization code histogram entropy (plotted as black lines in Fig. 4) does coincide with the high compression ratio regions for CESM and NWChem at these error bounds. However, if the dictionary does not have enough codebook entries, the nearest entropy codebook may reduce the compression ratio compared to cross entropy selection. For example, if we had a dictionary of two Cauchy-based codebooks with entropies of 1.7 and 4.0,



(a) CESM



(b) NWChem

Figure 4: Cross Entropy and Entropy effects on Compression Ratio using Cauchy distribution for CESM and NWChem. Each line style is associated with an absolute error bound. The black lines denote the quantization code entropy. Achieved compression ratio is in red and on right axis. Cross entropy is blue and on left axis. The green line denotes cuSZ compression ratio when using a custom Huffman codebook.

and an error bound of 1E-4 (entropy of 2.9) for compressing NWChem, the 4.0 codebook has the nearest entropy while the 1.7 codebook has the minimum cross entropy. Fig. 4b indicates that the 1.7 codebook would achieve higher compression ratio in this instance. For this reason, we opt for cross entropy over Shannon's entropy as the selection criterion since it discriminates among many codebooks and directly accounts for the input quantization codes.

4.4 GPU-based Design

Fig. 5 illustrates our GPU-based implementation, written in CUDA C/C++. The blue boxes indicate host code for managing data and the red boxes indicate kernels run on the GPU. When using cross entropy as the selection criterion, the dictionary holds the following for each codebook: $\log_2 q_i$ values as keys, codebook as one value and the reversed codebook as the second value. Since $\log_2 q_i$ values are pre-computed,

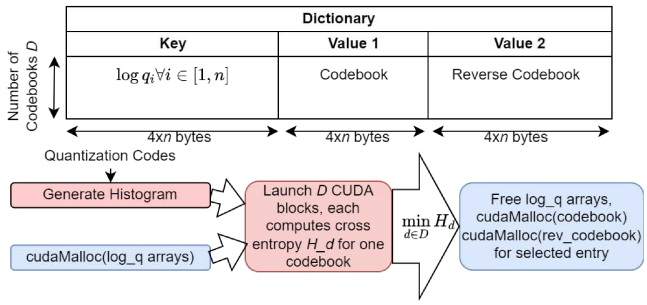


Figure 5: CUDA-based algorithm

the GPU kernel performs a fast multiply-accumulate operation. Each block is assigned one codebook and reads the quantization code histogram and the array storing the codebook's $\log_2 q_i$ values in a coalesced fashion. Before this kernel, only the $\log_2 q_i$ array needs to be additionally allocated on GPU global memory. Each thread is assigned one bin i and performs the multiplication $p_i \times \log_2 q_i$. After a barrier synchronization, an accumulate operation is performed and the resulting cross entropy is compared against all other cross entropy values to find the minimum. The minimum cross entropy codebook is selected and its codebook and reverse codebook are brought onto GPU for the encoding phase.

In terms of memory requirements, the pre-computed Huffman codebook dictionary requires relatively little space. For D codebooks and a quantization code range of $(-n/2, n/2)$, the dictionary keyset and each value set occupy $4 \times n \times D$ bytes. If the quantization code radius is 512 and there are 100 codebooks in the dictionary, the entire dictionary occupies about 1.17 MB, and at most only 400 KB is used on GPU memory since calculating the cross entropy only requires all $\log_2 q_i$ values at once. GPU memory is sized typically on the scale of tens of GBs, thus the dictionary occupies less than 0.1% of global memory.

4.5 Design Space Exploration

There are two primary design parameters for sizing the dictionary appropriately: 1) the number of codebooks that compose the dictionary, and 2) the quantization code range. On an NVIDIA A100 GPU, the cross entropy kernel takes about 0.023 ms, with little variation across the number of codebooks or the R2R error bound. A 100-codebook dictionary takes the longest, taking 0.025 ms. Fig. 6 explores the effect of varying the number of codebooks from 5 to 100 for the CESM dataset. Specifically, Fig. 6 plots the compression ratio relative to that achieved using 100 codebooks. As the number of dictionary entries increases, the entropy space becomes more finely sampled, leading to more codebook choices at runtime and better compression ratio. However, increasing the number of entries requires more memory and

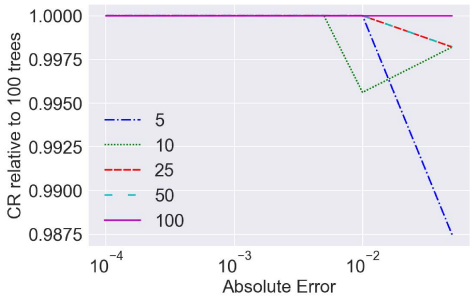


Figure 6: Compression ratio for NWChem dataset with varying R2R error bound and number of codebooks in the dictionary. The legend corresponds to the number of codebooks for each series plotted. The compression ratio is relative to that achieved using 100 codebooks.

at some point, negatively impacts performance. Having too few entries may hurt compression ratio since the dictionary may not have a close-to-optimal codebook. According to Fig. 6, having the number of entries around 25 achieves good performance and reduces the memory footprint while not greatly sacrificing compression ratio. For this reason and since the performance is similar across datasets, we size our dictionary to have 25 entries.

The quantization code range is specified by the user when calling cuSZ's compression API. Since the dictionary must be pre-computed, the number of bins for each distribution must be selected prior to use. We find that having a radius of 512 for quantization codes is suitable for the datasets we tested since there is little to no improvement in compression ratio as we increase the radius beyond 512. Thus, our dictionary assumes 1023 bins and the Cauchy, Laplace, and Gaussian PDFs are partitioned into 1023 segments for codebook generation. Larger radius values increase the runtime of both the cross entropy kernel and custom Huffman codebook kernel. The dictionary used in the evaluation ultimately occupies 300 KB, with only 100 KB at most allocated on GPU global memory.

5 EXPERIMENTAL EVALUATION

5.1 Methodology

We perform our experiments on a system equipped with an AMD EPYC 7742 64-core CPU and an NVIDIA A100 GPU, which features 64 FP32 cores per streaming multiprocessor (SM), 108 SMs, and a global memory size of 40 GB. We test the cross entropy kernel runtime against the codebook generation runtime using the CESM, NWChem, Miranda, and QTensor datasets in Table 1. To test the performance of our design in real-world use cases, we use our implementation as a compression filter for HDF5 chunking and as a compressor

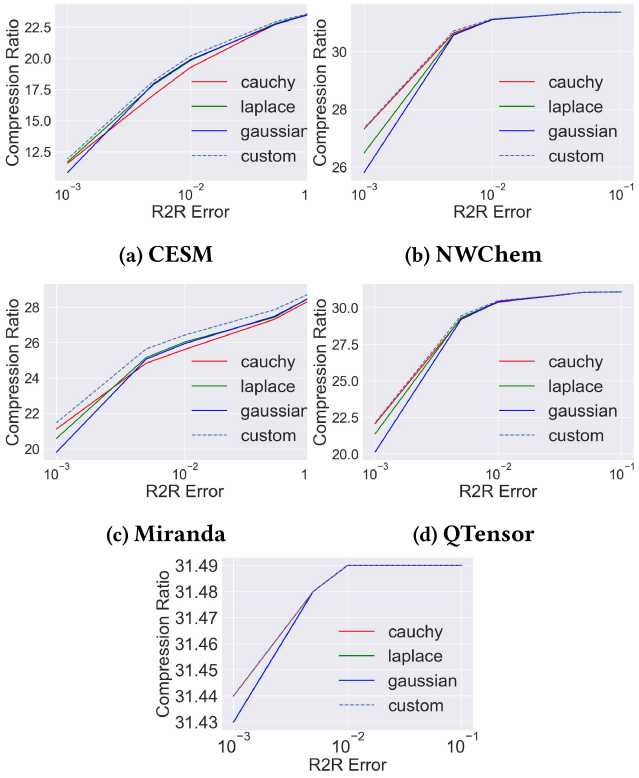


Figure 7: cuSZ compression ratio of Cauchy-, Laplace-, and Gaussian-based pre-computed Huffman codebooks. Custom Huffman codebook is plotted as a dashed line.

Table 4: Average codebook generation ("Book") time and cross entropy ("CE") kernel time for five datasets over 100 trials. Speedup is the ratio of codebook time to CE time.

Dataset	Book Time (ms)	CE Time (ms)	Speedup
CESM	1.815	0.023	78.2×
NWChem	1.735	0.023	74.1×
Miranda	1.782	0.023	76.3×
QTensor	2.166	0.024	91.7×
Nyx	2.064	0.022	93.8×

in an MPI application. For these use cases, we use the Nyx dataset described in Table 1 since it is stored in HDF5 file format and we can vary the data size easily. We use CUDA 11 and, for MPI experiments, the OpenMPI library [11].

5.2 Results

Raw Data Fig. 7 plots the compression ratio using pre-computed codebooks based on the Cauchy, Laplace, and Gaussian distributions with varying R2R error for the datasets

in Table 1. The "custom" series corresponds to the compression ratio of cuSZ when using the Huffman codebook generation algorithm at runtime instead of pre-computing the codebook, providing a baseline for performance.

Fig. 7 indicates that pre-computed codebooks for these three distributions have a limited impact on the final cuSZ compression ratio. As the R2R error bound increases, the pre-computed codebooks converge closer to optimal cuSZ compression ratio. For error bounds of 0.01 or larger, pre-computed codebook compression ratios are less than 4% lower than custom codebook compression ratios. For smaller R2R error bounds, such as 0.001, the pre-computed codebooks have a slightly more variable impact on compression ratio. For instance, Gaussian-based codebooks attain compression ratios 5-10% less than the custom codebook. Since increasing the error bound increases the entropy and "smoothness" of the quantization code histogram, common distribution PDFs can better model the histogram as they tend to avoid large relative frequency variations for points close to the distribution center. Of the three distributions, Gaussian codebooks tend to incur the greatest degradation in compression ratio. The Gaussian distribution has a shallower gradient for points approaching the distribution center, leading to longer encodings for quantization codes close to zero, which can negatively impact compression ratio.

Table 4 reports the execution time for the cross entropy kernel and custom codebook algorithm. Since both kernels are dependent on the number of quantization bins, and this value is fixed, their runtime is constant regardless of varying R2R error. The cross entropy kernel is 78-92× faster than custom codebook generation for the four datasets, greatly

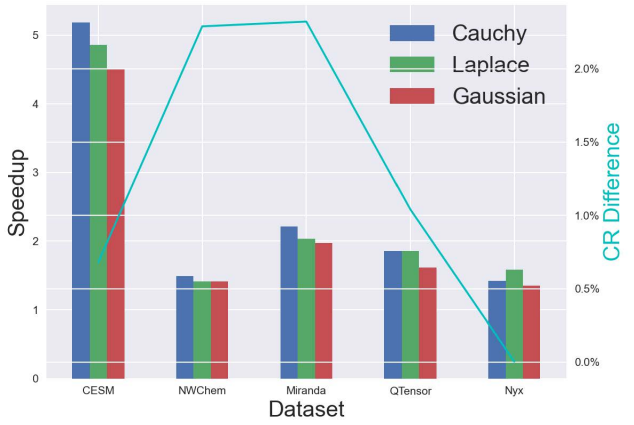


Figure 8: cuSZ with pre-computed codebook total time speedup over baseline cuSZ for five datasets, averaged over varying R2R error from $[0.001, 0.1]$. Codebook distribution is varied. The average compression ratio (CR) difference from baseline cuSZ for each dataset is plotted on the right y-axis (cyan).

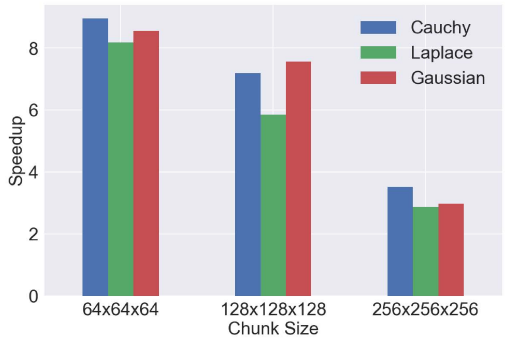
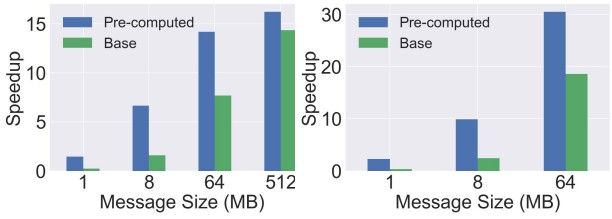


Figure 9: cuSZ with pre-computed codebook total time speedup over baseline cuSZ for varying HDF5 chunk sizes of Nyx dataset.

impacting performance for datasets on the scale of tens of MBs or less. CUDA data management APIs like *cudaMalloc* and *cudaMemcpy* are completely masked in pre-processing or during kernel execution, thus data movement times are not explicitly timed. Fig. 8 plots the speedup in cuSZ compression time when using pre-computed codebooks against custom codebooks, as well as the average compression ratio degradation. Across all datasets, compression ratio degradation was typically less than 3%. CESM, a 24.7 MB dataset, enjoys about a 5× speedup in compression runtime, while larger datasets like NWChem benefit less with about a 1.4× speedup. This speedup effect is due to Huffman codebook generation being a fixed cost independent of the data size. The main factor affecting codebook creation time is the quantization code range, fixed with a radius of 512. We expect that increasing the data size will lead to diminishing returns for pre-computed codebooks, in line with the results reported in Fig. 8.

HDF5 Compression for HDF5 chunks reduces data footprint and consumes less disk I/O bandwidth for each chunk, but costly compression operations, such as Huffman coding, must be accelerated to achieve these benefits. In Fig. 9, we report the total compression time speedup for pre-computed codebooks over the custom codebook algorithm. We use the Nyx dataset, described in Table 1, since it is typically stored in the HDF5 file format. Nyx is composed of many different fields, each with its own raw data in the shape 512x512x512.

We vary the HDF5 chunk size from 64x64x64 to 128x128x128 to 256x256x256. Since the data is 512x512x512, these chunk sizes respectively correspond to 512, 64, and 8 chunks. For the smallest chunk size tested, 64^3, pre-computed codebooks can lead to over 8× speedup in total compression time over custom codebooks. With 512 chunks for a 64^3 chunk size, the first time the entire dataset is written to disk would require 512 compression runs, requiring a high performance compressor. Since codebook creation time dominates runtime



(a) Point-to-Point **(b) Scatter**

Figure 10: cuSZ with pre-computed codebook (Pre-computed) and custom codebook (Base) total time speedup over no compression for varying MPI message sizes of Nyx dataset. Total time is defined as the time to compress and send the data.

for small data sizes, our proposed pre-computed codebook scheme has a significant impact on HDF5 chunking performance. Fig. 9 supports the idea that pre-computed codebook speedup diminishes as data size increases, since a chunk size of 256^3 has a more limited speedup of 3-4×.

MPI In order to test our method on MPI, we created two synthetic benchmarks: one using point-to-point and the other using collective communication primitives. In both cases, we use the Nyx dataset, we run 16 processes, and we set the R2R error bound to 0.005. All experiments are conducted on a single machine to isolate the effect of compression from interconnect performance. The first benchmark uses the *MPI_Send* and *MPI_Recv* primitives. Each process p sends a data message to the next process $p + 1$; the last process sends a message to the first. After receiving the message, each process performs a local reduction on it. Before sending the data, compression is performed to reduce the message size. Once received, the message is decompressed to yield the original data. The second benchmark performs a scatter operation, where one process (root) sends data to all other MPI processes. When using compression, the root process serially compresses equal-sized chunks of the input data and uses *MPI_Scatterv* to send compressed bytes to all other processes. Message size directly impacts communication time and consumes interconnect bandwidth, thus reducing message size can improve the scalability of MPI applications.

Fig. 10 reports the results of the first use case ("Point-to-Point"), and plots the communication speedup over using no compression to send a message with varying initial size. Specifically, we vary the size of a message generated from the Nyx dataset, increasing from 1 to 512 MB. The total communication time is the sum of compression time and *MPI_Send* time. Fig. 10a reports the results of two configurations: cuSZ with pre-computed codebooks averaged across Cauchy, Laplace, and Gaussian distributions, and "base" cuSZ (using custom codebook algorithm). We did not observe significant performance differences across distributions. The speedups reported are averages for the 16 processes launched. The APIs *MPI_Send* and *MPI_Recv* are blocking calls that return only when communication completes, thus we use these APIs to measure communication overhead. For 1 MB messages, compression with baseline cuSZ performs *worse* than sending uncompressed messages. However, using compression improves communication time in all configurations for message sizes >1 MB. Using pre-computed codebooks improves total communication time from 1.4 to 16.2× compared to custom codebook speedup of 0.25 to 14.3×.

Fig. 10b reports the communication speedup over using no compression for the second use case, "Scatter", with the same configurations as Fig. 10b. Again, we did not observe significant performance differences across probability distributions. In these experiments, we vary the size of the message received from 1MB to 8MB. When using compression, the actual message size transmitted is smaller, which reduces the *MPI_Scatterv* overhead. Even with serial compression on each message at the root node, using our pre-computed codebook dictionary improves communication performance 2.2-30.5×. Baseline cuSZ with custom codebook generation has a diminished effect, yielding speedup of 0.4-18.6×. In summary, compression with cuSZ in any form is well suited to improve communication time for larger message sizes, while pre-computed codebooks improve compression performance for smaller message sizes.

6 RELATED WORK

Ostadzadeh et al. [12] propose a parallel Huffman codebook generation algorithm that later, Tian et al. [18] adapt for GPU. Tian et al. focus on both the codebook generation phase and the encoding phase of Huffman coding and their implementation is used in the cuSZ pipeline. Compared to a serial CPU implementation, this GPU implementation can perform up to 45.5× faster than CPU, but only for 8192 symbols or more. For less symbols, corresponding to a smaller quantization code range, the GPU implementation can perform worse than a serial CPU implementation. Since many quantization codes ranges have a radius of 512, 1024, and 2048, our pre-computed codebooks have a large reduction in overhead and can be more suitable for GPU, a platform that performs worse with complex branch logic.

Patel et al. [13] parallelize three stages of the bzip2 pipeline: Burrows-Wheeler transform, move-to-front transform, and Huffman coding. Their Huffman tree generation algorithm uses parallel reduction to search for nodes with the smallest frequencies, but node merging cannot be parallelized. Thus, their implementation finds that move-to-front transform with Huffman coding is 1.34× slower than serial bzip2. Tian et al. [16] explore run-length encoding as an alternative to Huffman coding for quantization codes. Their findings

suggest that for very lower entropy histograms (average bit-length is less than 1.09), run-length encoding (RLE) can yield higher compression ratios with comparable performance. Huffman coding, in contrast with RLE, requires a codebook generation phase, which is the primary focus of our work. Additionally, Huffman coding yields higher compression ratios compared to RLE for quantization code histograms with entropy larger than 1.09.

Zhang et al. [20] implement a pre-computed Huffman codebook method for cuSZ's prediction and quantization. Their method targets FPGA and uses one pre-computed codebook, created from the average histogram of several scientific datasets. If the codebook does not lead to sufficiently high compression ratios, online codebook generation is performed. Our design differs in a few ways: 1) We use a dictionary of codebooks based on a variety of common probability distributions, 2) As the R2R error decreases, they report more significant compression ratio degradation (5.1-10.7% less) compared to the Cauchy and Laplace codebooks in our dictionary (approx. 4% less), 3) Online codebook generation can still occur, while our design relies completely on offline codebook generation, 4) The target platforms differ. Abali et al. [1] patented a general method of precomputing codebooks using a table storing the length of each symbol's encoding. At runtime, the dataset's values are assigned symbols with the corresponding length of each entry in the table and the overall bits required using each precomputed codebook is calculated. Their work relies on a sorting step after histogram generation and before selecting a table entry, a process that has additional overhead on GPU. Additionally, they do not perform a GPU- or cuSZ-specific design space exploration. Our work leverages properties of cuSZ quantization codes specifically, and we evaluate how to size the dictionary and quantify compression ratio degradation. The selection criteria between the patent and our work also differs since they calculate predicted bit length while we use cross entropy.

7 CONCLUSION

In this work, we have designed and evaluated a pre-computed Huffman codebook scheme to offload codebook generation during cuSZ compression. Using a dictionary of pre-computed codebooks, we can calculate the cross entropy of a quantization code distribution to select a best-fit codebook for Huffman encoding. Our evaluation on the A100 GPU suggests the following: **1)** Cauchy, Laplace, and Gaussian distributions can be leveraged to create a codebook dictionary, typically limiting compression ratio degradation to less than 4%, while achieving up to 5× total compression time speedup on MB-scale datasets over baseline cuSZ. **2)** HDF5 chunking with compression filters can enjoy a performance speedup over cuSZ of nearly 9×. Speedup benefits are multiplied by the number of chunks written to or read from disk. **3)** MPI communication time with our design has a speedup of 1.4-16.2× compared to baseline cuSZ speedup of 0.25-14.3× over using no compression prior to sending a message.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grants OAC-2003709, OAC-2104023 and CNS-1812727. We acknowledge the computing resources provided on Bebop (operated by Laboratory Computing Resource Center at Argonne) and on Theta and JLSE (operated by Argonne Leadership Computing Facility).

REFERENCES

[1] Bulent Abali, Bartholomew Balner, Hubertus Franke, and John J. Reilly. 2017. Creating a dynamic Huffman table.
[2] 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*.
[3] BlosC compressor. [n. d.]. http://blosc.org/. Online.
[4] M. Burtscher and P. Ratanaworabhan. 2009. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Trans. Comput.* 58, 1 (Jan 2009), 18–31. https://doi.org/10.1109/TC.2008.131
[5] 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. https://doi.org/10.1177/1094342019853336 arXiv:https://doi.org/10.1177/1094342019853336
[6] Yann Collet. 2015. Zstandard – Real-time data compression algorithm. *http://facebook.github.io/zstd/* (2015).
[7] HDF5. [n. d.]. https://portal.hdfgroup.org/display/HDF5/HDF5. Online.
[8] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898
[9] Sian Jin, Dingwen Tao, Houjun Tang, Sheng Di, Suren Byna, Zarija Lukic, and Franck Cappello. 2022. Accelerating Parallel Write via Deeply Integrating Predictive Lossy Compression with HDF5. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 61, 15 pages.
[10] 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. In *IEEE Big Data*. 438–447. https://doi.org/10.1109/BigData.2018.8622520

[11] OpenMPI. [n. d.]. https://www.open-mpi.org/. Online.

[12] SA Ostadzadeh, B Maryam Elahi, ZZ Tabrizi, M Amir Moulavi, and K Bertels. 2007. A two-phase practical parallel algorithm for construction of huffman codes. In *PDPTA 2007*. CSREA Press, 284–291.

[13] Ritesh A. Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D. Owens. 2012. Parallel lossless data compression on the GPU. In *2012 Innovative Parallel Computing (InPar)*. 1–9. https://doi.org/10.1109/InPar.2012.6339599

[14] Roman Schutski, Danil Lykov, and Ivan Oseledets. 2020. Adaptive algorithm for quantum circuit simulation. *Phys. Rev. A* 101 (Apr 2020), 042335. Issue 4. https://doi.org/10.1103/PhysRevA.101.042335

[15] Eugene S. Schwartz and Bruce Kallick. 1964. Generating a Canonical Prefix Encoding. *Commun. ACM* 7, 3 (mar 1964), 166–169. https://doi.org/10.1145/363958.363991

[16] Jiannan Tian, Sheng Di, Xiaodong Yu, Cody Rivera, Kai Zhao, Sian Jin, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data on GPUs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 283–293. https://doi.org/10.1109/Cluster48925.2021.00047

[17] Jiannan Tian and et al. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 3–15. https://doi.org/10.1145/3410463.3414624

[18] 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)*. 881–891. https://doi.org/10.1109/IPDPS49936.2021.00097

[19] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedrcegosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[20] 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 *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 12, 13 pages. https://doi.org/10.1145/3524059.3532362

[21] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1643–1654. https://doi.org/10.1109/ICDE51399.2021.00145

[22] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors. In *2020 IEEE International Conference on Big Data (Big Data)*. 2716–2724. https://doi.org/10.1109/BigData50022.2020.9378449

[23] Q. Zhou, C. Chu, N. S. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda. 2021. Designing High-Performance MPI Libraries with On-the-fly Compression for Modern GPU Clusters. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 444–453. https://doi.org/10.1109/IPDPS49936.2021.00053

[24] Zlib. [n. d.]. https://www.zlib.net/. Online.