Optimizing Huffman Decoding for Error-Bounded Lossy Compression on GPUs

Cody Rivera*, Sheng Di‡, Jiannan Tian†, Xiaodong Yu‡, Dingwen Tao†*, Franck Cappello‡
*Department of Computer Science, University of Alabama, Tuscaloosa, AL, USA
†School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA
‡Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA

***Abstract*—More and more HPC applications require fast and effective compression techniques to handle large volumes of data in storage and transmission. Not only do these applications need to compress the data effectively during simulation, but they also need to perform decompression efficiently for post hoc analysis. SZ is an error-bounded lossy compressor for scientific data, and cuSZ is a version of SZ designed to take advantage of the GPU's power. At present, cuSZ's compression performance has been optimized significantly while its decompression still suffers considerably lower performance because of its sophisticated lossless compression step—a customized Huffman decoding. In this work, we aim to significantly improve the Huffman decoding performance for cuSZ, thus improving the overall decompression performance in turn. To this end, we first investigate two state-of-the-art GPU Huffman decoders in depth. Then, we propose a deep architectural optimization for both algorithms. Specifically, we take full advantage of CUDA GPU architectures by using shared memory on decoding/writing phases, online tuning the amount of shared memory to use, improving memory access patterns, and reducing warp divergence. Finally, we evaluate our optimized decoders on an Nvidia V100 GPU using eight representative scientific datasets. Our new decoding solution obtains an average speedup of 3.64× over cuSZ's Huffman decoder and improves its overall decompression performance by 2.43× on average.**

I. Introduction

High-performance computing (HPC) applications are generating increasingly large amounts of data. For example, Hardware/Hybrid Accelerated Cosmology Code (HACC) is a cosmological simulation package designed for HPC environments. For HACC simulations consisting of trillions of particles, one snapshot of the simulation takes up 220 TB of data, while an entire simulation run can take up 22 PB of data [12]. Another such application is the pruning and compression of deep neural networks, which are becoming deeper and wider, and therefore larger, as computing capacity is increasing and deep neural networks are becoming more widely used [18]. However, leading supercomputers such as Summit [36] have limited storage capacities of approximately 50∼200 PB to share between hundreds of users. Thus, these applications require data reduction techniques that attain both high performance and high compression ratios. SZ, for instance, is a lossy data compressor that aims to achieve these goals. It offers over a 2x increase in compression ratio over state-of-the-art compressors. Furthermore, it allows users to specify how much error they wish to tolerate in their data and allows them to make a tradeoff between data distortion and compression ratios [37].

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

As a result of the evolution of supercomputer architecture (e.g., more powerful GPUs on a single node), many HPC applications are being implemented on graphics processing units (GPUs) due to their high performance and parallelism. For example, a recent study of cosmological simulations on the Summit supercomputer [33] shows a significant performance improvement compared to prior work without using GPUs [13]. However, even ignoring the time to transfer the uncompressed data from the GPU to the CPU, CPU-based lossy compressors would still cause more than 10% overhead of the overall performance, which would limit the I/O performance gain by lossy compression, according to prior studies [37, 19]. Thus, several lossy compressor development teams have recently released their GPU versions to reduce the compression overhead. These GPU versions can both accelerate the compression computation and reduce the time needed to transfer the data between the GPU and CPU after the compression. For example, both SZ and MGARD [1] have a GPU adaptation (known as cuSZ [39] and cuMGARD [5]), which have been implemented for GPU hardware, and more specifically Nvidia's CUDA platform.

Furthermore, an important use case of lossy compression is to reduce the memory footprint by storing the data in the compressed format and calculating from the lossy data in memory [4]. Compared to the amount of data handled in extreme-scale applications, memory is scarce on HPC systems. Lossy compression can be introduced to ease this pressure in applications that can tolerate some loss of fidelity in their working data. An example of in-memory compression is Wu *et al.*'s work on quantum circuit simulation [43], where in-memory compression reduced the total memory usage on 4,096 nodes from 32 exabytes to 768 terabyte. This compression allowed for 2∼16 more qubits to be simulated than if no compression was used. Another example is Jin *et al.*'s work on reverse time migration (RTM) simulation [17], where in-memory compression reduced the memory usage by about 10× on average. Moreover, for some applications, in-memory compression can decrease repetitive computations and accelerate execution [4]. For example, Gok *et al.*'s work on quantum chemistry simulation—GAMESS [10]—proposes to calculate, compress, and write each unique data block of two-electron integrals into the memory once; and whenever a block is needed again in simulations, it is read from the memory and decompressed.

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00075

Compared with the original GAMESS, where all blocks are generated and consumed by the simulation on the fly and are then deleted from the memory, Gok *et al.*'s approach achieves a reduction in the block re-computation costs. Note that all these computations should be done at runtime because integral blocks are generated and consumed repeatedly during a simulation. Consequently, in-memory (de)compression throughputs are critical to the overall performance. To this end, in this work, we focus on improving decompression throughputs on GPUs without considering GPU-to-CPU data-movement overheads, as decompressed data do not need to be transferred to CPUs or disks.

An important component of both cuSZ and cuMGARD is Huffman coding, a classic lossless compression technique initially developed by David Huffman in 1952 [14]. Tian *et al.*'s work proposes an optimized Huffman encoder for GPUs[40]; their work has been applied to improve cuSZ's compression throughput. Although efficient compression is important to speedup the overall data movement, efficient decompression is also important to enable fast and effective post-analysis based on compressed data. However, Huffman decoders used by error-bounded lossy compressors currently employ only a limited degree of parallelism and do not fully exploit the GPU's power. Two state-of-the-art works propose improved Huffman decoding on the GPU: one of these works, Weißenberger and Schmidt's [42], uses the self-synchronization property of Huffman codes to extract greater parallelism, while the other work, Yamamoto *et al.*'s [45], proposes a new data structure called a gap array to extract greater parallelism. But both works suffer from two main issues: (1) they do not fully take advantage of the GPU architecture for performance optimization, and (2) they must be adapted for use in error-bounded lossy compression.

To facilitate their efficient use in scientific data compression, we explore both Huffman decoding algorithms in depth. Furthermore, we identify opportunities for deep optimization of both algorithms based on GPU architecture considerations. Finally, we adapt both algorithms for use in error-bounded lossy compression such as cuSZ, and then evaluate them on eight representative scientific datasets. The contributions of our work are summarized as follows:

- We analyze Weißenberger and Schmidt's and Yamamoto *et al.*'s algorithms in depth by evaluating their performance on scientific datasets and understanding their tradeoffs.
- We perform a deep architectural optimization for both Huffman decoding algorithms by using shared memory in the decoding and writing phase, improving memory access patterns, and reducing warp divergence.
- We propose an efficient approach to online tune the amount of shared memory used to decode different parts of the data based on the data characteristics.
- We adapt our optimized decoders to multi-byte data for cuSZ and evaluate them on eight scientific datasets. Experiments show our solution can improve decoding throughput by 3.64×, compared with cuSZ's naïve decoder, and can improve cuSZ's overall performance by 2.43×, on average.

In §II, we present background information about scientific data compression, Huffman coding, and GPU-based lossy compression. In §III, we discuss both Weißenberger and Schmidt's and Yamamoto *et al.*'s Huffman decoding algorithms in detail, comparing them and discussing their limitations. In §IV, we describe our architectural optimizations for efficient Huffman decoding, as well as our adaptations to enable decoding of scientific datasets. In §V, we show the experimental evaluation results on scientific datasets. In §VI and §VII, we discuss the related work and conclude our work.

II. Background

A. Scientific Data Compression

Scientific data compression has been studied for decades and categorized into two types of compression: lossless compression and lossy compression. Lossless compressors such as FPZIP [29] and FPC [3] keep the data intact but can only provide a compression ratio of about 2× on scientific data [35]. Lossy compression, on the other hand, can compress data beyond lossless compression (typically one or more orders of magnitude) but introduces information loss in the reconstructed data. In recent years, a new generation of high accuracy lossy compressors for scientific data have been proposed and developed for scientific floating-point data, such as SZ [8, 37, 25], ZFP [28], and MGARD [1]. These lossy compressors provide parameters that allow users to finely control the information loss introduced by lossy compression. Unlike traditional lossy compressors such as JPEG [41] for images (in integers), SZ, ZFP, and MGARD are designed to compress floating-point data and can provide a strict error-controlling scheme based on the user's requirements. Generally, lossy compressors provide multiple compression modes, such as error-bounding mode and fixed-rate mode. Error-bounding mode requires users to set an error type, such as the point-wise absolute error bound and point-wise relative error bound, and an error bound level (e.g., 10^{-3}). The compressor ensures that the differences between the original data and the reconstructed data do not exceed the user-set error bound level. Fixed-rate mode means that users can set a target bit-rate (the number of bits to represent each data point), and the compressor guarantees the actual bit-rate of the compressed data to be lower than the user-set value.

B. Huffman Coding

Huffman Coding is a classic technique developed by David Huffman in 1952 for performing lossless compression [14]. It encodes a fixed-length value as a variable-length code. We call the fixed-length input value an *input symbol*, and we call the variable-length output value a *codeword*. In Huffman coding, space savings result from the fact that more frequently occurring symbols are represented by codewords with fewer bits, and vice versa for less frequently occurring symbols. Huffman codewords are also prefix-free; no one codeword can be a prefix of any other codeword.

C. CUDA Architecture

Thread: The thread is the basic programmable unit that allows programmers to use massive numbers of CUDA cores. CUDA threads are grouped at different levels including warp, block, and grid.

Warp: The warp is a basic-level scheduling unit in CUDA associated with SIMD (single-instruction multiple-data). Specifically, the threads in a warp achieve convergence when executing exactly the same instruction; otherwise, warp divergence happens. In the current CUDA architecture, the number of threads in a warp is 32, hence, it works as 32-way SIMT when converging. However, when diverging happens, diverged threads add extra overhead to the execution [44].

Block: Unlike the warp, the thread block (or simply block) is a less hardware-coupled description of thread organization, as it is explicitly seen in the kernel configuration when launching one. Threads in the same block can access the shared memory, a small pool of fast programmable cache. On one hand, shared memory is bound to active threads, which are completely scheduled by the GPU hardware; however, on the other hand, a grid of threads may exceed the hardware-supported number of active threads at a time. As a result, the data stored in the shared memory used by the previous batch of active threads may be invalid when the current or following batch of active threads are executing. Thus, we must carefully tune the shared memory size for different workloads to attain high performance.

Grid: A grid encompasses the entire set of blocks that are launched as part of a CUDA kernel. Usually, the grid of threads describes either the entire problem or a working set of the problem at hand. Moreover, all the blocks within a grid share a common configuration; each block within a grid contains the same amount of shared memory and the same number of threads per block.

D. Error-bounded Lossy Compression on GPU

SZ, ZFP, and MGARD were first developed for CPU architectures, and all started rolling out their GPU-based lossy compression recently. The SZ team, the ZFP team, and the MGARD team released their CUDA versions, called cuSZ [39], cuZFP [7], and cuMGARD [5], respectively. All the versions provide much higher throughputs for compression and decompression compared with their CPU versions [39, 19, 38]. Compared with cuSZ and cuMGARD, cuZFP provides slightly higher compression throughput, but it only supports fixed-rate mode [19], limiting its adoption in practice. Both cuSZ and cuMGARD use Huffman encoding to achieve high compression ratios and their decompression throughput is greatly limited by slow Huffman decoding on GPUs, but cuSZ has a much higher throughput than cuMGARD [38, 5]. Thus, in this work, we focus on optimizing Huffman decoding for cuSZ.

III. Analysis of Existing Huffman Decoders for Error-Bounded Lossy Compression

A. Coarse-Grained Versus Fine-Grained Parallelism

The current implementation of parallel Huffman decoding in cuSZ requires a number of fixed-size chunks containing thousands of codewords to be decoded sequentially by many threads [39]. Such a solution is called a coarse-grained solution, as there are fewer threads performing a relatively large amount of work. Although such a solution may provide good performance on a multi-core CPU, as multi-core CPUs tend to have either tens or hundreds of powerful, independent cores, GPUs have thousands of interdependent cores that work best when running together in lock-step. Since GPU cores are interdependent, and parallel Huffman decoding is not particularly amenable to lock-step parallelism, the apparent performance of a single thread is relatively weak compared to a CPU thread. Nevertheless, since GPUs have so many cores, we can improve performance by proposing a fine-grained solution—a solution that launches many threads that operate on fewer data elements.

It is possible to extract greater parallelism from cuSZ's existing coarse-grained Huffman decoder by decreasing the size of each chunk; however, since Huffman codes are variable length, very small chunks may not be able to be filled, leaving empty space in chunks that degrade the compression ratio. One avenue for increasing the parallelism in Huffman decoding is to determine a starting point in the bitstream for each thread; two connected strategies for doing this are described in the following subsections.

B. Self-Synchronization Based Huffman Decoding

Weißenberger and Schmidt proposed a parallel algorithm for Huffman decoding on the GPU using a property of Huffman codes called SELF-SYNCHRONIZATION [42]. Their technique is in turn based on an earlier CPU-based parallel Huffman decoder by Klein and Wiseman [22]. This algorithm is designed to work on pure Huffman codes; no modifications to the Huffman encoding step need to be done. It uses the self-synchronization property of Huffman codes to determine where in the bitstream each thread starts decoding, allowing for finer-grained parallelism than a chunk-based approach.

1) Self-Synchronization: The self-synchronization property of Huffman codes is the tendency for a Huffman decoder to correct itself even if a few bits of the input were skipped in error [9]. An example of self-synchronization is as follows: consider the bit pattern ``111000010111000'' with the Huffman codebook from [9], shown in Listing 1. A correct decoding of this pattern is ``(11)(10)(00)(010)(11)(10)(00)'', or ``CBADCBA''. However, if one bit is skipped in the input, then the pattern is decoded as ``(11)(00)(00)(10)(11)(10)(00)'', or ``**CAAB**CBA''. The first four characters' outputs are incorrect, but after decoding four erroneous characters (8 bits), the decoder starts decoding the correct characters, i.e., it self-synchronizes. There is a possibility that for a given Huffman codebook, there are codestreams that never self-synchronize, but Klein and Wiseman [22] demonstrated that for practical datasets, self-synchronization was achieved in less than 72 bits on average.

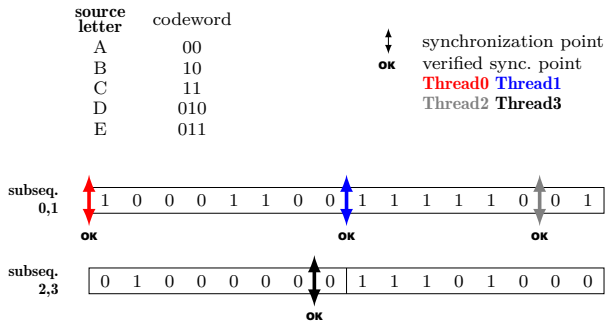

Fig. 1: Illustrating the final step of the self-synchronization phase of Weißenberger and Schmidt's decoder

```
symbol → codeword = {
    'A': '00', 'B': '10',
    'C': '11', 'D': '010', 'E': '011' }
```

Listing 1: Example self-synchronizing codebook from [9].

2) Fast Decoding via Self-Synchronization: Although self-synchronization is most often examined in the context of error correction, it can be used to determine starting locations for many-threaded Huffman decoders. This is the basic technique used in Weißenberger and Schmidt's decoder [42]. To determine where each thread starts, first, each thread is placed at evenly-spaced intervals throughout the Huffman bitstream. Then, each thread decodes but does not write, a certain number of bits in the Huffman bitstream, and stores the location where it stopped decoding, called a synchronization point. This decoding gives each thread an opportunity to self-synchronize; if this is the case, the stored location points to a valid codeword. Each thread's synchronization point is validated by the previous thread. If the previous thread 'meets' up with the current thread's synchronization point, then the synchronization point refers to a valid codeword; otherwise, the threads continue decoding until a valid synchronization point is found. For example, Figure 1 illustrates the conclusion of the self-synchronization phase of the algorithm, when each thread has had its synchronization point verified; if a thread starts decoding at one of these points, it will decode valid characters in the encoded string ``BACACCBDBAAEBBA''.

An overview of Weißenberger and Schmidt's algorithm is described as follows:

- Use self-synchronization to determine the synchronization points within each sequence;
- Use self-synchronization to adjust the synchronization points between sequences;
- Use a prefix sum to determine where each thread writes to in the output array;
- Have each thread decode data, starting at a synchronization point, and write it to the output array.

To understand the steps of the above algorithm, we define a SEQUENCE to be a chunk of the input data that a single CUDA thread block operates on. A subsequence is a subdivision of a sequence that a single CUDA thread works on. A subsequence in turn is divided into UNITS: unsigned 32-bit numbers that contain the individual codewords. The number of synchronization points within a given dataset is equal to the number of subsequences in the input data. Steps 1 and 2 ensure that all the synchronization points reference valid codewords. During this process, the number of valid codewords in each subsequence is recorded. These numbers are then prefix-summed in step 3 to determine the first output index for each subsequence. Finally, in step 4, each thread starts decoding at its synchronization point and outputs data starting at the index generated by the prefix sum. We refer readers to [42] for more details.

Limitation: Although the self-synchronization based Huffman decoding allows for finer-grained Huffman decoding, it contains some major performance bottlenecks. One major bottleneck is determining synchronization points; to do this, the algorithm needs to decode the input data multiple times depending on the particular dataset to be decoded.

C. Gap Arrays

To address self-synchronization's performance issue, Yamamoto *et al.*'s work proposed a new data structure called a gap array to eliminate this decoding bottleneck, in exchange for some extra encoding overhead [45]. Note that although the work also proposes an optimized encoding scheme, our main focus is its decoding scheme. Similar to Weißenberger and Schmidt's algorithm, Yamamoto *et al.*'s decoder divides its input data into sequences, subsequences, and units as defined above. However, instead of finding out where each thread starts decoding within a subsequence by determining synchronization points, this information is stored alongside the compressed data in a gap array. A GAP ARRAY is a byte array, with one byte per subsequence, that indicates to each thread how many bytes it must skip before accurate data can be decoded. For example, a gap array for the codewords in Figure 1 would be $[0, 0, -2, -1]$, as these are the offsets from the subsequence boundaries that each thread would have to keep track of in order to decode correctly. The gap array is used in combination with a technique called Single Kernel Soft Synchronization (SKSS) to determine output indices, decode the codewords, and write output symbols to memory. We refer readers to [45] for more details.

Limitation: Although the gap array removes the necessity of performing the self-synchronization phase and speeds up the decoding, gap arrays introduce other overheads. These overheads include the extra space required to store the gap array as well as extra work for the encoder. Nevertheless, the extra space required to store a gap array is minimal, as Yamamoto *et al.* has shown that the size of the gap array is less than 3% of the size of the data on their tested datasets of varying compression ratios [45]. However, the extra work the encoder must perform in generating the gap array means that the encoder and the decoder must be coupled. This reduces the flexibility of gap-array-based Huffman decoding, as it will not be able to decode Huffman codes generated by encoders not designed to create gap arrays. Nevertheless, since there are compelling reasons to use both self-synchronization and gap arrays in practice (will be discussed in detail in §V-C), we consider both solutions in our optimization work.

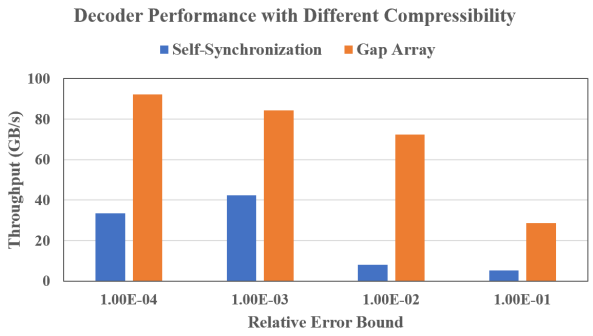

Fig. 2: Decoding performance versus error bounds on HACC dataset. Note that the larger the error bound, the larger the compression ratio.

D. Challenges of Using Existing Huffman Decoders

Both Weißenberger and Schmidt's and Yamamoto *et al.*'s works have been evaluated across a wide range of general-purpose datasets [42, 45]. However, in error-bounded lossy compression such as cuSZ, since input data are quantization codes, the resulting quantization codes are often highly compressible, especially in a well-predicted dataset. Our experiments show that both their works underperform on high-compressible datasets, as can be seen in Figure 2. Note that in general, as error bound increases, the resulting quantization codes become easier to compress. The figure illustrates a drop in the throughput of both decoders as data becomes more easily compressible. Thus, not only do we optimize both decoders in general, but we also specifically focus on techniques to optimize high compression-ratio cases commonly seen in scientific data reduction.

IV. Design Methodology

To allow efficient Huffman decoding of multi-byte input such as quantization codes in cuSZ, we perform a series of architectural optimizations on both Weißenberger and Schmidt's and Yamamoto *et al.*'s solutions. We start with Weißenberger and Schmidt's implementation as a baseline and adapt it to multi-byte input. We continue by examining the bottlenecks in the current algorithm: (1) the intra-sequence synchronization phase (i.e., Step 1), for self-synchronization based Huffman decoding; and (2) the decoding and writing phase (i.e., Step 4), for both self-synchronization and gap-array-based Huffman decoding. Details of the decoders can be found both in this subsection as well as in our online repository[1].

A. Optimized Self-Synchronization

We note that although the average behavior of self-synchronization is well-predictable, the amount of data each thread needs to decode to achieve self-synchronization can vary, as aforementioned. More severely, although each thread needs to decode only two subsequences on average to find and validate a synchronization point[2], up to 5% of threads decode greater than two subsequences, and individual threads can decode up to 125 subsequences on the test datasets in order to find and validate synchronization points. As a result, the local unpredictability of self-synchronization hinders GPU implementation because if one thread decodes more subsequences than other threads in the same CUDA warp, the other threads are held up until the longest-running thread finishes its job. This inefficiency is exacerbated by the fact that, in the self-synchronization phase, a CUDA block-wide thread barrier is required for correctness. Thus, the longest-running thread determines the running time of the entire block; and other threads within a block remain idle.

A potential solution to this issue is to perform load balancing to ensure that threads in a block are not idle; however, due to the overhead of load balancing and the relatively low occurrence of exceptionally long-running threads, we do not pursue the load balancing approach. Instead, we do optimize the intra-sequence self-synchronization kernel to minimize the impact of long-running threads and conform more closely to the CUDA architecture. Specifically, in the original code for self-synchronization based Huffman decoding, after each thread discovers and validates a synchronization point, it busy-waits not only until the longest-running thread in the thread block finishes but also until the maximum possible number of subsequences that a thread may decode until self-synchronization is reached (e.g., 128 subsequences in this case). To allow thread blocks to exit as early as possible, we record each thread's "finished" status in a Boolean variable. By using the CUDA warp primitive `__all_sync`, we can determine whether all the threads have finished finding their synchronization points; and if so, we will terminate the kernel immediately, freeing up warps within a CUDA Streaming Multiprocessor to execute other blocks in the kernel. This optimized intra-sequence self-synchronization runs, on average, 11% faster than the baseline code, and these benefits are concentrated in lower compression ratio datasets, in which this phase is a more significant bottleneck than in high compression-ratio datasets.

B. Optimized Decoding and Writing of Codewords

In both presented decoders, threads decode and write codewords directly to the GPU's global memory. There is a stride between different threads' output indices; this stride reflects the number of codewords that can be found between the two threads' input locations. This counters one of the characteristics of CUDA's memory architecture: coalesced memory loads and stores. Specifically, a coalesced memory access is when sequential global memory transactions are combined with each other to reduce the number of memory transactions actually performed. For example, in CUDA, a 32-thread warp writing 32-bit values to sequential locations in memory have its write requests processed as a single 128-byte transaction. Note that inefficient memory access patterns result in many more transactions being made, which decreases the throughput of the global memory.

For high compression-ratio datasets, this memory inefficiency is even worse, because not only are the gaps between

[1] https://github.com/codyjrivera/ipdps22-opthuffdec.

[2] Note that one subsequence containing four 32-bit units results in 128 bits decoded per subsequence.

adjacent threads' output indices large, but also the number of values written to global memory and hence the number of transactions are large. This is a significant factor in the dramatic drop in performance with high compression-ratio datasets shown in Figure 2. Although Yamamoto *et al.*'s work has each thread write multiple symbols at a time to global memory [45], performance still eventually degrades at high ratios, as can be seen in the same figure.

Algorithm 1: Decoding and writing using a shared memory buffer.

• DecodeWrite — decode and write using shared memory

```
1  sharedBuffer[n]                          ▷ The shared memory buffer of size n
2  si <- outIndex[blockIdx.x · blockDim.x]
3  ei <- outIndex[(blockIdx.x + 1) · blockDim.x]
4  gid <- threadIdx.x + blockDim.x · threadIdx.x
5  tempEnd <- ei
6  while si < ei do
7     start <- outIndex[gid] - si, end <- outIndex[gid + 1]
8     if si ≤ start and end ≤ si + n  then
9        outArray[start ... end) <- DECODE(inArray, startPoint[gid])
                                  ▷ If symbols can fit into the buffer, decode them
10    else if start < si + n and end ≥ si + n  then
11       tempEnd <- outIndex[gid]
          ▷ Executed by one thread if buffer is not large enough; results in another iteration
12    end if
13    outArray[si ... tempEnd) = sharedBuffer[0 ... tempEnd - si)
                          ▷ This write is performed cooperatively by threads in the block
14    si <- tempEnd
15 end while
```

To solve this issue, we propose to first decode the input data into a thread block-local buffer, and then write it out sequentially to global memory to attain coalesced and hence efficient writes. For the thread block-local buffer, we use shared memory. Specifically, first, given the global output index of each thread, the kernel will compute the local index where each thread will put the decoded symbol within the shared memory buffer. Then, the kernel will decode and have each decoder write its data into the shared memory. Finally, all threads in the thread block cooperatively write the data in the shared memory to the global memory output array. Note that the codebook that is used for decoding is kept in global memory; since this codebook is shared across all thread blocks, it is kept in cache, so we do not need to consider keeping a codebook in shared memory and can dedicate the shared memory for the decoding buffer. Note that if the shared memory is not large enough to store all the data that threads inside the block will decode, that the shared memory will be filled up with the initial chunk of decoded data by the initial group of threads, that data will be written, and then the rest of the threads will fill up the shared memory with the rest of the decoded data. More details about this procedure can be found in Algorithm 1.

C. Shared Memory Tuning for Decoding and Writing

The method proposed for decoding the codewords and writing the decoded symbols back to memory requires a certain amount of shared memory to be allocated to the appropriate kernel. Choosing this amount of shared memory can significantly impact the performance of this phase of the decoder, because (1) allocating too little shared memory can reduce parallelism, and (2) allocating too much shared memory can reduce occupancy. This can be illustrated in Figure 3, evaluated

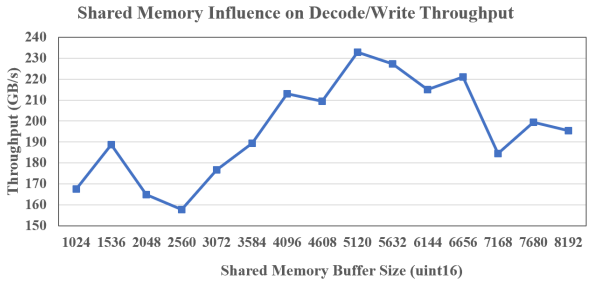

Fig. 3: Throughput of the decoding and writing phase with different shared memory sizes on the quantization codes generated by cuSZ on HACC dataset with relative error bound 10^{-3}.

on our HACC dataset described in § V-A, with an error bound of 10^{-3}. Note that the difference between the lowest and highest throughput is around 32% (i.e., 233 GB/s vs. 158 GB/s).

A first approximation to the amount of shared memory allocated would be to allocate an amount of shared memory proportional to the compression ratio: in this case, the compression ratio is 3.86, and the corresponding buffer size, rounding up 256 spots, would be 4096 16-bit integers. However, the optimal size of the shared memory buffer, in this case, is 5120. Therefore, an improved strategy of allocating shared memory is needed.

To this end, we propose a strategy that more effectively reflects the characteristics of actual data, where some portions of the data are highly-compressible but others are not so highly-compressible. We launch separate kernels for decompressing input sequences with different compression ratios. We determine which sequence is to be decoded by each kernel using an online selection process. That way, each sequence is decoded by a kernel launched with an optimal amount of shared memory.

Algorithm 2: Our proposed shared memory optimization that partitions input sequences among kernels launched with different amounts of shared memory. Lines implemented by host code are in blue, while lines implemented by CUDA kernels are in red.

• ShmemOptDecodeWrite — decode and write with optimal shared memory use

```
1  compRatio[n]              ▷ The precomputed compression ratios of the n sequences
2  for all i in [0 ...n) concurrently do
3     compClass[i] <- CLASSIFYCR(compRatio[i])
4  end for      ▷ Classifies all the compression ratios into one of the T_high + 1 groups
5  compClassFreq <- PARHISTOGRAM(compClass)
                                  ▷ Finds out how many sequences fall into each group
6  compIndex <- [0, 1, ..., n - 1]
7  PARKEYVALUESORT(compClass, compIndex)
          ▷ Allows decoding kernels to access sequences in its compression ratio group
8  compClassStart[0] <- 0
9  for i in [1 ...T_high + 1) do
10    compClassStart[i] <- compClassStart[i - 1] + compClassFreq[i - 1]
11 end for        ▷ Determines where in compIndex each compression ratio group starts
12 for all i in [0 ...T_high + 1) asynchronously do
13    DECODEWRITE(optShmem(i), compClassStart[i], compClassFreq[i])
14 end for ▷ Launches decoding kernels for each of the compression ratio groups. Each
   kernel gets an amount of shared memory optimized for its compression ratio group.
```

We now give more details about this strategy, which are illustrated in Algorithm 2. First, this strategy requires each sequence's compression ratio as input. This is taken from an earlier phase of each algorithm: the self-synchronization phase for Weißenberger and Schmidt's algorithm and redundant

decoding in Yamamoto *et al.*'s algorithm required to determine where each thread writes its data. After this is done, (1) the shared memory optimization starts by classifying each sequence's compression ratio into $T_{high} + 1$ groups, where T_{high} is an architecture-specific threshold, on lines 2-4. This classification is then stored inside an on-device array. T_{high} of these groups are the compression ratios in the intervals $(0, 1], (1, 2], ..., (T_{high} - 1, T_{high}]$, and the $T_{high} + 1$-th group encompasses compression ratios in the interval $(T_{high}, 16]$. Thus, at most $T_{high} + 1$ different kernels with varying amounts of shared memory are launched. (2) The array is then histogrammed on the GPU, in order to see how many sequences fall into each compression ratio group. The algorithm used is the same variation of Gómez-Luna *et al.* [11] that is used in cuSZ. (3) Once the classification array is histogrammed, on line 5, it is then sorted on the GPU as part of a key-value sort, with the classification being used as the key and a sequential list of indices being used as the values. The resulting values will be the primary means by which sequences in a particular compression ratio group are accessed and decoded. The algorithm used is the `DeviceRadixSort` routine in CUB [30]. Furthermore, since T_{high} is fairly small, sorting $T_{high} + 1$ groups is fast using CUB (will be proved in Table II). (4) After being transferred back to the CPU, the histogram is then used to generate a prefix sum that indicates where in the list of indices the sequences belonging to that compression ratio group begin. (5) Finally, up to $T_{high} + 1$ kernels are launched with an amount of shared memory (mostly) proportional to their corresponding compression ratio group's upper bound. For example, sequences in the $(3, 4]$ compression ratio group would be decoded by a kernel with a shared memory buffer of length 4096. Each kernel is launched on a separate CUDA stream in order to allow the CUDA driver maximum flexibility in scheduling and running the $T_{high} + 1$ kernels. These kernels then finish decoding the data and write their data to the output array.

	HACC	EXAALT	CESM	Nyx	Hurr.	QMC.	RTM	GAMESS
size in mebibyte	1071.8	951.7	642.7	512.0	381.5	601.52	180.7	306.19
tuned GB/s	217.0	213.4	188.2	143.3	154.2	194.7	137.7	146.9
best brute-force GB/s	235.5	212.2	175.8	147.6	152.5	214.8	147.0	146.6
shared memory buffer size	5120	3584	5632	5632	5632	3072	5632	3584
% diff. from tuned	8.5%	-0.5%	-6.5%	3.0%	1.1%	10.3%	6.7%	-0.2%
worst brute-force GB/s	157.2	159.5	117.8	92.1	92.6	131.5	88.6	88.9
shared memory buffer size	3072	7680	1024	1024	1024	2048	1024	1024
% diff. from tuned	27.5%	25.3%	37.4%	35.7%	40.0%	32.4%	35.6%	39.5%
tuning speed GB/s	2172.3	2126.8	1471.3	1087.0	745.4	1288.2	428.0	625.8
tuned w/tuning overhead GB/s	197.3	194.0	166.8	126.6	127.8	169.1	104.2	119.0
% diff. from best case	19.3%	9.4%	5.4%	16.6%	19.3%	27.0%	41.0%	23.2%
% diff. from worst case	20.3%	17.8%	29.4%	27.3%	27.5%	22.2%	14.9%	25.3%

TABLE I: Comparison between our shared memory optimization and brute-force search for decoding and writing. The input quantization codes are generated by cuSZ with a relative error bound of 10^{-3}. Negative percentages denote cases where our optimization outperformed the fastest brute-force case.

Table I examines this shared memory optimization by comparing the decoding throughputs achieved by our optimization and by a brute-force search (in increments of 512 bytes from 1024 to 8192) for the optimal amount of shared memory to launch in a single grid (the test datasets will be described in § V). According to the table, the throughputs of the decoding using this shared memory optimization on all of the datasets are within 10% of the maximum throughput in the brute-force search. Furthermore, some of the datasets tested performed better under the optimization than the maximum throughput achieved in the brute-force search; this is because different sections of the dataset have different compression ratios[3]. Additionally note the percent difference between the worst possible throughput and the optimized throughput; if shared memory is not used appropriately, the decoder may incur a performance penalty of up to 40%.

However, when one considers the overhead of the tuning mechanism itself, as is shown on the last rows of Table I, this overhead ranges from 10.0%∼32.2%. Note that this overhead is smaller on large datasets and vice versa. This is because tuning, in practice, takes approximately 220 microseconds on all datasets. Nevertheless, even with this tuning overhead, the decoder avoids a performance penalty of 23.1% on average. However, note that on the 3D reverse time migration (RTM) dataset—the smallest dataset—the performance penalty avoided is only 14.9%, while the performance loss compared to the best case achieved by brute force is 41.0%. This is because the relatively constant runtime of tuning impacts smaller datasets more. Furthermore, several entries in Table I suggest that one can just use an architecture-specific shared memory buffer size, like 5632 on the V100. However, on the EXAALT dataset, if a buffer size of 5632 is used, then decoding happens at 168.9 GB/s, which is 14.8% slower than the tuned decoder, even when accounting for tuning overhead. Therefore, it is worth using our proposed tuning approach to find optimal buffer sizes for different datasets on different architectures.

Note that in order to prevent a large reduction in occupancy caused by allocating too much shared memory, there is a threshold up to which we can allocate an amount of shared memory proportional to the compression ratio: this threshold is called T_{high}, as aforementioned. To attain T_{high} for a particular GPU architecture, calculate the amount of shared memory required to attain at least 25% occupancy, and divide that amount by 2048 to obtain T_{high}. For example, on the Nvidia Tesla V100, shared memory usage must be under 16384 bytes to attain that level of occupancy, so the corresponding value of T_{high} is 8. If the compression ratio exceeds T_{high}, our experiments have demonstrated that 3584 symbols are an optimal size for the buffer in most situations on the V100 GPU.

V. Performance Evaluation

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

A. Experiment Setup

1) Evaluation Platforms: We conduct our experimental evaluation on the Bridges-2 supercomputer [2] at Pittsburgh Supercomputing Center (PSC), of which each GPU node is equipped with two Intel Xeon Gold 6248 CPUs and eight 32 GB NVIDIA Tesla V100 GPUs.

[3]Note that in order to achieve maximum throughput, different sections must be decoded with different amounts of shared memory.

techniques	HACC	EXAALT	CESM	Nyx	Hurr.	QMC.	RTM	GAMESS	HACC	EXAALT	CESM	Nyx	Hurr.	QMC.	RTM	GAMESS	HACC	EXAALT	CESM	Nyx	Hurr.	QMC.	RTM	GAMESS
	ORIGINAL SELF-SYNC., GB/S								OPTIMIZED SELF-SYNC., GB/S								OPTIMIZED GAP ARRAY, GB/S							
size in mebibyte	1071.8	951.7	642.7	512.0	381.5	1071.8	601.5	180.7	306.2	951.7	642.7	512.0	381.5	601.5	180.7	306.2	1071.8	951.7	642.7	512.0	381.5	601.5	180.7	306.2
compr. ratio	3.18	2.40	9.60	15.99	9.78	2.46	8.41	12.10	3.18	2.40	9.60	15.99	9.92	2.45	8.62	12.45	3.00	2.26	9.04	15.05	9.33	2.31	8.12	11.71
intra-seq. sync.	155.4	133.8	264.4	469.9	295.3	112.3	245.9	363.4	208.0	169.2	345.9	430.8	253.1	144.0	234.2	352.0	-	-	-	-	-	-	-	-
inter-seq. sync.	2295.3	1844.1	4240.8	6097.6	3436.6	1770.4	2086.2	3460.8	2145.6	1783.2	4813.2	4761.9	3443.0	1646.4	1838.4	3036.6	-	-	-	-	-	-	-	-
get output idx.	573.8	435.1	1516.0	2293.6	1487.7	439.3	1155.0	1765.1	569.0	432.3	1494.4	2189.1	1405.8	433.2	1060.6	1614.5	268.6	239.7	495.1	694.4	384.5	225.6	355.1	515.7
tune shared mem.	-	-	-	-	-	-	-	-	2172.3	2126.8	1471.3	1087.0	745.4	1180.0	406.5	683.0	2111.0	2128.8	1473.3	1234.6	741.8	1201.3	442.6	734.7
decode and write	60.3	70.8	7.0	5.6	7.0	59.8	10.2	6.0	219.7	211.4	186.5	143.9	153.7	194.1	138.8	161.3	214.0	210.1	186.1	168.9	153.0	195.4	148.5	173.0
overall, decode	39.7	40.9	6.8	5.5	6.8	35.1	9.6	5.9	83.0	71.5	101.9	92.1	78.1	63.1	64.8	87.3	112.8	106.4	123.9	122.4	95.4	96.3	84.7	110.1
speedup	1.50×	1.57×	0.27×	0.09×	0.27×	1.48×	0.33×	0.16×	3.14×	2.74×	4.05×	1.55×	3.15×	2.66×	2.25×	2.36×	4.27×	4.08×	4.92×	2.07×	3.85×	4.07×	2.94×	2.98×

TABLE II: A comprehensive evaluation of all proposed decoding solutions on V100: Huffman decoders using cuSZ quantization codes generated with a relative error bound of 10^{-3} on V100. GB/s is computed relative to the size of the generated quantization codes, i.e, half the dataset size.

2) Comparison Baselines: We compare our optimized Huffman decoding with multiple baselines. Specifically, we compare our solution with (1) the original self-synchronization-based Huffman decoder [42], (2) the original gap-array-based Huffman decoder [45], and (3) cuSZ's naïve Huffman decoder [39]. Note that as the original gap-array-based Huffman decoder [15] cannot be adapted to multi-byte inputs due to a bug, we estimate its performance by trimming each multi-byte quantization code to a single byte, considering most quantization codes are concentrated in the middle.

datasets	datum size **dimensions**	#fields **examples(s)**
cosmology **HACC**	1,071.75 MB 280,953,867	6 in total xx, vx
molecular dynamics **EXAALT**	951.73 MB 2338×106711	6 in total dataset2.x
climate **CESM-ATM**	642.70 MB 26×1800×3,600	33 in total CLDICE, RELHUM
cosmology **Nyx**	512 MB 512×512×512	6 in total baryon_density
climate **Hurricane**	381.47 MB 4×100×500×500	13 in total CLDICE, QRAIN
quantum circuits **QMCPack**	601.52 MB 115×69×69×288	2 in total einspline, einspline.pre
petroleum exploration **RTM**	180.73 MB 449×449×235	1 in total (3600 snapshots) snapshot-1000
quantum chemistry **GAMESS**	306.19 MB 80,265,168	3 in total dddd, ffdd, ffff

TABLE III: Real-world `float`-type datasets used in the evaluation.

3) Test Datasets: We conduct our evaluation and comparison based on eight typical 1D∼4D real-world HPC simulation datasets, including six from Scientific Data Reduction Benchmarks [34]: 1D HACC cosmology simulation [12], 2D LAMMPS (part of the EXAALT ECP project) molecular dynamics simulation [24], 3D CESM-ATM climate simulation [6], 3D Nyx cosmology simulation [31], 4D Hurricane ISABEL simulation [16], and 4D QMCPack quantum simulation [32]. They have been widely used in much prior work [37, 26, 27, 46, 38, 40, 39, 20, 4] and are good representatives of production-level simulation datasets. Additionally, we also evaluate two datasets that highlight our decoders' potential to be used as in-memory compressors as discussed in §I, including 3D RTM simulation data for petroleum exploration [17] and 1D GAMESS data for quantum chemistry simulation [10]. Each dataset includes multiple snapshots and diverse fields. Table III presents the test datasets in detail. The data sizes per snapshot are 1.1 GB, 952 MB, 643 MB, 512 MB, 381 MB, 602 MB, 181 MB, and 306 MB for the above eight datasets, respectively. Note that the datasets tested are over 100 MB in size. This is because larger snapshots are more likely to be found in scientific applications, especially in in-memory applications where the datasets to be compressed often take up a significant portion of the total available memory. However, as we have verified by truncating and decoding the HACC dataset, datasets as small as 10 MB can exhibit speedups over the baseline cuSZ decoder. In addition, the datasets tested are all single-precision data, because the current cuSZ only works with single-precision data. However, since our underlying optimizations work on Huffman decoding of multibyte symbols, our technique applies to double-precision data as well.

B. Experimental Results

1) Huffman Decoding: Table IV illustrates the compression ratios of our optimized decoders and the baselines on the test datasets. We note that the differences between the compression ratios of different methods are up to around 10%. Thus, compression ratio is not the primary factor for choosing the most appropriate Huffman decoding approach; by comparison, throughput is more important. Note that, although the "original gap-array" row in Table IV refers to an 8-bit decoder, we double the compression ratio, so it can be used as a baseline for comparison with 16-bit decoders.

	HACC	EXAALT	CESM	Nyx	Hurr.	QMC.	RTM	GAMESS
size in mebibyte	1071.8	951.7	642.7	512.0	381.5	601.5	180.7	306.2
baseline cuSZ	3.20	2.40	9.06	15.64	9.78	2.46	8.41	12.10
	1.000×	1.000×	1.000×	1.000×	1.000×	1.000×	1.000×	1.000×
ori. self-sync	3.18	2.40	9.60	15.99	9.92	2.45	8.62	12.45
	0.996×	1.000×	1.059×	1.022×	1.014×	0.998×	1.026×	1.029×
opt. self-sync	3.18	2.40	9.60	15.99	9.92	2.45	8.62	12.45
	0.996×	1.000×	1.059×	1.022×	1.014×	0.998×	1.026×	1.029×
ori. gap-array*	3.11	2.60	9.42	15.51	9.68	2.41	8.43	12.10
	0.972×	1.079×	1.040×	0.992×	0.990×	0.982×	1.002×	1.000×
opt. gap-array	3.00	2.26	9.04	15.05	9.33	2.31	8.12	11.71
	0.938×	0.941×	0.997×	0.962×	0.954×	0.939×	0.965×	0.968×

TABLE IV: Compression ratio of eight evaluated methods. The original gap-array-based method is of 8-bit symbols, so their compression ratios are doubled to provide a fair comparison.

On the other hand, Table V shows the throughput of each decoding method in GB/s. The average speedup of our optimized self-synchronization solution compared to the baseline (in this case cuSZ's decoder) is 2.74×, and the average speedup of our optimized gap-array solution is 3.64×. Note that that the speedup over the original implementations of self-synchronization and gap-array solutions is more notable on high compression-ratio datasets. This is because the original implementations do not write out symbols to global memory in an efficient manner which is in turn exacerbated by the fact that high compression-ratio datasets have more symbols to be written out to memory. This underscores the importance of the optimizations for efficient memory access and use of shared memory introduced in §IV-B, especially when considering quantization codes generated by effective prediction methods. Note further that the original gap-array solution, although its GB/s numbers are computed relative to 8-bit quantization

codes, still achieves performance numbers that are greater than our optimized self-synchronization solution. Nevertheless, in addition to the practical reasons detailed above, that solution also exhibits the same performance issues on high compression-ratio datasets described earlier.

	HACC	EXAALT	CESM	Nyx	Hurr.	QMC.	RTM	GAMESS
size in mebibyte	1071.8	951.7	642.7	512.0	381.5	601.5	180.7	306.2
baseline cuSZ	26.4 1.00×	26.1 1.00×	25.2 1.00×	59.2 1.00×	24.8 1.00×	23.7 1.00×	28.8 1.00×	37.0 1.00×
ori. self-sync	39.7 1.50×	40.9 1.57×	6.8 0.27×	5.5 0.09×	6.8 0.27×	35.1 1.48×	9.6 0.33×	5.9 0.16×
opt. self-sync	83.0 3.14×	71.5 2.74×	101.9 4.05×	92.1 1.55×	78.1 3.15×	63.1 2.66×	64.8 2.25×	87.3 2.36×
ori. gap-array 8-bit	84.4 3.20×	87.7 3.36×	30.0 1.19×	17.5 0.30×	37.3 1.50×	87.2 3.68×	31.9 1.11×	25.9 0.70×
opt. gap-array	112.8 4.27×	106.4 4.08×	123.9 4.92×	122.4 2.07×	95.4 3.85×	96.3 4.07×	84.7 2.94×	110.1 2.98×

TABLE V: Decoding throughputs of eight evaluated methods.

Table II illustrates more details of the original self-synchronization solution as well as our optimized self-synchronization and gap array solutions, breaking down the algorithms into multiple phases. The table shows the impact of our architectural optimizations on our optimized Huffman decoders in break down. As for the architectural optimizations for the self-synchronization phase, we obtain average speedups of 11% for the most expensive self-synchronization phase—intra-sequence self-synchronization. We can see that significant speedups of up to 34% are shown on lower compression ratio datasets, where this phase is a more significant bottleneck in decoding. The decoding and writing phase of both our optimized solutions, which both use our customized decoder, perform on average 7.1× faster than the baseline decoder's Huffman decoding phase and 15.6× faster than the original self-synchronization based Huffman decoding and writing phase. Nevertheless, our customized Huffman decoder does decode at a somewhat reduced bandwidth at high compression ratios, but this is compensated by performance gains elsewhere while decoding a high compression-ratio dataset.

We note that our optimized decoder achieves the lowest speedup relative to the baseline (1.55× and 2.07×) on the Nyx-quant dataset. This can be explained by examining both the dataset and the baseline cuSZ decoder: (1) The Nyx dataset is extremely high-compressible, and the encoded Nyx dataset consists mostly of codewords of length one. Since cuSZ's decoder works one bit at a time, it is able to decode more codewords. (2) Since fewer threads run on cuSZ's coarse-grained decoder, it does not encounter the same issues with high compression-ratio input that other finer-grained decoders have. As a result, the baseline cuSZ decoder has a relatively high performance on the Nyx dataset compared to the other datasets.

2) cuSZ Decompression: Figure 4 demonstrates the impact of our optimized decoders on the overall performance of cuSZ's decoder, by comparing the baseline decoder and our two optimized solutions. On average, substituting the baseline decoder with our optimized decoders resulted in 2.08× faster decompression using self-synchronization and 2.43× faster decompression using gap arrays. Note that in this scenario, we calculate GB/s with regard to the size of the scientific dataset itself rather than just the quantization codes. The reason

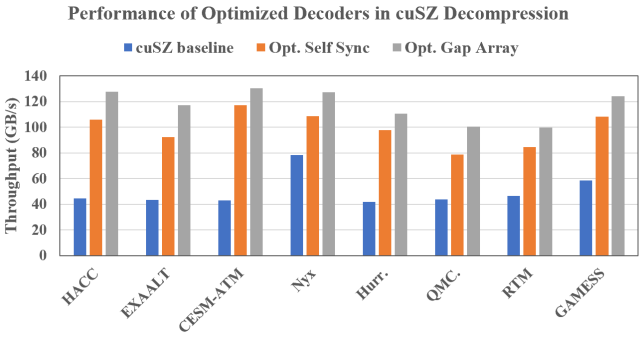

Fig. 4: Performance comparison between our optimized decompression and the cuSZ baseline on V100 (with relative error bound 10^{-3}). GB/s is computed relative to the size of the entire dataset.

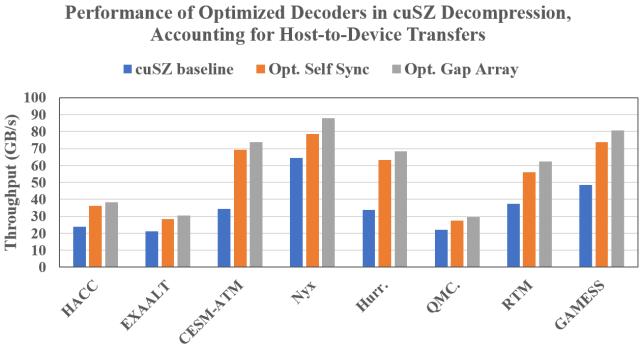

Fig. 5: Performance comparison between our optimized decompression and the cuSZ baseline on V100 (with relative error bound 10^{-3}), taking into account host-to-device memory transfers of compressed data. GB/s is computed relative to the size of the entire dataset.

that such a significant speedup can be attained is that cuSZ spends a substantial amount of time doing Huffman decoding; in the HACC dataset, cuSZ spent over 83% of the overall decompression time doing Huffman decoding. As a result, with our optimized decoders, the optimized cuSZ can decode at speeds of over 100 GB/s on the V100 on most of the test cases.

Additionally, in many GPU applications, compressed data is retained on CPU memory, which is a larger resource than GPU memory. When data is needed for processing on the GPU, before decompression, compressed data must be transferred from CPU memory to GPU memory. Thus, in Figure 5, we incorporate host-to-device "memcpy" into our evaluation. In this case, our optimized decompression performed, on average, 1.53× faster for self-synchronization and 1.65× faster for gap arrays over the cuSZ baseline. These speedups are lower than the results shown in Figure 4 as data transfers are a bottleneck due to a relatively slow bandwidth between the GPU and the CPU. Further note that the datasets with a relatively high throughput are those with a high compression ratio; this is because there is less actual data being transferred, so the compressed data transfer is relatively fast for those datasets.

C. Use-case of Our Two Decoders

In this paper, we introduced two algorithms from the literature for fast parallel Huffman decoding and implemented deep optimizations for these two algorithms. Although both

approaches are designed for fine-grained parallel Huffman decoding, and both approaches benefit from our architectural optimizations with regard to shared memory and decoding, both self-synchronization and gap array based parallel Huffman decoding are more suitable in some circumstances than others. Specifically, on one hand, if raw decoding performance is essential, gap array based Huffman decoding will inherently be faster than the self-synchronization based approach due to the costly and relatively unpredictable nature of finding synchronization points (particularly on GPUs). However, on the other hand, to obtain this raw decoding performance, applications must compute and store a gap array, which adds storage overhead as well as overhead to the encoder. Even in situations where these added costs are relatively insignificant, the encoder and the decoder must be coupled, meaning the encoder needs to be re-engineered. Thus, self-synchronization based Huffman decoding is more suited towards applications where encoded input data comes from different decoupled sources (e.g., where data is collected and compressed on a remote sensor).

VI. Related Work

In addition to works focusing on parallel Huffman decoding that have been refered to extensively throughout the paper (namely, Weißenberger and Schmidt's work [42], Yamamoto *et al.*'s work [45], and to a lesser extent Klein and Wiseman's work [22]), Johnston and McCreath additionally proposed an algorithm for massively parallel Huffman decoding [21]. Their algorithm proposes to deal with the problem of decoding variable length codes by starting decoding from every location in the bit sequence, eventually decoding the bit sequence correctly. Taking advantage of the GPU manycore architectures, the algorithm performs slightly faster on the GPU than a single-CPU-core Huffman decoder. This approach is not well-suited for our purposes, as their approach results in large amounts of computation for only marginal gains over CPU-based decoders. Thus, in this work we only consider the other two algorithms.

Many works have been done that focus on optimizing parallel Huffman-type encoding. For example, Lal *et al.* proposed a Huffman-based entropy encoding system (E^2MC) for GPUs [23]. More recently, Tian *et al.* proposed a fast parallel Huffman codebook construction algorithm and a parallel Huffman encoder for modern GPU architectures [40]. Since much work has already been focused on optimizing Huffman encoding, we do not presently consider optimizing encoding in our work.

VII. Conclusion

In this work, we comprehensively analyze two state-of-the-art Huffman decoding algorithms for error-bounded lossy compression of scientific data and propose a deep architectural optimization for both algorithms. We also propose an efficient online approach to tune the shared memory to decode different parts of the data based on the data characteristics. We then adapt our optimized decoders to multi-byte data and integrate it into cuSZ. Our evaluation on eight representative scientific datasets shows that our solution can improve cuSZ's Huffman decoding throughput by 3.64× on average and cuSZ's overall decoding throughput by 2.43× on average. In the future, we plan to optimize and evaluate our Huffman decoder for generic datasets such as text data on Nvidia A100 GPU.

Acknowledgments

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science and Office of Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357. This work was also supported by the National Science Foundation under Grants OAC-2003709, OAC-2034169, OAC-2042084, OAC-2104023, and OAC-2104023.

References

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

[2] Shawn T Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A Nystrom. "Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research". In: *Practice and Experience in Advanced Research Computing*. 2021, pp. 1–4.

[3] Martin Burtscher and Paruj Ratanaworabhan. "FPC: A high-speed compressor for double-precision floating-point data". In: *IEEE Transactions on Computers* 58.1 (2008), pp. 18–31.

[4] 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. "Use cases of lossy compression for floating-point data in scientific data sets". In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1201–1220.

[5] Jieyang Chen, Lipeng Wan, Xin Liang, Ben Whitney, Qing Liu, David Pugmire, Nicholas Thompson, Jong Youl Choi, Matthew Wolf, Todd Munson, et al. "Accelerating multigrid-based hierarchical scientific data refactoring on gpus". In: *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2021, pp. 859–868.

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

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

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

[9] T. Ferguson and J. Rabinowitz. "Self-synchronizing Huffman codes (Corresp.)" In: *IEEE Transactions on Information Theory* 30.4 (1984), pp. 687–693.

[10] Ali Murat Gok, Sheng Di, Alexeev Yuri, Dingwen Tao, Vladimir Mironov, Xin Liang, and Franck Cappello. "PaSTRI: A novel data compression algorithm for two-electron integrals in quantum chemistry". In: *IEEE International Conference on Cluster Computing*. IEEE, 2018, pp. 1–11.

[11] Leonardo A Bautista Gomez and Franck Cappello. "Improving floating point compression through binary masks". In: *IEEE International Conference on Big Data*. 2013, pp. 326–331.

[12] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Vishwanath, et al. "HACC: Extreme scaling and performance across diverse architectures". In: *Communications of the ACM* 60.1 (2016), pp. 97–104.

[13] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Morozov, et al. "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures". In: *New Astronomy* 42 (2016), pp. 49–65.

[14] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101.

[15] Huffman Coding with Gap Arrays. https://github.com/daisuke-takafuji/Huffman_coding_Gap_arrays.

[16] Hurricane ISABEL Simulation Data. http://vis.computer.org/vis2004contest/data.html.

[17] Sian Jin, Sheng Di, Suren Byna, Dingwen Tao, and Franck Cappello. "Improving Prediction-Based Lossy Compression Dramatically Via Ratio-Quality Modeling". In: *arXiv preprint arXiv:2111.09815* (2021).

[18] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. "DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression". In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 2019, pp. 159–170.

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

[20] Sian Jin, Jesus Pulido, Pascal Grosset, Jiannan Tian, Dingwen Tao, and James Ahrens. "Adaptive configuration of in situ lossy compression for cosmology simulations via fine-grained rate-quality modeling". In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 2020, pp. 45–56.

[21] Beau Johnston and Eric McCreath. "Parallel Huffman Decoding: Presenting a Fast and Scalable Algorithm for Increasingly Multicore Devices". In: *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. 2017, pp. 949–958.

[22] S. T. Klein and Y. Wiseman. "Parallel Huffman Decoding with Applications to JPEG Files". In: *The Computer Journal* 46.5 (2003), pp. 487–497.

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

[24] Large-scale atomic/molecular massively parallel simulator. https://lammps.sandia.gov/.

[25] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. "An efficient transformation scheme for lossy data compression with point-wise relative error bound". In: *IEEE International Conference on Cluster Computing*. IEEE. 2018, pp. 179–189.

[26] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. "Error-controlled lossy compression optimized for high compression ratios of scientific datasets". In: *2018 IEEE International Conference on Big Data*. IEEE, 2018, pp. 438–447.

[27] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Bogdan Nicolae, Zizhong Chen, and Franck Cappello. "Improving Performance of Data Dumping with Lossy Compression for Scientific Simulation". In: *IEEE International Conference on Cluster Computing*. IEEE, 2019, pp. 1–11.

[28] Peter Lindstrom. "Fixed-rate compressed floating-point arrays". In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 2674–2683.

[29] Peter Lindstrom and Martin Isenburg. "Fast and efficient compression of floating-point data". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 1245–1250.

[30] *NVIDIA/cub: Cooperative primitives for CUDA C++*. https://github.com/NVIDIA/cub.

[31] NYX. https://amrex-astro.github.io/Nyx/.

[32] QMCPACK. http://vis.computer.org/vis2004contest/data.html.

[33] Habib Salman. *Marching to Exascale: Extreme-Scale Cosmological Simulations with HACC on Summit*. https://www.olcf.ornl.gov/wp-content/uploads/2018/10/habib_2019OLCFUserMeeting.pdf. 2019.

[34] Scientific Data Reduction Benchmarks. https://sdrbench.github.io/.

[35] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. "Data compression for the exascale computing era-survey". In: *Supercomputing Frontiers and Innovations* 1.2 (2014), pp. 76–88.

[36] Summit supercomputer. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

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

[38] Jiannan Tian, Sheng Di, Xiaodong Yu, Cody Rivera, Kai Zhao, Sian Jin, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. "Optimizing Error-Bounded Lossy Compression for Scientific Data on GPUs". In: *IEEE International Conference on Cluster Computing*. 2021.

[39] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, et al. "cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data". In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2020, pp. 3–15.

[40] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. "Revisiting huffman coding: Toward extreme performance on modern gpu architectures". In: *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2021, pp. 881–891.

[41] Gregory K Wallace. "The JPEG still picture compression standard". In: *IEEE Transactions on Consumer Electronics* 38.1 (1992), pp. xviii–xxxiv.

[42] André Weißenberger and Bertil Schmidt. "Massively Parallel Huffman Decoding on GPUs". In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

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

[44] Ping Xiang, Yi Yang, and Huiyang Zhou. "Warp-level divergence in GPUs: Characterization, impact, and mitigation". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture*. IEEE, 2014, pp. 284–295.

[45] Naoya Yamamoto, Koji Nakano, Yasuaki Ito, Daisuke Takafuji, Akihiko Kasagi, and Tsuguchika Tabaru. "Huffman Coding with Gap Arrays for GPU Acceleration". In: *49th International Conference on Parallel Processing-ICPP*. 2020, pp. 1–11.

[46] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Zizhong Chen, and Franck Cappello. "Significantly Improving Lossy Compression for HPC Datasets with Second-Order Prediction and Parameter Optimization". In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 2020, pp. 89–100.