Missouri University of Science and Technology
Scholars' Mine

Computer Science Faculty Research & Creative Works

Computer Science

21 May 2021

Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures

Jiannan Tian

Cody Rivera

Sheng Di

Jieyang Chen

et. al. For a complete list of authors, see https://scholarsmine.mst.edu/comsci_facwork/1101

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_facwork

 Part of the Computer Sciences Commons

Recommended Citation

J. Tian et al., "Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures," *Proceedings of the 35th IEEE International Parallel and Distributed Symposium (2021, Portland, OR)*, Institute of Electrical and Electronics Engineers (IEEE), May 2021.

This Article - Conference proceedings is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Faculty Research & Creative Works by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures

Jiannan Tian*, Cody Rivera†, Sheng Di‡, Jieyang Chen§, Xin Liang§, Dingwen Tao*, and Franck Cappello‡¶
*School of Electrical Engineering and Computer Science, Washington State University, WA, USA
†Department of Computer Science, The University of Alabama, AL, USA
‡Mathematics and Computer Science Division, Argonne National Laboratory, IL, USA
§Oak Ridge National Laboratory, TN, USA
¶University of Illinois at Urbana-Champaign, IL, USA

***Abstract*—Today's high-performance computing (HPC) applications are producing vast volumes of data, which are challenging to store and transfer efficiently during the execution, such that data compression is becoming a critical technique to mitigate the storage burden and data movement cost. Huffman coding is arguably the most efficient Entropy coding algorithm in information theory, such that it could be found as a fundamental step in many modern compression algorithms such as DEFLATE. On the other hand, today's HPC applications are more and more relying on the accelerators such as GPU on supercomputers, while Huffman encoding suffers from low throughput on GPUs, resulting in a significant bottleneck in the entire data processing. In this paper, we propose and implement an efficient Huffman encoding approach based on modern GPU architectures, which addresses two key challenges: (1) how to parallelize the entire Huffman encoding algorithm, including codebook construction, and (2) how to fully utilize the high memory-bandwidth feature of modern GPU architectures. The detailed contribution is four-fold. (1) We develop an efficient parallel codebook construction on GPUs that scales effectively with the number of input symbols. (2) We propose a novel reduction based encoding scheme that can efficiently merge the codewords on GPUs. (3) We optimize the overall GPU performance by leveraging the state-of-the-art CUDA APIs such as Cooperative Groups. (4) We evaluate our Huffman encoder thoroughly using six real-world application datasets on two advanced GPUs and compare with our implemented multi-threaded Huffman encoder. Experiments show that our solution can improve the encoding throughput by up to 5.0× and 6.8× on NVIDIA RTX 5000 and V100, respectively, over the state-of-the-art GPU Huffman encoder, and by up to 3.3× over the multi-thread encoder on two 28-core Xeon Platinum 8280 CPUs.**

I. Introduction

With the ever-increasing scale of HPC applications, vast volumes of data are produced during simulation, resulting in a bottleneck for both storage and data movement due to limited capacity and I/O bandwidth. For example, Hardware/Hybrid Accelerated Cosmology Code (HACC) [16] (twice finalist nominations for ACM Gordon Bell Prize) produces 20 petabytes of data to store in one simulation of 3.5 trillion of particles with 300 timesteps, whereas leadership-class supercomputers such as Summit [37] have limited storage capacities (around 50∼200 PB) to be shared by hundreds of users. On the other hand, network and interconnect technologies in HPC systems advance much more slowly than computing power, causing intra-/inter-node communication cost and I/O bottlenecks to become a more serious issue in fast stream processing [6]. Compressing the raw simulation data at runtime and decompressing them before post-analysis can significantly reduce communication and I/O overheads and hence improving working efficiency.

Huffman coding is a widely-used variable-length encoding method that has been around for over 60 years [17]. It is arguably the most cost-effective Entropy encoding algorithm according to information theory, though some other coding methods such as arithmetic coding and range coding offer slightly better compression ratios in a few specific cases. As such, Huffman coding algorithm serves as the critical step in many general-purpose lossless compression software or libraries such as GZIP [9], Zstd [47], and Blosc [5]. It is also an integral part of many lossy compressors for image and video, such as JPEG [42]. Moreover, Huffman coding is also extensively used in many error-bounded lossy compressors (such as SZ [10, 40] and MGARD [2]), which have been very effective in compressing big scientific datasets with high data fidelity, as verified by many existing studies [24, 22, 46].

In this paper, we focus on parallelizing the entire Huffman encoding algorithm on GPUs rather than the decoding stage. On the one hand, the Huffman encoding algorithm plays a critical role in more and more HPC applications [18, 7] whose runtime performances are heavily relying on the GPU accelerators of supercomputers. On the other hand, compared with decompression performance, compression performance (or encoding efficiency) is particularly important to HPC applications, since large amounts of data need to be compressed on the fly and poor compression performance may substantially counteract performance improvement resulting from the reduced data size, causing inferior I/O performance [23]. By contrast, decompression generally happens only during the post-analysis, which has nothing to do with runtime performance of the simulation.

However, there are no efficient Huffman encoding algorithms designed for the GPU, leaving a significant gap that needs to be filled to meet modern HPC applications' requirements. Although many multi-thread Huffman encoding algorithms already exist (e.g., Zstd extended its capability to run on multiple CPU cores [1]), their design is limited to coarse-grained par-

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

1530-2075/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00097

allelism, which is unsuitable for today's modern GPU architectures (featuring massive single-instruction-multiple-thread (SIMT) mechanisms and high memory-bandwidth features). A few GPU-based Huffman encoders, such as Rahmani et al.'s encoder [32], adopt a rather simple parallel prefix sum algorithm to calculate the location of each encoded symbol, which cannot fully utilize the GPU memory bandwidth due to masses of movements of fragmented and variable-length data cells. Moreover, it is worth noting that traditionally, Huffman coding is used in cases where there are 8 bits per symbol (i.e., 256 symbols in total), which is far less than enough for many emerging HPC use cases. Error-bounded lossy compression, for example, often requires more than 8 bits per codeword (e.g., 16 bits are required if 65536 symbols are used in the codebook), because of potentially large amount of integer numbers produced after the error-bounded quantization step [40]. However, constructing a large Huffman codebook sequentially may incur a significant performance bottleneck to the overall Huffman encoding on GPUs, which was not addressed by any prior work.

To address the significant gap, we present an efficient Huffman encoder on GPUs, which is compatible with many emerging HPC scenarios. The basic idea is leveraging a battery of techniques to optimize performance on modern GPU architectures, based on an in-depth analysis of Huffman encoding stage. Specifically, we optimize the parallel Huffman codebook construction for GPUs and significantly reduce the overhead of constructing the codebook that involves a large number of symbols. Moreover, we propose a novel reduction-based encoding scheme, which can significantly improve the memory bandwidth utilization by iteratively merging codeword groups. To the best of our knowledge, our proposed and implemented Huffman encoder is the first work that achieves hundreds of GB/s encoding performance on V100 GPU. The detailed contributions are listed as follows.

- We carefully explore how parallelization techniques can be applied to the entire Huffman encoding algorithm including histogramming, codebook construction, and encoding, and optimize each stage using state-of-the-art CUDA APIs such as Cooperative Groups.
- We develop an efficient parallel codebook construction on GPUs, especially for scenarios requiring a codebook with a large number of symbols, opening new possibilities for non-traditional use cases of Huffman coding.
- We propose a novel reduction-based encoding scheme that iteratively merges the encoded symbols, significantly improving GPU memory bandwidth utilization.
- We evaluate our Huffman encoder on six real-world datasets using two state-of-the-art GPUs and compare it with other state-of-the-art Huffman encoders on both CPUs and GPUs. Experiments show that our solution can improve the encoding throughput by up to 6.8× on V100 and 3.3× on CPUs.

In §II, we present the background for Huffman coding and parallel algorithms for its codebook construction. In §III, we discuss the limitation of current Huffman encoding on GPUs. In §IV, we present our proposed parallel Huffman codebook construction and reduction-based encoding scheme on GPUs. In §V, we show the experimental evaluation results. In §VI and §VII, we discuss the related work and conclude our work.

II. Background

A. Huffman Coding and Its Emerging Applications

Huffman coding is a fundamental data compression algorithm proposed by David Huffman in 1952 [17]. In essence, it assigns codes to characters such that the length of the code depends on the relative frequency of the corresponding character (a.k.a., input symbol). Huffman codes are variable-length and prefix-free. Here prefix-free means no code is a prefix of any other. Any prefix-free binary code can be visualized as a binary tree (called the Huffman tree) with the encoded characters stored at the leaves.

In recent years, data reduction attracts more and more attention in the HPC field, and Huffman coding becomes an integral part of many data reduction techniques such as error-bounded lossy compression [10, 25, 2]. For multiple HPC use cases, Huffman coding usually needs to be customized with a large number of symbols, instead of using the classic Huffman coding with only 256 symbols/characters in the codebook. For example, SZ requires a customized Huffman coding with 65536 quantization bins in default, such that a large majority of integer codes generated by its quantization could be covered by the encoding scheme. Such a customized Huffman coding is particularly critical when the data is difficult to be predicted accurately, which is very common in scientific datasets.

Another important scenario is n-gram compression [21]. For example, some languages have morphology in the structure of words or morphemes, and it is important to utilize this syllable-based morphology for developing an efficient text compression approach for these languages. Nguyen et al. proposed a method [28] to partition words into its syllables and then to produce their bit representations for compression. The number of bits in syllables (symbols) depends on the number of entries in the dictionary file. As another example, segmenting, encoding, and decoding DNA sequences based on n-gram statistical language model is a critical research topic in bioinformatics to handle the vast volumes of DNA sequencing data. Specifically, in this work [21], researchers find the length of most DNA words/symbols (e.g., 12∼15 bits) and build an n-gram biology language model by analyzing the genomes of multiple model species. Then, they design an approach to segment the DNA sequences and encode them accordingly.

In all the above cases, Huffman coding may require generating a codebook with a large number of symbols which is usually far smaller than that of the input codewords. However, since such a large codebook is generated serially, codebook construction can become a significant bottleneck, especially on small to medium-sized datasets. Thus it is vital to develop an approach to build a Huffman codebook efficiently.

B. PRAM Model

PRAM is a classic model to describe parallel algorithms where multiple processors are attached to a single memory en-

tity. It essentially assumes that (1) a set of processors of uniform type exist, and (2) all the processors share a common memory unit with their accesses equal (via a memory access unit). This model is made independent from specific hardware by introducing some ideal assumptions. The conventional taxonomy of read/write (R/W) conflicts emphasize on the concurrency (denoted by C) and exclusiveness (denoted by E). Thus, there are four different constraints that have been enforced on the PRAM model: EREW, ERCW, CREW, CRCW. In this paper, we focus on the CREW PRAM model used by our parallel codebook construction algorithm and implement it on the GPU.

C. Parallel Huffman Codebook Construction

The serial Huffman codebook construction algorithm with the complexity of $\mathcal{O}(n \log n)$ constructs a naïve binary tree—a data structure not well-suited for the GPU memory. Specifically, the naïve Huffman tree has an inefficient GPU memory access pattern, which would incur a significant performance overhead on codebook construction. This is confirmed by our experiment: constructing a Huffman codebook with 8,192 input symbols takes 144 ms on NVIDIA V100 GPU, which degrades the throughput of compressing 1 GB data to less than 10 GB/s.

Obviously, it is very important to develop an efficient parallel codebook construction algorithm on GPU to match the high speed of other stages in Huffman encoding. To this end, we review the literature carefully for existing parallel Huffman tree and codebook construction algorithms. Larmore and Przytycka [20] proposed a parallel Huffman tree construction algorithm under the CREW PRAM model using n processors with $\mathcal{O}(\sqrt{n} \log n)$ time per processor, the first algorithm whose processor count scales linearly with the number of input symbols. Here both the number of processors and the number of input symbols are n. However, the proposed algorithm is known for its inefficiency of performing $\mathcal{O}(n^2)$ work. Millidiú et al. [27] later proposed another CREW PRAM algorithm for the same problem, with n processors, $\mathcal{O}\big(H \cdot \log \log \frac{n}{H}\big)$ time per processor, and $\mathcal{O}(n)$ work, where H is the length of the longest codeword. Since Huffman codes can be up to $\mathcal{O}(n)$ in length, the proposed algorithm has a worst case performance of $\mathcal{O}(n)$ per processor, but this is rarely encountered in practice, especially in HPC scenarios whose floating-point data tends to be mostly smooth and predictable.

The previously discussed algorithms all output Huffman trees, where the trees still need to be traversed serially to generate a codebook. To address this issue, Ostadzadeh et al. proposed a CREW PRAM algorithm that directly generates the codebook [31]. To do this, it generates the length of each symbol, with n processors and $\mathcal{O}(H \cdot \log \log \frac{n}{H})$ time per processor, and then converts the generated symbol lengths into codes, with n processors and $\mathcal{O}(H)$ time per processor. We propose our Huffman codebook construction method based on this algorithm due to its direct output of Huffman codes as well as its outstanding performance on most Huffman work.

III. Problem Statement

In this section, we first discuss the scalability constraints posed by modern GPU architectures, especially the CUDA architecture. We then analyze the performance bottleneck of the state-of-the-art Huffman encoding method. Finally, we formulate our research problem under certain constraints.

A. Scalability Constraints from GPU Hardware

First, we analyze the scalability constraints from the GPU hardware perspective. The thread is the basic programmable unit that allows the programmer to use the massive amount of CUDA cores. CUDA threads are grouped at different levels, including warp, block, and grid levels.

a) Rigid SIMD-ness Against Randomness: 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, it may cause discrepancy from the PRAM model, because diverged threads add extra overhead to the execution. Thus, we relax the use of the PRAM model. Nevertheless, the GPU's massive parallelism allows our implementation to exhibit the theoretical complexity of parallel Huffman coding under PRAM.

b) Block-Shared Memory Lifecycle Binding: 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.

Therefore, we must make use of both coarse- and fine-grained parallelization in our design due to the scalability constraints from the CUDA architecture. For coarse-grained parallelization, we divide the data into multiple independent chunks, not only because it is easy to map chunks to thread blocks and utilize local shared memory, but also because it will facilitate the reverse process, decoding. In addition, coarse-grained chunking can improve performance significantly with only a minimum overhead in compression ratio. For fine-grained parallelization, the state-of-the-art work [32] only addresses the encoding stage rather than Huffman codebook construction. In the next section, we will further analyze the performance issue of the existing fine-grained encoding approach.

B. Fully Enabling GPU's High Memory Bandwidth

Compared to compute-bound algorithms such as matrix-matrix multiplication, Huffman encoding tends to be more memory-bound [15]. In practice, Huffman encoding that has

TABLE I: Parallelism implemented for Huffman coding's subprocedures (kernels). "sequential" denotes that only 1 thread is used due to data dependency. "coarse-grained" denotes that data is explicitly chunked. "fine-grained" denotes that there is a data-thread mapping with little or no warp divergence.

	sequential	coarse-grained	fine-grained	data-thread many-to-one	data-thread one-to-one	atomic write	reduction	prefix sum	
histogram									boundary
blockwise reduction			•	•		•	•		sync block
gridwise reduction			•	•		•	•		sync device
build codebook									
get codeword lengths		•	•	•	•	•			sync grid
get codewords			•		•	•			sync grid
canonize									
get numl array			•		•	•		•	sync grid
get first array (RAW)	•			•					sync grid
canonization (RAW)	•			•					sync grid
get reverse codebook			•						sync device
Huffman enc.									
`reduce`-merge		•	•	•			•		sync block
`shuffle`-merge		•	•		•				sync device
get blockwise code len		•	•		•			•	sync grid
coalescing copy		•	•		•				sync device

not been highly optimized usually underutilizes GPU memory bandwidth. In this section, we analyze the root causes of the existing method's low memory bandwidth utilization, which will guide our design of an efficient Huffman encoding that fully enables high GPU memory bandwidth.

a) Variable Lengths of Codewords: Due to the variable lengths of Huffman codes, the serial encoding must calculate the location of each encoded symbol and perform a *write* operation. Thus, it is easy to implement a relatively efficient Huffman encoding on CPU because of the CPU's sophisticated branch prediction and caching capabilities, which can effectively mitigate the irregular memory access pattern, even with a relatively low memory bandwidth (e.g., Summit [37] has about a theoretical peak memory bandwidth of about 60∼135 GB/s). In comparison, the GPU has a much higher memory bandwidth but lower branch prediction and caching capabilities. We note that coarse-grained parallel encoding (i.e., chunking data and assigning each chunk to a processor) cannot fully utilize the GPU's high memory bandwidth, as it disregards memory coalescing. This is confirmed by a prior work, CUSZ [41] where coarse-grained parallel encoding only achieves a throughput of about 30 GB/s on the V100 (1/30 of the peak).

b) Limitations of Existing GPU Encoding Method: A prefix-sum based Huffman encoding algorithm was proposed to make use of the massive parallelism on GPUs [32]. In this method, before memory copies, a classical parallel prefix-sum algorithm is used to calculate the write locations of all encoded symbols. However, it has two main drawbacks to limit its use in all scenarios. As discussed in Section II-A, many scientific applications generate the data that contains the symbols each with more than one byte, thus, the lengths of the corresponding codewords are fairly variable and diverse. On the one hand, the prefix-sum based method does not exhibit good performance in the high-compression-ratio use cases (i.e., short codeword length averagely) [41]. This is because, by moving only few bits in a single-/multi-byte codeword, the codeword-length agnostic solution makes low use of the GPU memory bandwidth given the same degree of launched parallelism, which is also confirmed by our experiment—the prefix-sum based method can only achieve a throughput of 37 GB/s on V100 on a dataset with the average codeword length of 1.02717 bits. On the other hand, even though the last step—concurrent write to global memory—is theoretically low in time complexity (i.e., $\mathcal{O}(1)$), the hardware implementation makes it tend to be CREW (exhibiting memory contention). For example, our experiment shows that the concurrent iterative solution has similar performance as one-time exclusive parallel write.

Overall, in this work, we aim to fully enable the high GPU bandwidth for Huffman encoding in a wide range of emerging scenarios (i.e., more than 256 symbols in the codebook) without loss of generality. Note that achieving high performance on GPUs requires to rigidly follow the coalescing and SIMD characteristics, which are against the irregular memory access pattern. Therefore, in order to develop a high-performance Huffman encoder on GPUs, we need (1) to balance the SIMD-ness from the GPU programming model and the inherent randomness of Huffman coding and (2) to develop an adaptive solution to solve the low memory bandwidth utilization issue.

IV. Design Methodology

In this section, we propose our novel GPU Huffman encoding design for the CUDA architecture. We propose several optimizations for different stages. Specifically, we modularize the Huffman encoding into the following four stages: (1) calculating the frequencies of all input symbols, namely, histogramming; (2) Huffman codebook generation/construction based on the frequencies; (3) canonizing the codebook and generating the reverse codebook for decoding; and (4) encoding according to the codebook, and concatenate Huffman codes into a bitstream. We first propose an efficient, fine-grained parallel codebook construction on GPUs, especially for scenarios requiring a large number of symbols (stage 2). We then propose a novel reduction-based encoding scheme that iteratively merges the encoded symbols, which significantly improves memory bandwidth utilization (stage 4). A summary of our proposed techniques is shown in Table 1. It shows the parallelism and CUDA APIs for each substage. We highlight the corresponding *granularity*, *model*, *scalability*, and *complexity*.

A. Histogramming

The first stage of Huffman encoding is to build a histogram representing the frequency of each integer-represented symbol from the input data. The GPU histogramming algorithm in use is derived from that proposed by Gómez-Luna et al. [13]. This algorithm minimizes conflicts in updating the histogram bin locations by replicating the histogram for each thread block and storing the histogram in shared memory. Where possible, conflict is further reduced by replicating the histogram such that each block has access to multiple copies. All threads inside a block read a specified partition of the input and use atomic operations to update a specific replicated histogram. As each block finishes its portion of the predicted data, the replicated

histograms are combined via a parallel reduction into a single global histogram, which is used to construct the final codebook.

B. Two-Phase Canonical Codebook Construction

In the second stage, we implement an efficient parallel Huffman codebook construction algorithm on the GPU and modify it to produce canonical codes for fast decoding.

1) Codebook Construction: Now that we have a single global histogram, the next step is to efficiently construct a base codebook. We implement the parallel codebook construction algorithm proposed by Ostadzadeh et al., as described earlier, on the GPU [31]. This algorithm provides a parallel alternative to the original Huffman codebook construction algorithm in $\mathcal{O}(n \log n)$ and directly generates codewords. To the extent of our knowledge, this algorithm has not been implemented on the GPU elsewhere. The algorithm is split into two phases, (1) `GenerateCL`, which calculates the codeword length for each input symbol, and (2) `GenerateCW`, which generates the actual codeword for each input symbol. Both phases utilize fine-grain parallelism, with one thread mapped to one input symbol or intermediate value. Additionally, both phases are implemented as single CUDA kernels with Cooperative Groups [8], which we use to synchronize an entire CUDA grid. We describe both phases in Algorithm 1, with our modifications colored blue, and emphasize our GPU implementation in the following discussion. We refer readers to [31] for more details of the original algorithm.

`GenerateCL` takes `F`, a sorted n-symbol histogram, and outputs `CL`, a size n array of codeword lengths for each symbol. This phase of the algorithm runs in $\mathcal{O}(H \cdot \log\log \frac{n}{H})$ time on PRAM, where H is the longest codeword. Its parallelism can be derived from the fact that for a given set of Huffman sub-trees, all sub-trees whose total frequencies are less than the sum of the two smallest sub-tree frequencies can be combined in parallel, a result which Ostadzadeh et al. prove [31].

Before `GenerateCL` is launched, the histogram is sorted in ascending order using Thrust [30]. This operation is low-cost, as n is relatively small compared to the input data size. Once launched, lines 1–4 of Algorithm 1 initialize the array `lNodes` with each input symbol's leaf node, and initialize the array (and queue) `iNodes` as empty. Each array element describes a Huffman node, storing its total frequency, its leader, or topmost parent, and auxiliary information. To increase memory access efficiency, each of these arrays are stored in *structure-of-arrays format*, rather than the intuitive array-of-structures format. The advantage of this is that accesses to single fields of consecutive elements are coalesced.

Next, lines 5–26 construct the Huffman tree while there are still leaf and internal nodes to process. Lines 6–16 create a new node t from the smallest two leaf or internal nodes, and selects leaf nodes whose frequencies are less than t. `ParMerge` (line 17) merges these selected leaf nodes and internal nodes, which are sorted by ascending frequency, together. This is done using a $\mathcal{O}(\log\log n)$ parallel merge under the PRAM model [31].

To implement this merge, we customize the parallel GPU Merge Path algorithm proposed by Green et al. [14] for our intermediate structure-of-arrays representations in `GenerateCL`. In practice, this algorithm does not attain the proposed theoretical time complexity, as its time complexity is $\mathcal{O}(n/p + \log n)$, with p being the number of partitions (i.e., the number of thread blocks). However, we use a number of thread blocks proportional to the number of streaming multiprocessors (SMs), making n/p in practice $\mathcal{O}(\log n)$. This component of parallel codebook construction employs coarse-grain parallelism, as once the merge partitions are determined, each partition is merged serially. To remove the overhead of calling a separate kernel with dynamic parallelism, we incorporate `ParMerge` into the same kernel as the rest of `GenerateCL`, keeping unneeded threads idle until the merging phase. Once these nodes are merged, they are melded together into new nodes, with the appropriate `CL` values and leaf nodes being updated, on lines 18–25, and the iteration is repeated as appropriate.

`GenerateCW` takes `CL` as input and outputs `CW` (i.e., the actual codewords). It takes $\mathcal{O}(H)$ time per thread in the PRAM model, where H is the longest codeword, and our GPU implementation

Algorithm 1: Modified parallel Huffman code construction based on [31].

• **GenerateCL — codeword length procedure**

```
1  iNodes <- ∅, c <- 0
2  for all i in [0...n) concurrently do
3     lNodes[i].freq <- F[i], lNodes[i].leader <- (-1), CL[i] <- 0
4  end for                ▷ Initialize array of leaf nodes and set codeword lengths to zero
5  while c < n or iNodes.size > 1 do
6     t <- NewNodeFromSmallestTwo(lNodes, c, iNodes)
7     iNodes <- iNodes ∪ {t}                              ▷ Create first internal node
8     for all i in [c...n) concurrently do
9        if lNodes[i].freq < t.freq then
10          copy[i-c] <- lNodes[i]
11          copy.size <- AtomicMax(i-c+1, copy.size)
12       end if
13    end for
14    c <- c + copy.size                                  ▷ Select eligible leaf nodes
15    ℓ <- copy.size + iNodes.size - 1                          of nodes ↩P
16    s <- iNodes.size - 1 - (ℓ mod 2)  ▷ Ensure temp will have an even number
17    temp <- ParMerge(copy, iNodes[0...s) )     ▷ Merge leaf and remainder of
18    iNodes <- iNodes[s...iNodes.size]                         internal nodes
19    for all i in [0...temp.size/2) concurrently do
20       iNodes[iNodes.size+i] <- Meld(temp[2·i], temp[2·i+1])
21    end for
22    iNodes.size <- iNodes.size + temp.size   ▷ Meld each two adjacent nodes
23    for all i in [0...n) concurrently do                      in parallel
24       UpdateLeafNode(lNode[i], CL[i])  ▷ Update codeword lengths and leader
25    end for                                               pointers for leaf nodes
26 end while
```

• **GenerateCW — codeword generation procedure**

```
27 ParReverse(CL)
28 CCL <- CL[0], PCL <- CL[0], FCW <- the codeword 0, CDPI <- 0
29 First[CCL] <- 0, Entry[CCL] <- 0
30 while CDPI < n-1 do
31    newCDPI <- n-1
32    for all i in [CDPI...n-1) concurrently do
33       if CL[i] > CCL then
34          newCDPI <- AtomicMin(newCDPI, i)
35       end if
36    end for          ▷ Count number of codewords with current codeword length
37    for all i in [CDPI...newCDPI) concurrently do
38       CW[i] <- FCW + (CDPI - (newCDPI-i-1))
39    end for                    ▷ Build codewords of a given CCL in reverse
40    First[CCL] <- InvertCW(CW[newCDPI-1])
41    Entry[CCL] <- Entry[PCL] + (newCDPI-CDPI)   ▷ Record decoding metadata
42    CLDiff <- CL[newCDPI] - CL[newCDPI-1]
43    FCW <- CW[CDPI] + 1 · pow(2, CLDiff)
44    PCL <- CCL, CCL <- CL[newCDPI], CDPI <- newCDPI      ▷ Prepare for next
45 end while                                                   codeword length
46 for all i in [1...n) concurrently do
47    CW[i] <- InvertCW(CW[i])
48 end for          ▷ Invert codewords, making them canonical and reordering them
49 ParReverse(CW)
```

is consistent with this theoretical complexity (see Table III). This phase of the algorithm utilizes fine-grained parallelism, as codewords are generated by individual threads. It assigns numerically increasing codewords to all symbols with a given codeword length `CCL` in parallel (lines 31–39). If there are more codewords to generate that are longer than `CCL`, first, `CCL` and other local variables are updated (lines 42–44). Also, the first codeword for the new codeword length is generated by incrementing the existing codeword and left-shifting the codeword by the difference between the old and new codeword lengths (line 43). Finally, lines 31–39 are repeated. Once all codewords are generated, `CW` is reversed, and the codewords are resorted by the actual input symbol they represent to generate the forward codebook.

It is worth noting that throughout codebook construction, we use the state-of-the-art CUDA Cooperative Groups instead of existing block synchronization for global synchronization [8]. This is because CUDA blocks are limited to 1024 threads, and we employ fine-grained parallelism for codebook sizes greater than 1024. An alternative technique to achieve the same global synchronization is to separate the parallel regions into different kernels. We chose Cooperative Groups over this technique to avoid the overhead of kernel launches and `cudaDeviceSynchronize`. Our profiling for the NVIDIA V100 GPU reveals that a CUDA kernel launch takes about 60 microseconds (60 μs), and each of our parallel regions performs very little work. Also, since many of these parallel regions are performed in a loop, we effectively avoid unnecessary CPU-GPU transfers.

2) Canonizing Codebook: A canonical Huffman codebook [33] holds the same bitwidth of each codeword as the original Huffman codebook (i.e., base codebook). Its bijective mapping between input symbol and Huffman codeword is more memory-efficient than Huffman-tree traverse for encoding/decoding. The time complexity of serially building a canonical codebook from the base codebook is $\mathcal{O}(n)$, where n is the number of symbols, and is sufficiently small compared with the data size. By using a canonical codebook, we can (1) decode without the Huffman tree, (2) efficiently cache the reverse codebook for high decoding throughput, and (3) maintain exactly the same compression ratio as the base Huffman codebook.

We start our implementation which contains a partially-parallelized canonization CUDA kernel, utilizing Cooperative Groups. It performs (1) linear scanning of the base codebook (sequentially $\mathcal{O}(n)$), which is parallelized at fine granularity with atomic operations; (2) loose radix-sorting of the codewords by bitwidth (sequentially $\mathcal{O}(n)$), which cannot be parallelized because of the intrinsic RAW dependency; and (3) building the reverse codebook (sequentially $\mathcal{O}(n)$), which is enabled with fine-grained parallelism. The canonization process is relatively efficient—it only costs about 200 us to canonize a 1024-codeword codebook on V100.

Nevertheless, our choice of codebook construction algorithm provides a sufficient base for further optimization. Since the output of `GenerateCL` and input of `GenerateCW`, `CL`, is sorted by codeword length, the intrinsic RAW dependency of (2) is removed. In fact, the existing codewords generated by `GenerateCW` are almost canonical, except for the fact that given two codewords c_1 and c_2 where c_2 is longer than c_1 and ℓ is c_1's length, the most significant ℓ bits of c_2 are numerically greater than c_1. The opposite should be the case, as efficient decoding relies on this fact. To alleviate this problem, the codewords for each level are generated in decreasing order per level (line 38 of Algorithm 1), and the bits in each codeword are then inverted before they are stored (line 47). Moreover, additional metadata to facilitate decoding also needs to be generated in $\mathcal{O}(1)$ extra time with little storage overhead. The metadata that we generate consists of two H-element arrays, where H is the longest codeword's length. The `First` array contains the first codeword for a given length, and the `Entry` array contains a prefix sum of the number of codewords shorter than that length. Right after all the codewords for a given length are generated, the `First` and `Entry` arrays are updated appropriately at that index (lines 40–41). Both arrays are used for efficient treeless canonical decoding.

The theoretical time complexity of our codebook construction, including our modifications to generate canonized codes, is $\mathcal{O}(H \cdot \log\log \frac{n}{H})$; however, due to the particular implementation of merge, which is the most expensive operation, the practical complexity is increased to $\mathcal{O}(H \cdot \frac{H}{n}/p + \log \frac{H}{n})$, where H is the longest codeword length, n is the number of symbols, and p is the number of blocks launched. Nevertheless, due to H in practical being small and p being sufficiently large, we observe the complexity of $\mathcal{O}(\log n)$, and our experiments are consistent with this (see Table III).

C. Encoding

We propose an iterative merge comprised of `reduce`-merge and `shuffle`-merge that is the key to improving memory bandwidth utilization for encoding. In each iteration, every two codewords are merged into one, with their lengths summed up. Formally, given two code-length tuples $(a, \ell)_{2k}$ and $(a, \ell)_{2k+1}$, we define

$$\texttt{Merge}\big((a,\ell)_{2k}, (a,\ell)_{2k+1}\big) = (a_{2k} \oplus a_{2k+1}, \ell_{2k} + \ell_{2k+1}),$$

where $\oplus$ represents for concatenating bits of a_{2k+1} right after bits of a_{2k}. Note that the merge is not *commutative* for the encoded symbols and must follow the original order.

We further split this merge into `reduce`-merge and `shuffle`-merge phases. Note that the first merge includes a codebook lookup to get the codewords. After that, the merge is performed iteratively on two codewords each time.

a) Reduce-Merge: The practical average bitwidth of Huffman codewords can be fairly low (e.g., most are 1 or 2 bits in many HPC datasets), while data movement is in terms of single-/multi-byte words (i.e. a multiple of 8 bits, such as `uint{8,16,32}_t`). This can significantly hurt the performance due to an extravagant use of threads on the GPU if the data-thread mapping is too fine, e.g., 1-to-1. In each iteration of reduction, active threads for data movement halve every iteration before bits saturates the representing words, leading to a waste of parallelism. Hence, we map multiple

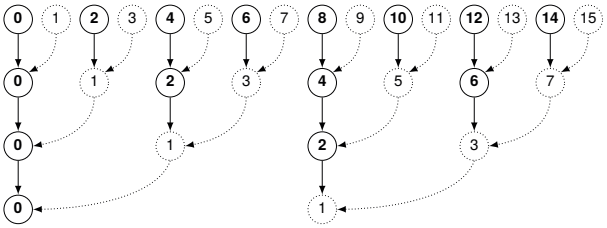

Fig. 1: reduce-merge of 8-to-1.

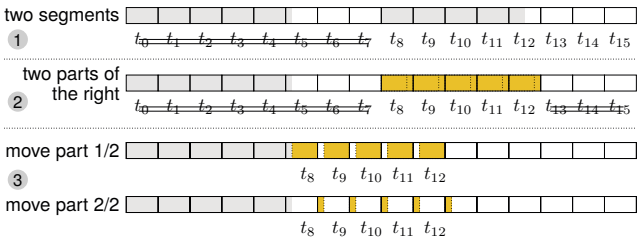

Fig. 2: Two-step batch move of grouped and typed data. By batch-moving the right grouped data, warp divergence is decreased.

codewords to one thread to merge until the merged bitwidth exceeds that a representing word can handle. More precisely, the condition of stopping `reduce`-merge is that the average bitwidth of the merged codeword exceeds half of the bitwidth of the data type. For example, merging codewords with an average bitwidth of 2.3 bits for 3 times is expected to result in an $8\times$ length, averagely 18.4 bits, and end before going beyond 32 bits. The number of iterations (denoted by *reduction factor* r) can be determined by the entropy in bits of input data (from histogram). After `reduce`-merge is done, the number of merged codewords is shrunk by $2^r\times$.

b) `Shuffle`-Merge: After `reduce`-merge completes, threads are grouped to move the corresponding merged codewords, with one thread assigned to one unit of typed data. Following the same scheme as `reduce`-merge, a right group of typed data is moved to append to its corresponding left group. To provide more detail, a left group $(A,\ell)_{2k}$, with data segment (the representation data type or word is marked with W) and its length ℓ_W, has a starting index i_{2k} (already known) and an ending index $i_{2k,\bullet}=\left(i_{2k}+\ell_{2k}/\ell_W\right)$ (easy to calculate). Moreover, the ending bit's location can be calculated as $\ell_{2k,\bullet}=(\ell_{2k} \mod \ell_W)$, and the number of residual bits is $\ell_{2k,\circ}=\ell_W-\ell_{2k,\bullet}$. Thus, the right group $(A,\ell)_{2k+1}$ needs $\lceil \ell_{2k+1}/\ell_H \rceil$ threads for data movements. For each thread, the $\ell_{2k,\circ}$ bits are first moved to fill the residual bits, and then the $\ell_{2k,\bullet}$ bits are moved right after the $\ell_{2k,\circ}$ bits (in the next typed data cell), as shown in Figure 2. Note that this process is free of data contention. The iterative process will be performed for s times (denoted by *shuffle factor*) until a dense bitstream is formed. We also note that `shuffle`-merge can be finished within a continuous memory space of 2^s typed data cells.

c) Interface: Our encoding kernel is interfaced as

`ReduceShuffleMerge`$\langle M, r\rangle$(in, out, metadata).

We expose two independent parameters to describe the problem size and the two merge phases—magnitude M, reduction

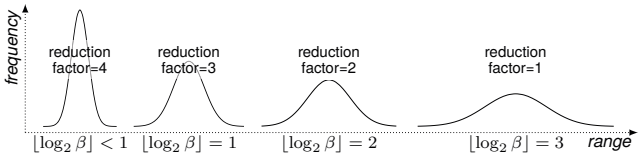

Fig. 3: Average bitwidth being a consideration to decide reduction factor.

factor r, and shuffle factor is derived from $s=M-r$. In total, there are M iterations in the entire encoding stage. After that, we generate a densely encoded bitstream for this chunk. We use $N\equiv 2^M$ to denote the problem size for each chunk, and $n\equiv 2^s$ to denote the reduced size before `shuffle`-merge.

We use the average codeword bitwidth to determine the reduction factor r. Specifically, given a word W of length ℓ_W, a "proper" r is tentatively determined according to $\lfloor \log \beta \rfloor + r + 1 = \log \ell_W$, such that the length of the r-time-merged codeword $\ell^{(r)}$ is expected as $\ell_W/2 \le \ell^{(r)} < \ell_W$, toward maximized memory bandwidth utilization. Note that there are certain codewords that exceed 32 bits after r-time merge, and these (denoted by "breaking" in Table II) are filtered out and handled otherwise.

We give a quantitative example to illustrate the importance of determining a proper reduction factor and how magnitude may affect the performance due to longer `shuffle`-merge. We compare a magnitude of $\{12, 11, 10\}$ and reduction factor of $\{4, 3, 2\}$ on `Nyx-Quant` from `baryon-density` field, whose average bitwidth is 1.02717 with 1024 symbols. The performance is shown in Table II. We find that the combination of $M{=}10$ and $r{=}3$ results in the highest performance, which can be empirically generalized. Although reducing magnitude would result in more metadata, as we prioritize the performance in this work, we choose the combination of $M=10$ and $r=3$ for the following evaluation.

TABLE II: Performance (in GB/s) of our Huffman encoding with different chunk magnitudes (mag.) and reduction factors on Longhorn and Frontera.

reduction factor	mag. →	Longhorn 2^{12}	Longhorn 2^{11}	Longhorn 2^{10}	Frontera 2^{12}	Frontera 2^{11}	Frontera 2^{10}	breaking
(16×)	4	227.60	274.40	291.04	110.94	124.42	133.84	0.000434%
(8×)	3	191.41	274.42	314.63	94.27	124.56	135.86	0.003277%
(4×)	2	68.32	106.87	172.54	42.70	55.53	79.45	0.007536%

d) Complexity: For `reduce`-merge, we map multiple codewords for reduction such that we can effectively move more bits against the given holding codeword W with length ℓ_W. We maintain a block of 2^s threads for data movement. The time complexity is $2^{(r-i)}$ for the ith iteration, and the time complexity for reduction in parallel is $\sum_1^r 2^{r-i}$. Note that the operations are homogeneous (i.e. there is only one if-branch) without being affected from warp divergence.

For `shuffle`-merge, the magnitude of reduced data chunk remains s, and each typed data cell is assigned with a thread. At the ith iteration, the chunk is split into $(s+1-i)$ groups. Compared with reduction, shuffle creates warp divergence at a factor of 2, given the groups are to merge with their corresponding other groups. With s parallel `shuffle`-merges, the total time complexity is $\mathcal{O}(s)$. Note that bank conflicts may affect performance, as the read and written locations inevitably

overlap due to the variable length [1]. This proof-of-concept work is intended to show the effectiveness of the method, the effect of bank conflict is to be further investigated.

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 using the Frontera supercomputer [11] and its subsystem Longhorn [26]. We perform our experiments on an NVIDIA Tesla V100 GPU from Longhorn and an NVIDIA Quadro RTX 5000 from Frontera, and compare with CPU implementations on two 28-core Intel Xeon Platinum 8280 CPUs from Frontera.[2] We use NVIDIA CUDA 10.1 and its default profiler to measure the time. In this section, we use TU to denote Turing RTX 5000 and V to denote Volta V100.

2) Comparison Baselines: CUHD [43] and cuSZ [41] are two state-of-the-art Huffman encoders for GPUs, but both of them are coarse-grained and embarrassingly parallelized. We note that CUHD's source code [44] only focuses on GPU Huffman decoding and implements a serial CPU Huffman encoder, so we compare our GPU Huffman encoder only with cuSZ [41]. We also compare our GPU encoder with the serial encoder implemented in SZ [39] and with our implemented multi-thread encoder[3] on single and multiple CPU cores, respectively.

3) Test Datasets:

a) Single-Byte Based Datasets: Generic Huffman coding takes one byte per symbol, hence, at most 256 symbols in total. Without reinterpreting a bytestream into multi-byte (un)signed integers or floating-point numbers, the generic encoding simply treats all input data as `uint8_t`. Our evaluation includes these popular datasets: (1) `enwik8` and `enwik9` from *Large Text Compression Benchmark* [19], the first 10^8 and 10^9 bytes of XML-based English Wikipedia dump; (2) `nci` from *Silesia Corpus* [35], a file for chemical database of structures; (3) `mr` from *Silesia Corpus* [35], a sample file of medical magnetic resonance image; and (4) `Flan_1565` from *SuiteSparse Matrix Collection* [36], a sparse matrix in Rutherford Boeing format.

b) Multi-Byte Based Datasets: We also evaluate two datasets with multiple bytes as a symbol: (5) `Nyx-Quant` is the quantization codes generated by SZ (a famous error-bounded lossy compression for HPC data) based on *Nyx*'s (cosmological simulation) `baryon_density` from *Scientific Data Reduction Benchmarks* [34]; and (6) `gbbct1.seq` is a sample DNA sequence data from *GenBank* [4], where every k nucleotides (k-mer) forms a symbol. We test $k = \{3, 4, 5\}$ in our evaluation.

[1] Due to the page limit, we refer readers to the discussion of time complexity at https://github.com/szcompressor/huffre/blob/main/doc/benchmark.md.

[2] V100 has 16GB HBM2 memory at 900 GB/s; RTX 5000 has 16GB GDDR6 memory at 448 GB/s; Xeon 8280 has 192GB of 2933 MT/s DDR4 memory.

[3] Note that SZ's current OpenMP version [38] only divides data into multiple blocks and applies its compression to each block independently.

B. Experimental Results

1) Parallel Codebook Construction: Call back to §IV-B1, we observe a practical complexity of $\mathcal{O}(H \cdot \log(n/H))$ or approximately $\mathcal{O}(\log n)$, where H and n are the height of the built tree and the problem size, respectively. Although we exhibit speedups in codebook construction for $n = 256$ ranging from 2.0∼2.9 in Table V, greater benefits come from using more input symbols since the $\mathcal{O}(n \cdot \log n)$ serial construction algorithm scales slower. To demonstrate this speedup, we use the `gbbct1.seq` gene dataset with k-mer analytics, where $k = 3, 4, 5$. We also evaluate our codebook construction on the quantization codes generated by cuSZ from the Nyx dataset. Note that data other than the 4 bases of DNA are stored in `gbbct1.seq`, and as a result, the number of input symbols needed is greater than 4^k.

TABLE III: Breakdown comparison of Huffman codebook construction time (in milliseconds) on RTX 5000 and V100 with different numbers of symbols.

		ref. CPU	TU	V	TU	V	TU	V
cuSZ serial	#sym.	serial	gen. codebook		canonize		total time	
Nyx-Quant	1024	0.045	3.051	3.689	0.095	0.115	3.416	3.804
3-mer	2048	0.208	8.381	9.760	0.242	0.284	8.623	10.044
4-mer	4096	0.695	20.148	24.684	0.519	0.663	20.667	25.347
5-mer	8192	1.806	61.748	59.092	1.453	1.449	63.201	60.541
Ours parallel	#sym.	serial	gen. CL		gen. CW		total time	
Nyx-Quant	1024	0.045	0.315	0.383	0.134	0.161	0.449	0.544
3-mer	2048	0.208	0.494	0.570	0.180	0.209	0.674	0.779
4-mer	4096	0.695	0.633	0.682	0.173	0.185	0.806	0.867
5-mer	8192	1.806	1.330	1.145	0.154	0.187	1.484	1.332

Table III compares GPU codebook construction between cuSZ's serial implementation and our parallel implementation on several datasets with different numbers of input symbols. Ours exhibits more dramatic speedups over cuSZ's when using more input symbols, consistent with our theoretical analysis and performing up to 45.5× faster when creating a codebook for 8192 symbols. Note that ours is no faster than the CPU serial construction when the number of symbols is below 8192. This is because caching, high frequency, and superior branch prediction in combination result in low latency of CPU threads. However, to avoid long histogramming and other CPU-GPU data transfers, it is desirable to purely perform codebook construction on the GPU.

TABLE IV: Performance (in milliseconds) of multi-thread codebook construction with different numbers of input symbols. The length of the bar under the number reflects the execution time.

	#sym.	serial	1 core	2 cores	4 cores	6 cores	8 cores
Nyx-Quant	1024	**0.045**	0.219	0.469	0.622	0.700	0.840
3-mer	2048	**0.208**	0.361	0.691	1.101	1.122	1.303
4-mer	4096	0.695	**0.626**	1.006	1.309	1.456	1.707
5-mer	8192	1.806	**1.167**	1.513	1.657	1.836	2.158
Synthetic	16384	3.671	**1.683**	1.796	1.705	2.055	2.222
Synthetic	32768	5.783	2.974	2.858	**2.626**	2.873	3.139
Synthetic	65536	7.641	5.221	4.850	**4.411**	4.952	5.713

Moreover, since SZ [39] currently does not support building the Huffman tree in parallel, we also implement a multi-thread

TABLE V: Breakdown comparison of Huffman performance on tested datasets. Gathering time is excluded.

			avg. bits	#reduce	breaking	hist. TU (GB/S)	hist. V (GB/S)	codebook TU (MS)	codebook V (MS)	encode TU (GB/S)	encode V (GB/S)	hist+enc TU (GB/S)	hist+enc V (GB/S)
cuSZ	enwik8	95 MB	5.1639	-	-	102.5	252.4	1.375	1.635	10.1	12.2	8.2	9.8
	enwik9	954 MB	5.2124	-	-	108.2	259.6	1.382	1.640	7.2	11.3	6.8	10.8
	mr	9.5 MB	4.0165	-	-	36.2	86.5	1.565	1.831	9.6	15.2	3.5	3.8
	nci	32 MB	2.7307	-	-	66.1	150.6	0.706	1.027	8.6	14.9	6.6	9.6
	Flan_1565	1.4 GB	4.1428	-	-	104.2	256.6	0.758	0.950	8.5	10.7	7.8	10.2
	Nyx-Quant	256 MB	1.0272	-	-	74.8	197.7	3.416	3.804	17.7	29.7	12.1	18.9
Ours	enwik8	95 MB	5.1639	2 (4×)	0.034915%	102.8	252.0	0.594	0.707	42.2	94.0	25.4	46.1
	enwik9	954 MB	5.2124	2 (4×)	0.021747%	108.1	276.1	0.626	0.666	49.7	94.6	34.0	70.6
	mr	9.5 MB	4.0165	2 (4×)	0.000174%	36.2	99.0	0.300	0.312	42.0	76.8	12.3	18.4
	nci	32 MB	2.7307	3 (8×)	0.152880%	56.4	169.1	0.507	0.514	63.7	154.8	20.6	36.1
	Flan_1565	1.4 GB	4.1428	2 (4×)	nearly 0%	103.5	274.7	0.314	0.327	50.0	94.9	33.5	69.5
	Nyx-Quant	256 MB	1.0272	3 (8×)	0.003277%	74.8	197.6	0.449	0.544	145.2	314.6	45.4	96.0

codebook construction using OpenMP. We evaluate its performance and compare it with SZ's serial codebook construction on our tested datasets with $1024 \sim 8192$ symbols and synthetic normally-distributed histograms[4] with 16384∼65536 symbols, as shown in Table IV. We note that in many cases, even with only one thread, its performance is better than the serial construction, as it uses internal cache-friendly arrays rather than the binary trees and priority queues used by the serial construction. We also note that on our test datasets (with codebooks in the order of 10^3), the multi-thread construction does not improve the performance because the OpenMP introduces more overhead than it reduces from multiple threads.

We find that our multi-thread CPU codebook construction needs least 32768 symbols to be able to overcome the OpenMP overhead and obtain a speedup using multiple threads. Unlike CPU, GPU parallel construction can always yield a speedup over serial construction in our tested cases. This is due to the relatively high latency and low performance of a single GPU thread. On the other hand, parallelization and synchronization are relatively low-latency on the GPU. Furthermore, since the GPU can launch vastly more threads than the CPU, it takes advantage of fine-grained parallelism in our parallel codebook construction algorithm.

2) Encoding: Without loss of generality, we evaluate our Huffman encoder on multiple datasets with various types, as shown in Table V. Most of our tested datasets have a relatively large average bitwidth (e.g. 4 or more bits vs. uncompressed 8 bits), leading to a relatively low compression ratio. According to the aforementioned r decision-making mechanism, only `nci` and `Nyx-Quant` can use $r{=}3$ (potentially $r{=}4$ for `Nyx-Quant`), making their throughputs over 100 GB/s. While compared with `Nyx-Quant`, `nci` has a relatively small data size, so it is difficult to undersaturate the memory bandwidth. Moreover, `Nyx-Quant` has much lower writing effort due to 2.66× higher compression ratio than `nci`, so its encoding throughput is high, at 314.6 GB/s.

As mentioned in §IV-C, the rigid fixed-size typed data makes it possible to meet conflict. For example, when merging `Flan_1565`, there is less than 1.4e-6% of the data that breaks the fixed size, while there is 3e-3% of the data for `Nyx-Quant`.

[4]The symbol numbers in the tested real datasets are no more than 8192, so we use synthetic data for more than 8192 symbols. Also, note that 8192 is limited by the current optimal GPU histogramming.

TABLE VI: Performance of multi-thread Huffman encoder on `Nyx-Quant`.

cores	1	2	4	8	16	32	56	64	TU	V
hist. (GB/s)	2.24	4.42	8.83	17.61	34.97	63.59	61.47	63.14	74.80	197.60
par. efficiency	1.00	0.99	0.98	0.98	0.97	0.89	0.49	0.44		
codebook (ms)					0.22				0.45	0.54
enc. (GB/s)	1.22	2.43	4.83	9.64	19.16	37.85	55.71	29.33	145.20	314.60
par. efficiency	1.00	0.99	0.99	0.99	0.98	0.97	0.81	0.37		
hist+enc (GB/s)	0.79	1.57	3.12	6.23	12.38	23.73	29.22	20.03	45.35	96.01

Our solution is to backtrace the breaking points in batch, which starts at the last iteration of `reduce`-merge and refers to the 2^r points that go beyond 32-bit limit together. The reduction is about 300 μs, including one-time read from global memory. The total number of breaking points (in percentile) are shown in the "breaking" column in Table V, which is negligible to affect the compression ratio. After we filter out the breaking data, we can use the state-of-the-art `cuSPARSE` API or perform a "device-wide" using ready tools such as `NVIDIA::cub` [29] to perform a dense-to-sparse conversion to save them. In addition, optional gathering coarse-grained chunks into an even denser format can be done, using the merged length; due to the compression ratio (CR), there is only as much as 1/CR additional data movement, which is a short-time `memcpy` to the `Reduce-Shuffle`-merge time. Compared with cuSZ's implementation, our encoder can improve the performance by $3.1\times \sim 5.0\times$ and $3.8\times \sim 6.8\times$ on RTX 5000 and V100, respectively, on the tested data.

Finally, we evaluate the performance of our implemented multi-thread encoding and our overall encoder on `Nyx-Quant`[5] with different numbers of CPU cores, as shown in Table VI. For encoding, the multi-thread version achieves a peak performance of 56 GB/s, and maintains high parallel efficiency up to 32 cores (with 56 available cores). However, this is still about $5.6\times$ lower than the performance of our fine-grained Huffman encoding on the V100 (i.e., 314.6 GB/s). For the overall encoder (including histogramming, codebook construction, and encoding), compared with the multi-thread encoder on the CPUs, our GPU version on the V100 improves the histogram-encoding performance by 3.3× (i.e., 29.22 GB/s v.s. 96.01 GB/s). This is because Huffman encoding tends to be memory-bound, and the GPU memory such as HBM2 in the V100 has a much higher bandwidth than state-of-the-art CPU memory such as DRAM4.

[5]We note that multi-thread histogramming/encoding have relatively stable throughput, thus we only evaluate on `Nyx-Quant` for demonstration purpose.

VI. Related Work

We note that several approaches [3, 43] have been proposed to accelerate Huffman decoding on GPUs, yet few studies considered optimizing Huffman encoding by fully utilizing GPU computing power. Despite the limited work on GPU-based Huffman encoding, we still search for some related works and discuss them as follows.

In general, parallel Huffman coding obtains each codeword from a lookup table (generated by a Huffman tree) and concatenates codewords together with other codewords. However, a severe performance issue arises when different threads write codewords of varying lengths, which results in warp divergence on GPU [45]. The most deviation between methods occurs in concatenating codewords. Fuentes-Alventosa et al. [12] proposed a CUDA implementation of Huffman coding with a given table of variable-length codes, reaching 20× the serial CPU encoding performance. Rahmani et al. [32] also proposed a CUDA implementation of Huffman coding based on serially constructing the Huffman codeword tree and parallel generating the bytestream, which can achieve up to 22× over the serial CPU performance by disregarding constraint on the maximum codeword length or data entropy. Lal et al. [18] proposed a Huffman-coding-based memory compression technique for GPUs (called E^2MC) based on a probability estimation of symbols. Recently, Tian et al. [41] developed a coarse-grained parallel Huffman encoder for error-bounded lossy compressor on GPUs; however, this implementation does not address the non-coalesced memory issue, reaching only 30 GB/s on V100.

VII. Conclusion and Future Work

In this work, we propose and implement an efficient Huffman encoder for NVIDIA GPU architectures. Specifically, we develop an efficient parallel codebook construction and a novel reduction based encoding scheme for GPUs. We also implement a multi-thread Huffman encoder for a fair comparison. We evaluate our encoder using six real-world datasets on NVIDIA RTX 5000 and V100 GPUs. Compared with the state-of-the-art Huffman encoder, our solution can improve the parallel encoding performance up to 5.0× on RTX 5000, 6.8× on V100, and 3.3× on CPUs. We plan to further optimize the performance in the future work. For example, we seek to (1) tune the performance for low-compression-ratio data, (2) explore more efficient gathering methods, and (3) explore how intrinsic data feature affects the compression ratio and the throughput.

Acknowledgements

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This work was also supported by the National Science Foundation under Grants CCF-1619253, OAC-2003709, OAC-2034169, and OAC-2042084. We acknowledge the Texas Advanced Computing Center for providing HPC resources that have contributed to the research results reported within this paper.

References

[1] 5 ways Facebook improved compression at scale with Zstandard. https://engineering.fb.com/core-data/zstandard/. Online. 2018.

[2] 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.

[3] Carlos A Angulo, Christian D Hernández, Gabriel Rincón, Carlos A Boada, Javier Castillo, and Carlos A Fajardo. "Accelerating Huffman Decoding of Seismic Data on GPUs". In: *2015 20th Symposium on Signal Processing, Images and Computer Vision*. IEEE. 2015, pp. 1–6.

[4] Dennis A Benson, Mark Cavanaugh, Karen Clark, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. "GenBank". In: *Nucleic acids research* 41.D1 (2012), pp. D36–D42.

[5] Blosc. https://github.com/inikep/lizard. Online. 2020.

[6] 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.

[7] Esha Choukse, Michael B Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W Keckler. "Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE. 2020, pp. 926–939.

[8] *Cooperative Groups: Flexible CUDA Thread Programming | NVIDIA Developer Blog*. https://developer.nvidia.com/blog/cooperative-groups/.

[9] L Peter Deutsch. *GZIP file format specification version 4.3*. 1996.

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

[11] Frontera supercomputer. https://www.tacc.utexas.edu/systems/frontera. Online. 2020.

[12] Antonio Fuentes-Alventosa, Juan Gómez-Luna, José Mª González-Linares, and Nicolás Guil. "CUVLE: Variable-length encoding on CUDA". In: *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*. IEEE. 2014, pp. 1–6.

[13] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides Benítez, and Nicolás Guil Mata. "An optimized approach to histogram computation on GPU". In: *Machine Vision and Applications* 24 (2012), pp. 899–908.

[14] Oded Green, Robert McColl, and David A. Bader. "GPU Merge Path: A GPU Merging Algorithm". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 331–340. ISBN: 9781450313162. DOI: 10.1145/2304576.2304621. URL: https://doi.org/10.1145/2304576.2304621.

[15] Ahsan Habib and Mohammad Shahidur Rahman. "Balancing decoding speed and memory usage for Huffman codes using quaternary tree". In: *Applied Informatics*. Vol. 4. 1. Springer. 2017, p. 5.

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

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

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

[19] Large Text Compression Benchmark. http://mattmahoney.net/dc/text.html. Online. 2020.

[20] Lawrence L Larmore and Teresa M Przytycka. "Constructing Huffman trees in parallel". In: *SIAM Journal on Computing* 24.6 (1995), pp. 1163–1169.

[21] Wang Liang. "Segmenting DNA sequence into words based on statistical language model". In: *Nature Precedings* (2012), pp. 1–1.

[22] Xin Liang, Sheng Di, Sihuan Li, Dingwen Tao, Bogdan Nicolae, Zizhong Chen, and Franck Cappello. "Significantly improving lossy compression quality based on an optimized hybrid prediction model". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. Denver, CO, USA: ACM, 2019, p. 33.

[23] 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)*. Seattle, WA, USA: IEEE, 2018, pp. 438–447.

[24] 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: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. Albuquerque, NM, USA: IEEE, 2019, pp. 1–11.

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

[26] Longhorn subsystem. https://www.tacc.utexas.edu/systems/longhorn. Online. 2020.

[27] R. L. Milidiu, E. S. Laber, and A. A. Pessoa. "A work-efficient parallel algorithm for constructing Huffman codes". In: *Proceedings DCC'99 Data Compression Conference (Cat. No. PR00096)*. 1999, pp. 277–286.

[28] Vu H Nguyen, Hien T Nguyen, Hieu N Duong, and Vaclav Snasel. "n-Gram-based text compression". In: *Computational intelligence and neuroscience* 2016 (2016).

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

[30] *NVIDIA/thrust: The C++ parallel algorithms library.* https://github.com/NVIDIA/thrust.

[31] S. Arash Ostadzadeh, B. Elahi, Zeinab Zeinalpour-tabrizi, M. Moulavi, and Koen Bertels. "A Two-phase Practical Parallel Algorithm for Construction of Huffman Codes." In: Jan. 2007, pp. 284–291.

[32] Habibelahi Rahmani, Cihan Topal, and Cuneyt Akinlar. "A parallel Huffman coder on the CUDA architecture". In: *2014 IEEE Visual Communications and Image Processing Conference*. IEEE. 2014, pp. 311–314.

[33] Eugene S Schwartz and Bruce Kallick. "Generating a canonical prefix encoding". In: *Communications of the ACM* 7.3 (1964), pp. 166–169.

[34] Scientific Data Reduction Benchmarks. https://sdrbench.github.io/. Online. 2020.

[35] Silesia Corpus. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia. Online. 2020.

[36] SuiteSparse Matrix Collection. https://sparse.tamu.edu/Janna/Flan_1565. Online. 2020.

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

[38] *SZ/sz_omp.c at master · szcompressor/SZ*. https://github.com/szcompressor/SZ/blob/master/sz/src/sz_omp.c.

[39] szcompressor.org. *szcompressor/SZ: Error-bounded Lossy Data Compressor (for floating-point/integer datasets)*. https://github.com/szcompressor/SZ.

[40] 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: *2017 IEEE International Parallel and Distributed Processing Symposium*. Orlando, FL, USA: IEEE, 2017, pp. 1129–1139.

[41] 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.

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

[43] 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.

[44] *weissenberger/gpuhd: Massively Parallel Huffman Decoding on GPUs*. https://github.com/weissenberger/gpuhd.

[45] 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.

[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.

[47] Zstd. https://github.com/facebook/zstd. Online. 2020.