# GPULZ: Optimizing LZSS Lossless Compression for Multi-byte Data on Modern GPUs

Boyuan Zhang
Indiana University
Bloomington, IN, USA
bozhan@iu.edu

Jiannan Tian
Indiana University
Bloomington, IN, USA
jti1@iu.edu

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

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

Martin Swany
Indiana University
Bloomington, IN, USA
swany@indiana.edu

Dingwen Tao*
Indiana University
Bloomington, IN, USA
ditao@iu.edu

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

## ABSTRACT

Today's graphics processing unit (GPU) applications produce vast volumes of data, which are challenging to store and transfer efficiently. Thus, data compression is becoming a critical technique to mitigate the storage burden and communication cost. LZSS is the core algorithm in many widely used compressors, such as Deflate. However, existing GPU-based LZSS compressors suffer from low throughput due to the sequential nature of the LZSS algorithm. Moreover, many GPU applications produce multi-byte data (e.g., int16/int32 index, floating-point numbers), while the current LZSS compression only takes single-byte data as input. To this end, in this work, we propose GPULZ, a highly efficient LZSS compression on modern GPUs for multi-byte data. The contribution of our work is fourfold: First, we perform an in-depth analysis of existing LZ compressors for GPUs and investigate their main issues. Then, we propose two main algorithm-level optimizations. Specifically, we (1) change prefix sum from one pass to two passes and fuse multiple kernels to reduce data movement between shared memory and global memory, and (2) optimize existing pattern-matching approach for multi-byte symbols to reduce computation complexity and explore longer repeated patterns. Third, we perform architectural performance optimizations, such as maximizing shared memory utilization by adapting data partitions to different GPU architectures. Finally, we evaluate GPULZ on six datasets of various types with NVIDIA A100 and A4000 GPUs. Results show that GPULZ achieves up to 272.1× speedup on A4000 and up to 1.4× higher compression ratio compared to state-of-the-art solutions.

## CCS CONCEPTS

• **Theory of computation** → **Massively parallel algorithms**; **Data compression**;

## KEYWORDS

Lossless compression; LZSS; GPU; performance.

*Corresponding author: Dingwen Tao.


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

ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593706

**ACM Reference Format:**
Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Martin Swany, Dingwen Tao, and Franck Cappello. 2023. GPULZ: Optimizing LZSS Lossless Compression for Multi-byte Data on Modern GPUs. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3577193.3593706

## 1 INTRODUCTION

Many applications running on high-performance parallel and distributed systems generate large amounts of data, which leads to storage bottlenecks due to limited capacity. Meanwhile, interconnect technologies in distributed systems advance relatively more slowly than computing power, causing inter-node communication and I/O bottlenecks to become a severe issue [4]. This motivates the design of software solutions to increase the interconnect bandwidth, such as communication-avoiding linear algebra [2, 14].

Data compression is a popular solution to reduce communication and I/O overheads significantly. For example, due to the high data reduction capabilities, lossy compression has recently been extensively studied to alleviate I/O bottlenecks in large-scale distributed applications such as high-performance computing (HPC) simulations. Since the saved data is often used for post-analysis and visualization, errors introduced by error-bounded lossy compression are acceptable for many applications [4, 10, 19, 20, 33, 43].

However, lossy compression may not be applicable for inter-node communication in most distributed applications since data is usually exchanged between nodes at least once per time step, resulting in an accumulation of compression errors beyond the acceptable level. This is especially important for HPC simulations where numerical stability is critical, as accumulated compression errors can affect the correctness of the results.

Unlike lossy compression, lossless compression can avoid the loss of accuracy despite the relatively low compression ratio. In practice, among many lossless compression algorithms, LZ-series lossless compression is one of the most important algorithms. It can identify repeated subsequences/patterns, thereby reducing spatial redundancy of the input sequence. Specifically, LZSS [36] is a derivative of the classical LZ77 algorithm [48] (i.e., the first LZ compression algorithm). It holds a sliding window for the input stream to search for the longest match and then encodes each match as one pointer, including its length and offset (will be discussed in §2). Input data with longer repeated subsequences are more likely to achieve higher compression ratios with LZSS. As an entropy coder, LZSS is often combined with other types of lossless coders to (e.g., with Huffman encoding as Deflate [9]) remove both spatial and frequency redundancy.

On one hand, multi-byte data such as long integers and floating-point numbers are common as input to lossless compression [26, 37, 46]. However, the classic LZSS compression only takes a single byte as the input unit, ignoring the data characteristics of different data types. Using multiple bytes as units in LZSS can improve both compression throughput (due to fewer symbols to process) and ratio (due to longer repeated patterns).

On the other hand, more and more applications are being implemented on the GPU due to its high performance and energy efficiency [12], resulting in multiple critical use cases of GPU compression. For example, GPU compression can speed up GPU-CPU data transfers [45]. It can also reduce GPU memory footprint to support larger input in deep learning [21]. However, it is challenging to parallelize LZSS on GPUs due to its strong data dependency [36]. Simply chunking data and distributing them to different GPU threads would cause warp divergence [47].

CULZSS [31] is a state-of-the-art open-source GPU LZSS compressor. It can achieve relatively higher compression throughput on the GPU than the CPU solution [13]. However, CULZSS faces several critical issues: 1 It cannot handle multi-byte data, and simply modifying its algorithm to accommodate multi-byte input may result in a significant drop in compression ratio. 2 It lacks tuning of parameters such as block size and sliding window size for different GPU architectures. 3 Its encoding process is performed on the CPU, which introduces high CPU-GPU data movement overhead.

To solve the above issues, we propose a highly optimized LZSS compression for multi-byte data on modern GPUs (called gpuLZ[1]). Specifically, we deeply analyze CULZSS and identify its performance issues. Based on these issues, we propose two main algorithm-level optimizations and a series of performance optimizations. These optimizations can improve compression throughput and ratio simultaneously. To the best of our knowledge, *this is the first work that optimizes LZSS compression for multi-byte data on GPUs.*

The main contributions of this paper are summarized as follows.

- We develop a highly efficient LZSS compression on GPUs for multi-byte data. We perform an in-depth analysis of CULZSS and investigate its main performance issues.
- We optimize the prefix sum from one pass to two passes and fuse multiple kernels (e.g., matching and local prefix sum) to reduce data movement between shared memory and global memory.
- We propose a pattern-matching method for multi-byte data, which can reduce computational complexity and explore longer repeated patterns.
- We propose a data partitioning method that can adapt to different GPU architectures to maximize shared memory utilization.
- We evaluate gpuLZ on six datasets with NVIDIA A100 and A4000 GPUs. The evaluation demonstrates that gpuLZ outperforms CULZSS by up to 272.1× in compression throughput with no degradation of compression ratio (even 20.6% improvement).

In §2, we present the background about CUDA architecture, LZSS algorithm, GPU implementations of LZSS, and their issues. In §3, we present the design of gpuLZ with our algorithm-level and architectural performance optimizations. In §4, we evaluate gpuLZ and compare it with other GPU LZ compression. In §5, we conclude the paper and discuss our future work.

[1]The code is available at https://github.com/hipdac-lab/GPULZ.

## 2 BACKGROUND AND MOTIVATION

In this section, we present the background of CUDA architecture, LZSS algorithm, and its state-of-the-art GPU implementations.

### 2.1 CUDA Architecture

CUDA is a parallel computing platform and API that allows the software to use NVIDIA GPUs for general-purpose processing. Thread is the basic programmable unit for GPU programmers to use massive numbers of CUDA cores. CUDA threads are organized into three levels, grid, block, and thread. Specifically, a group of 32 threads is called a *warp*. All threads in the same warp will execute the same instruction. However, if different threads in a warp follow different control paths, some threads are masked from performing any useful work. This situation is called *warp divergence*, which is one of the fundamental factors that limit the performance of GPUs. Multiple warps are combined to form a thread *block*, and a set of thread blocks form a thread *grid*.

Regarding the CUDA memory hierarchy, the largest and slowest memory is called the *global memory*, which is accessible by all threads. The next layer is *shared memory*, which is a fast and programmable cache. All the threads in the same thread block have access to the same shared memory. Lastly, the fastest layer is the thread-private register to each thread. To achieve good performance, CUDA programmers must effectively utilize the memory subsystem. For example, when threads in a warp request contiguous global-memory locations, these requests can be aggregated into a single transaction (called *coalesced memory access*); non-coalesced memory access will cause a significant performance slowdown.

### 2.2 LZSS

LZSS is a variant of LZ77 [48], the first algorithm in the LZ compression family. LZSS has the same fundamental idea as other LZ algorithms: search through a sliding window for the longest possible sub-sequence match and encode all identified matches. To clearly explain the LZSS algorithm, we introduce some basic concepts as follows.

- **Input stream** is the sequence of bytes to be compressed.
- **Symbol** is the single-/multi-byte unit of the input stream.
- **Look-ahead buffer** is the byte sequence from the coding position to the end of the input stream.
- **Coding position** is the byte position in the input stream currently encoded in the look-ahead buffer.
- **Sliding window** is a buffer (of size $W$), which is the number of bytes from the coding position backward. The window is empty at the beginning, then grows to size $W$ as the input stream is processed, and "slides" along with the coding position.
- **Pointer** contains two numbers: the first one is the length of the match, and the second one is the starting offset. The starting offset is the count of bytes from the coding position back to the window, and the length is the number of bytes to read forward from the starting offset.
- **Literal** represents the current byte if there is no match.

```
 0: I meant what I said            0: I meant what I said
20: and I said what I meant       20: and(11,7)(23,8)(36,5)
44:                               30:
45: From there to here            31: From there to (8,4)
64: from here to there            47: f(19,4)(18,8)(27,5)
83: I said what I meant           55: (59,19)
```

**Figure 1: An example of LZSS algorithm. The left is original data, and the right is compressed data. Two numbers in brackets denote length and offset.**

- **Flag array**'s each bit indicates whether its corresponding bytes (in compressed data) represent a pointer or a literal.

The basic steps of LZSS can be summarized as follows.

1) Set the coding position to the start of the input stream;
2) Find the longest match started from the coding position;
3) If a match is found, output the pointer $P$ and move the coding position and the sliding window $L$ bytes forward, where $L$ denotes the length of the match;
4) If no match is found, output the first byte in the look-ahead buffer and move the coding position and the sliding window one byte forward;
5) Use a flag array to record whether a match is a found;
6) If the look-ahead buffer is not empty, return to Step 2).

Figure 1 illustrates a simple example to demonstrate how LZSS works. Specifically, the original 102 bytes are compressed to 56, bytes including a flag array. A pair of numbers in brackets represents a pointer, where the first is the offset and the second is the length. If $W$ (also the maximum match length) is less than 256, both the offset and the length can be represented in one byte. This example demonstrates why LZSS is well suited for compressing data with many repeated patterns. However, it also indicates LZSS's strong sequential execution characteristics, since the current match must start from the coding position determined by the last match. Due to this strong dependency, LZSS cannot fully leverage the massive parallelism of GPU for high performance.

## 2.3 GPU LZ Compression

CULZSS [7, 31] is a state-of-the-art GPU implementation of the LZSS algorithm. It first partitions the input data into multiple chunks to increase the parallelism and then launches a matching kernel on the GPU and an encoding kernel on the CPU. Specifically, the matching kernel lets each GPU thread find the longest match for each byte of the input stream and stores all matches in the global memory, After that, all matches will be copied from the GPU to the CPU. Finally, the CPU encoding kernel will sequentially process these matches like the original LZSS. Note that not all matches will be used in the encoding: if one match covers the following matches, these overlapped matches will be skipped. Furthermore, Ozsoy *et al.* [32] improved CULZSS by overlapping the GPU and CPU computations to increase the performance. In addition, CULZSS-Bit [30] adapted CULZSS to handle bit-wise symbols.

We also note that the nvCOMP library [27] developed by NVIDIA provides a series of lossless compressors on the GPU, including LZ4. LZ4 is a fast LZ compression implementation, especially featuring fast decoding. Unlike LZSS, LZ4 does not use a flag array to indicate the match pointer; instead, it uses a fixed format with a token (including literal and match length) to save each match. However,

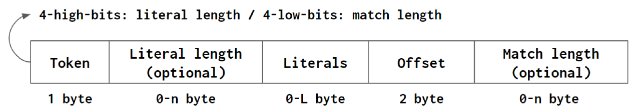



**Figure 2: LZ4 format.**

compared to LZSS's flag array, the fixed-length token may result in a lower compression ratio, especially when the length of continuous literals is relatively short. More importantly, we cannot modify the proprietary nvCOMP to accommodate multi-byte data. Thus, in this work, we focus on CULZSS as our major comparison baseline.

## 2.4 Issues of CULZSS

CULZSS faces several critical issues: 1 **No support of multi-byte data**: CULZSS treats the input as a sequence of single bytes regardless of the data type. This can lead to lower compression ratios because patterns are in multi-byte units and/or lower compression throughput due to the higher computational complexity of searching for matches in units of single bytes. 2 **Fixed data chunk size**: Data chunk size highly impacts the compression ratio and throughput, but it is a fixed value in CULZSS. Thus, it is challenging to adapt CULZSS to different GPU architectures with different shared memory sizes. 3 **Fixed sliding window size:** CULZSS uses a fixed sliding window size, which prevents a potential tradeoff between compression ratio and throughput. 4 **Under-utilization of shared memory:** CULZSS underutilizes the GPU shared memory, resulting in multiple buffer updates in shared memory. 5 **CPU encoding:** CULZSS copies matches (twice the size of the input stream) from the GPU to the CPU and performs the encoding, which causes a significant performance drop due to data copies and slow sequential CPU encoding.

# 3 OUR PROPOSED DESIGN

In this section, we first overview our GPULZ. Then, we describe our proposed algorithm-level optimizations to solve the above issues. Finally, we present the implementation details of our GPU kernels.

## 3.1 Overview of GPULZ

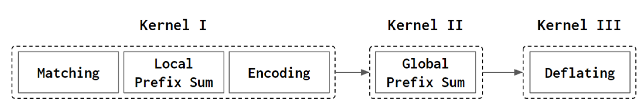



**Figure 3: Workflow of our proposed GPULZ.**

The goal of our design is to fully utilize the GPU resources for high compression throughput and maintain as high in compression ratio as the original/sequential LZSS algorithm. We illustrate our proposed GPULZ in Figure 3. Specifically, GPULZ consists of five steps: matching, local (block-level) prefix sum, encoding, global (grid-level) prefix sum, and deflating. We propose three kernels for these steps. Specifically, Kernel I is for the matching step, the local prefix sum, and encoding to generate the compressed symbols for each data chunk (§3.3.2); Kernel II is for the global prefix sum to

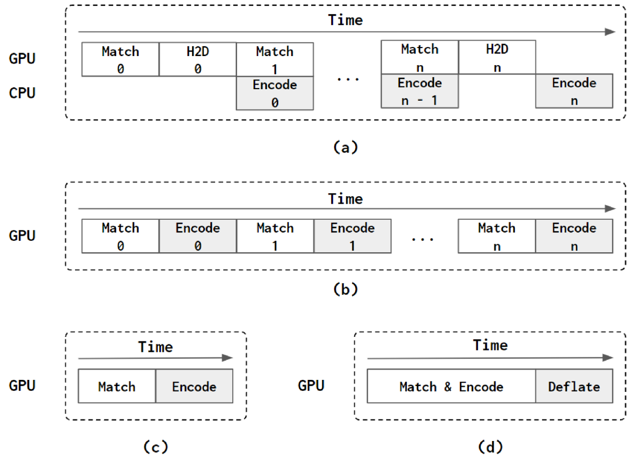



Figure 4: Three workflows of GPU LZSS.

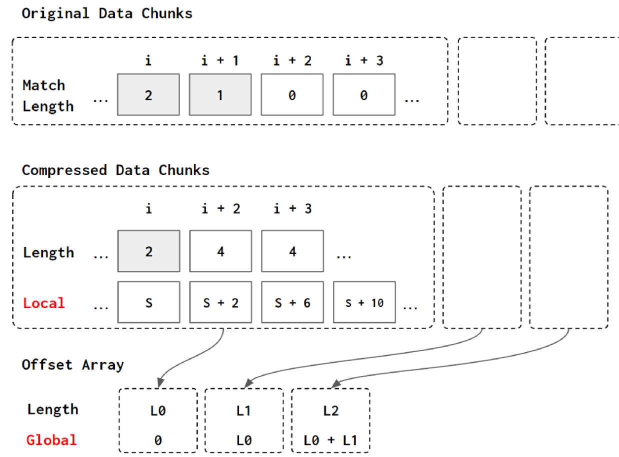



Figure 5: Our proposed two-pass prefix sum.

calculate the memory address (or write offset) of each compressed data chunk in the global memory (§3.3.3); and Kernel III is for deflating empty bytes based on the calculated offsets and writing the compressed data (§3.3.4).

Note that we use three separate kernels because a grid-level synchronization across thread blocks is needed to calculate the global memory addresses. In comparison, there is an implicit synchronization between two kernels. Moreover, although we cannot use a single kernel to handle all computations in the shared memory due to hardware constraints (§3.2.2), we propose an optimization (i.e., two-pass prefix sum with kernel fusion) that minimizes data movement between the shared and global memories and reduces the global memory footprint (§3.2.2).

For the matching step, similar to CULZSS, we also find the longest match in the sliding window for each symbol in the input stream, even though some matches will not be used in the encoding process. However, as discussed in §2.4, CULZSS's matching step does not consider multi-byte symbol, which significantly degrades the compression ratio and throughput. To solve this issue, we propose a multi-byte matching approach, which can reduce the computational complexity and find longer matches (§3.2.3). For the encoding step, as discussed in §2.4, CULZSS must copy matches from the GPU to the CPU and perform the CPU encoding sequentially, leading to low throughput. This makes CULZSS impractical for use cases where data generated on the GPU needs to be compressed. Thus, we propose a new compression workflow, including encoding and deflating (see §3.2.2 and §3.2.1, respectively) to get rid of the handling from the CPU side completely.

## 3.2 Algorithm-level Optimizations

Next, we describe our four algorithm-level optimizations in detail.

*3.2.1 Exploring Optimal Workflow.* First, we explore the optimal workflow of LZSS compression on the GPU. CULZSS uses the CPU to encode/compress the matches found by the GPU, as shown in Figure 4 (a). The CPU encoding kernel and the GPU matching kernel are executed asynchronously to maximize overlapping. However, this workflow makes the encoding process difficult to parallelize, and the GPU-to-CPU data movement is time-consuming. To solve this issue, we propose to perform the encoding on the GPU. One straightforward solution is to replace the CPU encoding directly with a GPU kernel without changing the workflow, as illustrated in Figure 4 (b). However, this workflow is still not optimal because every encoding kernel depends on the last matching kernel. This dependency causes only one kernel to be executed at one time (i.e., sequential execution), which brings the GPU resources starvation problem, especially for small data chunks. Moreover, the multiple kernel launches further increase the time overhead.

To address this issue, we propose our second workflow, as illustrated in Figure 4 (c). In this design, we perform the matching and encoding steps in two separate GPU kernels. While the dependency between these two kernels still exists, the size of the data processed is changed from one data chunk to the entire input stream, which solves the starvation problem. However, this design requires a large global memory space to store the intermediate data (i.e., high memory footprint) and causes a large amount of data to be moved between global memory and shared memory. To this end, we propose our third workflow that performs the matching and encoding steps in the same kernel, as shown in Figure 4 (d). One major issue with fusing the matching and encoding kernels is that the output includes empty bytes. Thus, we need to add another kernel to eliminate these empty bytes (called "deflating kernel"). This workflow is non-trivial because it is only feasible with our proposed two-pass prefix sum (detailed in §3.2.2).

*3.2.2 Two-pass Prefix Sum with Kernel Fusion.* Fine-grained parallelization of LZSS on GPU is more challenging than coarse-grained parallelization on CPU because GPU thread blocks do not communicate with each other while the kernel function is running, causing the memory offset of each compressed data chunk to be unknown. To solve this issue, we first locally calculate the size of each symbol after compression (could be either a pointer or a literal), then globally synchronize these sizes, and finally calculate the global offset for each symbol based on a prefix sum. One approach to achieve global synchronization in a CUDA kernel is to use "cooperative groups" [28]. However, the cooperative groups API has a limited number of threads (e.g., 1,280 threads on A4000), smaller than we need, which is typically 5,000 threads. Thus, we need to divide the kernel into two with an implicit device-level synchronization involved. However, this design requires moving compressed data back to the global buffer, incurring multiple data movements between shared memory and global memory.

To solve this issue, we propose an optimization called two-pass prefix sum. It includes both a local prefix sum and a global prefix sum. The design is shown in Figure 5. Specifically, the local prefix sum calculates the offset for each compressed symbol within each data chunk/thread block. Here we adopt an optimized two-sweep prefix-sum algorithm that fits GPU well [3]. It includes up-sweep and down-sweep processes, detailed in §3.3.2.

After we get the compressed size of each data chunk from the local prefix sum, we can calculate the offset of each compressed chunk through a global prefix sum across data chunks. Compared with the single-pass prefix sum, our proposed two-pass prefix sum only needs to store the size of each compressed data chunk instead of the size of each compressed symbol, which significantly reduces the amount of data written to and read from the global memory (e.g., by at least $C$ times, where $C$ is the number of symbols per data chunk). Moreover, our two-pass prefix sum can also reduce space complexity and the global memory footprint. Note that to avoid moving the match result back and forth between the shared and global memories for the local prefix sum, we propose to fuse the local prefix-sum computation into the matching kernel. Thus, we can perform the local prefix-sum computation directly on the matching result stored in the shared memory. This can also reduce the global memory footprint.

After the local prefix sum, each GPU thread encodes symbols based on the calculated local offsets and the found matches, similar to the CPU sequential encoding in LZSS, as mentioned in §2.2. Note that since this encoding is performed at the thread-block level, no grid-level synchronization is needed. As a result, the encoding can be further fused with the matching and local prefix sum steps to form Kernel I. It is also worth noting that compared with CULZSS, our encoding enables massive GPU threads, which maximizes the parallelism and encoding throughput.

*3.2.3 Multi-byte Matching Approach.* CULZSS only performs the matching step on a single-byte basis, which leads to a decrease in compression throughput and a potential loss of compression ratio. This is because, for datasets based on multi-byte symbols, single-byte matching would lose the characteristics of a specific data structure. To this end, we propose a novel multi-byte matching approach that finds matches based on symbols instead of bytes. This strategy has two advantages: 1 Searching for matches based on symbols is less expensive than searching for matches based on bytes since there are far fewer symbols than bytes, which increases compression throughput. 2 It can bring potentially higher compression ratios because each match can contain more bytes. We use $S$ to denote the symbol length in the following discussion.

However, the potential gain in compression ratio is not guaranteed, especially when the match length is generally short. Therefore, to maximize the chance of increasing the compression ratio with our multi-byte matching approach, we propose to adaptively select the symbol length and increase the sliding window size. For example, assuming that the input data type is int32, by default, gpuLZ adopt the 4-byte symbol length and the sliding window size of 128. Our approach is to adapt the symbol length (ranging from 1 to 4) and the sliding window size (ranging from 32 to 255[2]) to achieve the best trade-off between the compression ratio and the throughput.

After studying the impacts of symbol length $S$ and sliding window size $W$ on various datasets (detailed in §4.2), we propose a lightweight parameter selection approach. Specifically, assuming the datasets contain multiple fields that are the input to gpuLZ at one time, we monitor the average compression ratio with the multi-byte matching strategy (default). 1 When the average compression ratio is relatively low (for instance, lower than 1.5), we switch back to single-byte matching, considering that the multi-byte matching is not effective under low compression ratio circumstances (will be illustrated in §4.2). This is because the multi-byte matching results in a smaller number of repeated patterns and ignores byte-level repeated patterns. 2 When the average compression ratio is high, we keep using multi-byte matching, considering that multi-byte matching has a significant improvement in compression ratio over single-byte matching. On the other hand, for $W$, we enlarge it to $S$ times when we use the multi-byte matching strategy since the multi-byte matching can bring a speedup of about $S$ times over the single-byte matching, which will offset the higher time complexity brought by a larger sliding window size.

In addition, we provide another option that allows users to set different sliding window sizes, e.g., 32/64/128/255 as level 1-4. A higher level will bring a higher compression ratio but lower throughput. The user can decide the level based on their needs. For example, if compression throughput is a priority, users should select level 1; if the compression ratio is a priority, users should select level 4.

## 3.3 Details of gpuLZ and Its Implementation

Finally, we introduce our architectural performance optimizations with some implementation details. We describe these details following our compression workflow. We first introduce the data partition and then describe the kernel details.

*3.3.1 Data Partition.* First, we divide the input data into multiple blocks and then divide each block into multiple chunks. We launch a GPU kernel for each data block and map each data chunk to one GPU thread block in the kernel. Our data partition strategy is illustrated in Figure 6. The reasons for performing a two-level data partition are as follows: 1 *Data block*: GPUs have limited memory capacity (e.g., 16 GBs for an NVIDIA A4000 GPU), so partitioning data into blocks allows the GPU to process datasets that are larger than its memory space. 2 *Data chunk*: As aforementioned, the encoding step requires iterating over the found matches and encoding the matches that are not covered by previous matches, introducing data dependencies and sequential execution. Thus, data partitioning can enforce matches not across different chunks, increasing the encoding parallelism.

*Data block size:* CULZSS divides the input data into many small blocks (e.g., 1 MB) such that it overlaps CPU encoding with GPU matching as much as possible. However, this very small block significantly limits the GPU resource utilization and overall computational efficiency. In contrast, since our design does not involve the CPU for encoding, we can use a relatively large block size to

[2]Note that we set the maximum $W$ to 255 because we only use one byte to save the sliding window size and reserve 0 for no match found.

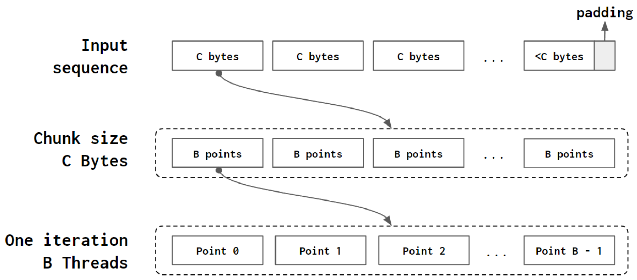



**Figure 6: Data partition strategy**

increase GPU efficiency. Therefore, we choose the block size of 30% of the global memory (e.g., 12 of 40 GB for NVIDIA A100 GPU).

*Data chunk size:* On the one hand, our data partition results in lower compression ratios since we ensure matches cannot span chunks; in other words, larger chunks have higher compression ratios. On the other hand, the chunk size needs to satisfy the GPU hardware constraint as each chunk is stored temporarily in the GPU shared memory. Note that the shared memory is part of the GPU's L1 cache, so the more shared memory is used the less L1 cache is left. The trade-off between shared memory and L1 cache is discussed in detail in §4.2. In addition, each thread in a thread block processes multiple symbols in a data chunk. We use $C$ to denote the data chunk size in the following discussion.

### 3.3.2 *Kernel I.*

Next, we describe Kernel I and our optimization in warp divergence. We also present its pseudocode in Listing 1.

**Listing 1: Proposed Kernel I**

```
1  input: original data
2  output: compressed data, flag array, offset arrays
3
4  // find matches
5  for iteration in chunkSize / blockDim.x:
6      tid = threadIdx.x + iteration * windowSize
7      while windowPointer < bufferStart && bufferPointer
           < blockSize
8          if buffer[bufferPointer] == buffer[
               windowPointer]:
9              if offset[bufferPointer] == 0:
10                 offset = bufferPointer - windowPointer
11             len++
12             bufferPointer++
13         else:
14             if len > maxLen:
15                 maxLen = len
16                 maxOffset = offset
17             len = 0
18             offset = 0
19             bufferPointer = bufferStart
20         windowPointer++
21     writeToSharedMem(lengthArr, offsetArr)
22     initializeToZero(prefixArr)
23
24 // synchronize threads
25 __syncthreads()
26
27 // find encoding information
28 __shared__ uint8_t byteFlagArr[(dataChunkSize / 8)]
29
30 if threadidx.x == 0:
31     while encodeIndex < blockSize:
32         if lengthBuffer[encodeIndex] < minEncodeLength:
33             prefixBuffer[encodeIndex] = sizeof(dataType)
34             encodeIndex++
35         else:
36             prefixBuffer[encodeIndex] = 2
37             encodeIndex += lengthBuffer[encodeIndex]
38     generateFlagArr(byteFlagArr)
39
40 // local prefix sum
41 localPrefixUpSweep(prefixArr)
42 saveTheCompressedChunkSize()
43 localPrefixDownSweep(prefixArr)
44
45 // compress data
46 for iteration in chunkSize / blockDim.x:
47     encodeBasedOnlocalPrefix()
48
49 // copy flag array back to global mem
50 writeBackToGlobal(byteFlagArr)
```

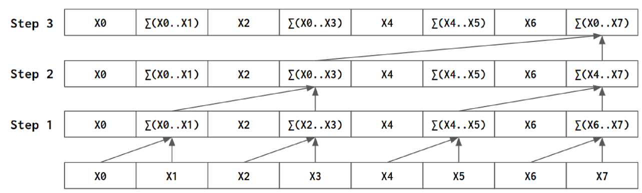
At the beginning of Kernel I, we load the input stream into the shared memory. Different from CULZSS, which uses two arrays in the shared memory to store the input stream and the sliding window, we integrate them into one array and use pointers to indicate the start positions of the input stream and sliding window. This design saves the context switch time for updating arrays and fully utilizes the shared memory to accelerate the matching step.

Then, we find matches for every symbol in the designated data chunk. In the sequential LZSS, the matching process is highly time-consuming. In the best case, it takes O(n) (when no match is found), while in the worst case, it takes O($n^2$) time complexity (when the sliding window and the look-ahead buffer contain the same symbol). This unstable time complexity would cause a severe divergence among different GPU threads. To solve this issue, we use an optimized matching method with a stable time complexity to reduce the divergence among threads. Our method is described as follows.

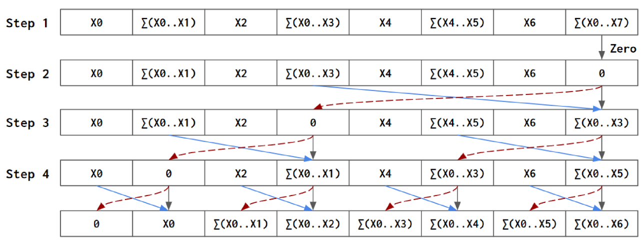
1) We use a search pointer to the start of the window and a position pointer to the coding position (Lines 12–25).
2) We move the search pointer until it reaches the same value as the position pointer points to, and then move both pointers until they point to different values (Lines 13-17).
3) We record the length and offset of the current match only if it is longer than the previously recorded match (Lines 18–24).
4) We let the position pointer point to the coding position again, advance the search pointer to the next location, and repeat step 2) until the sliding window is iterated all over or the look-ahead buffer is empty (Lines 12, 22–24).

Although this method cannot guarantee an optimal output (i.e., always finding the longest matches), it gives a sufficient result (will be shown in the evaluation) with a very stable time complexity of O($n$). Moreover, unlike the traditional LZSS, the maximum encoding length in our method does not exceed the offset. Both aspects can minimize the possibility of warp divergence.

After that, we encode the matches (Lines 35-43). Specifically, we use one thread per thread block to calculate the compressed size and produce the flag array (due to LZSS's sequential nature). Specifically, if a match is found for the current symbol, it takes two bytes (i.e., one for length and one for offset) to encode the match and appends one bit "1" to the flag array to denote "match" and then skips the symbols that the match covers (Lines 40-42). If no match




**(a) Up-sweep**




**(b) Down-sweep**

**Figure 7: Two-sweep algorithm**

is found, it saves the original symbol (i.e., the number of bytes for the input data type) and appends one bit "0" to the flag array to denote "no match" (Lines 37-39). Note that before the local prefix sum to calculate the memory offsets, we allocate an array in the shared memory to save the size of each compressed symbol (Line 7) and initialize the array to all zeros (Line 27). Thus, we can skip the symbols covered by a match to further improve performance.

Finally, we use the local prefix sum (described in §3.2.2) to calculate the memory offset of each symbol within its data chunk and write the compressed data chunks and their sizes to the global memory for deflating. We use a two-sweep method [3] to implement our local prefix sum, as illustrated in Figure 7. Specifically, in the up-sweep phase, we calculate the summation of two elements with a distance of $2^{\text{step}-1}$ (Figure 7a). Then, we can get the summation of all elements stored in the last position. After that, we save this back to the global memory for the following global prefix sum and reinitialize it with 0. In the down-seep phase, we copy each element to a new position and calculate the summation of two elements (Figure 7b). Finally, we get the prefix sum of the whole array.

#### 3.3.3 *Kernel II.*

Next, we present Kernel II and show its pseudocode in Listing 2. Specifically, we perform the global prefix sum (discussed in §3.2.2) in Kernel II. Since prefix sum has been implemented and highly optimized in the CUB library [6], we directly call it to achieve high performance. Note that Kernel II launches the CUB's prefix-sum kernel twice, because we not only need to calculate the offset of the compressed data (Line 5) but also need to calculate the offset of the flag array (Line 8).

**Listing 2: Proposed Kernel II**

```
1 input: compressed size, flag array size
2 output: compressed offset, flag array offset
3
4 // calcualte the offset for compressed size
5 cub::DeviceScan::ExclusiveSum(compressedSize, prefix)
6
7 // calcualte the offset for flagarray
8 cub::DeviceScan::ExclusiveSum(flagSize, flagPrefix)
```

#### 3.3.4 *Kernel III.*

Finally, we implement the deflating process in Kernel III. The pseudocode of Kernel III is presented in Listing 3. We use the same granularity as the matching step to fully utilize the parallelism of the GPU. Specifically, we calculate the sizes of the flag array and the compressed data chunk (Lines 5-6) and then write the flag array (Lines 9-11) and the compressed data (Lines 14-16) based on the memory offsets.

**Listing 3: Proposed Kernel III**

```
1  input: compressedData, cOffset, flagArray, fOffset
2  output: compressedOut, flagArrayOut
3
4  int tid = threadIdx.x
5  flagArrSize = fOffset[Index + 1] - fOffset[Index]
6  compressedSize = cOffset[Index + 1] - cOffset[Index]
7
8  // write back flag array
9  while tid < flagArrSize:
10     write(flagArray, fOffset[index], tid)
11     tid += blockDim.x
12
13 // write back compressed data
14 while tid < compressedSize:
15     write(compressedData, cOffset[index], tid)
16     tid += blockDim.x
```

## 4 PERFORMANCE EVALUATION

In this section, we present our evaluation of GPULZ on six representative multi-byte datasets and its comparison with state-of-the-art LZ GPU solutions, i.e., CULZSS and nvCOMP's LZ4.

### 4.1 Experimental Setup

*Platforms.* We use two platforms in our evaluation: 1 One node from the Big Red 200 supercomputer [1], equipped with two 64-core AMD EPYC 7742 CPUs @2.25GHz and four NVIDIA Ampere A100 GPUs (108 SMs, 40GB), running CentOS 7.4 and CUDA 11.4.120. 2 An in-house workstation equipped with one 24-core Intel Xeon W-2265 CPU @3.50GHz and two NVIDIA GTX A4000 GPUs (40 SMs, 16 GB), running Ubuntu 20.04.5 and CUDA 11.7.99.

*Datasets.* We conduct our evaluation using six representative multi-byte datasets from two benchmarks, i.e., TPC-H benchmark [42] and Scientific Data Reduction Benchmarks (SDRBench) [35]. Specifically, TPC-H benchmark is a suite of business-oriented ad-hoc queries and concurrent data modifications. It includes one `int32` integer dataset (i.e., tpch-int32) and one utf-8 string dataset (i.e., tpch-string). SDRBench includes three uint16 datasets (i.e., hurr-quant, hacc-quant, nyx-quant) and one float32 dataset (i.e., rtm). Note that the three uint16 datasets are intermediate data [3] generated from three real-world HPC simulation datasets, i.e., HACC (cosmology particle simulation) [15], Hurricane (ISABEL weather simulation) [17], and Nyx (cosmology simulation) [29], which have been widely used in previous studies on scientific data compression [24, 25, 34, 39–41, 44]; the `float32` rtm dataset is from a seismic imaging application for petroleum exploration [18, 22].

[3]The intermediate data is the quantization code generated by cuSZ [8] under the relative error bound of 1e-3, and is stored in uint16 format.

**Table 1: Compression ratio of GPULZ. Note that some fields are noted as "n/a" due to out of the limited shared memory.**

| | | chunk size: 2048 | | | chunk size: 4096 | | | chunk size: 8192 | | | chunk size: 16384 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| window size ↓ | | 1 byte | 2 bytes | 4 bytes | 1 byte | 2 bytes | 4 bytes | 1 byte | 2 bytes | 4 bytes | 1 byte | 2 bytes | 4 bytes |
| hurr | 32 | 3.14 | 3.77 | 3.58 | 3.18 | 3.84 | 3.66 | n/a | 3.88 | 3.70 | n/a | n/a | 3.72 |
| quant | 64 | 3.79 | 4.39 | 4.05 | 3.86 | 4.50 | 4.18 | n/a | 4.56 | 4.25 | n/a | n/a | 4.28 |
| | 128 | 4.39 | 4.91 | 4.44 | 4.51 | 5.09 | 4.64 | n/a | 5.18 | 4.75 | n/a | n/a | 4.81 |
| | 255 | 4.89 | 5.32 | 4.78 | 5.07 | 5.59 | 5.15 | n/a | 5.73 | 5.36 | n/a | n/a | 5.47 |
| hacc | 32 | 1.55 | 1.67 | 1.59 | 1.55 | 1.68 | 1.60 | n/a | 1.68 | 1.61 | n/a | n/a | 1.61 |
| quant | 64 | 1.71 | 1.82 | 1.71 | 1.72 | 1.84 | 1.73 | n/a | 1.85 | 1.74 | n/a | n/a | 1.75 |
| | 128 | 1.87 | 1.97 | 1.83 | 1.88 | 2.00 | 1.86 | n/a | 2.02 | 1.88 | n/a | n/a | 1.89 |
| | 255 | 2.01 | 2.12 | 1.92 | 2.03 | 2.18 | 1.99 | n/a | 2.20 | 2.03 | n/a | n/a | 2.05 |
| nyx | 32 | 3.97 | 5.07 | 4.80 | 4.04 | 5.20 | 4.95 | n/a | 5.27 | 5.02 | n/a | n/a | 5.06 |
| quant | 64 | 5.06 | 6.18 | 5.73 | 5.19 | 6.42 | 6.00 | n/a | 6.54 | 6.14 | n/a | n/a | 6.21 |
| | 128 | 6.14 | 7.19 | 6.52 | 6.36 | 7.57 | 6.99 | n/a | 7.79 | 7.25 | n/a | n/a | 7.38 |
| | 255 | 7.08 | 8.03 | 7.11 | 7.46 | 8.65 | 7.94 | n/a | 9.01 | 8.42 | n/a | n/a | 8.64 |
| tpch | 32 | 1.31 | 1.25 | 1.29 | 1.32 | 1.26 | 1.30 | n/a | 1.26 | 1.30 | n/a | n/a | 1.30 |
| int32 | 64 | 1.37 | 1.30 | 1.34 | 1.38 | 1.31 | 1.35 | n/a | 1.31 | 1.35 | n/a | n/a | 1.36 |
| | 128 | 1.43 | 1.34 | 1.38 | 1.44 | 1.35 | 1.39 | n/a | 1.36 | 1.40 | n/a | n/a | 1.41 |
| | 255 | 1.50 | 1.38 | 1.41 | 1.51 | 1.39 | 1.43 | n/a | 1.40 | 1.44 | n/a | n/a | 1.45 |
| tpch | 32 | 1.55 | 1.58 | 1.46 | 1.56 | 1.59 | 1.47 | n/a | 1.60 | 1.48 | n/a | n/a | 1.48 |
| string | 64 | 2.02 | 1.96 | 1.72 | 2.04 | 1.99 | 1.76 | n/a | 2.01 | 1.78 | n/a | n/a | 1.79 |
| | 128 | 2.57 | 2.43 | 2.03 | 2.62 | 2.50 | 2.12 | n/a | 2.54 | 2.17 | n/a | n/a | 2.20 |
| | 255 | 3.08 | 2.84 | 2.27 | 3.19 | 3.00 | 2.47 | n/a | 3.09 | 2.58 | n/a | n/a | 2.64 |
| rtm | 32 | 2.45 | 2.72 | 2.88 | 2.47 | 2.75 | 2.91 | n/a | 2.77 | 2.93 | n/a | n/a | 2.94 |
| float32 | 64 | 2.59 | 2.80 | 2.92 | 2.61 | 2.83 | 2.96 | n/a | 2.85 | 2.98 | n/a | n/a | 2.99 |
| | 128 | 2.66 | 2.84 | 2.94 | 2.69 | 2.88 | 2.99 | n/a | 2.89 | 3.01 | n/a | n/a | 3.02 |
| | 255 | 2.69 | 2.85 | 2.97 | 2.72 | 2.90 | 3.02 | n/a | 2.92 | 3.05 | n/a | n/a | 3.07 |

*Baselines.* We compare GPULZwith two baselines: 1 *CULZSS*: CULZSS is the state-of-the-art GPU implementation (open-source) [7] of LZSS, but it uses the GPU to find matches and the CPU to encode matches. 2 *nvCOMP's LZ4*: LZ4 is similar to LZSS but uses a particular data format to achieve portability. We use the state-of-the-art GPU implementation (closed-source) of LZ4 from nvCOMP [27]. We use the latest nvCOMP 2.6.0.

*Evaluation metrics.* We focus on evaluating and analyzing GPU-based LZ compressors on two main metrics. 1 *Compression ratio* is one of the most commonly used metrics in compression research. It can be calculated as the ratio of the original data size and reconstructed data size. Higher compression ratios mean denser information aggregation against the original data and faster data transfer. 2 *Compression throughput* is the primary consideration when using a GPU-based lossy compressor instead of a CPU-based one. It can be calculated as the ratio of original data size to compression/decompression time. Higher throughput means faster compression and more significant benefits of using compression.

## 4.2 Impacts of Parameters $C$, $W$, and $S$

First, we evaluate the impacts of parameters $C$, $W$, and $S$. We conduct the experiments on both the A100 and A4000 platforms. The compression ratio is shown in Table 1, and the compression throughput is shown in Table 2. Specifically, we choose the data chunk sizes (i.e., $C$) of 2048, 4096, 8192, and 16,384. The data chunk size directly decides the shared memory size we utilize in our design. Because the shared memory is part of the L1 cache. As a result, we can observe the impact of the trade-off between shared memory and L1 cache on the overall throughput. Note that some fields in the table are empty because of the limited shared memory. The sliding window size $W$ will directly decide the time complexity. The longer the sliding window is, the higher the time complexity will be. It will also potentially increase the compression ratio. Moreover, we introduce multi-byte symbols into the LZSS algorithm to explore the potential compression ratio and throughput gains. To this end, we select three symbol lengths (i.e., $S$): 1, 2, and 4 bytes.

First, we focus on the impact of $C$. As mentioned before, we partition the data into chunks to allow LZSS to execute in parallel. However, due to the independence of each data chunk, the compression ratio would drop slightly because the match does not span the boundaries of data chunks, leading to the limited match length. The evaluation result also proves this, as illustrated in Table 1. The compression ratio increases as the data chunk size increases in all test cases. The average improvement is 1.02×. However, as the data chunk size increases, the compression throughput decreases in almost all test cases. This proves that a larger L1 cache is better for compression throughput than utilization of shared memory, at least in the range of feasible data chunk sizes of GPULZ. With smaller data chunk size, the compression throughput is improved by 1.33× on average. Note that the compression throughput drops significantly with larger data chunk sizes. For example, on the 4-byte nyx-quantization dataset, the compression throughput drops from 19.05 to 18.76 when the data chunk size changes from 2048 to 4096. At the same time, it drops from 14.67 to 8.36 when the data chunk size changes from 8192 to 16,384. This is because when the data chunk size is 16,384, the shared memory size is close to the hardware's limit, resulting in a fairly small L1 cache size and further impacting the overall throughput. This phenomenon is more obvious when the data chunk size is bigger. Note that A100 has a higher speedup than A4000 when the block size is large because A100 has larger L1 cache (192 KB/SM) than A4000 (128 KB/SM).

Next, we explore the impact of $W$. On the one hand, as analyzed before, a larger sliding window brings a potentially longer match, increasing the compression ratio. Table 1 shows that the ratio of compression ratio to the sliding window size is near linearly in

**Table 2: Compression throughput of GPULZ on both A100 (blue) and A4000 (gray) GPUs. The red bars show the performance gain when scaling from A4000 to A100.**

| | | chunk size: 2048 | | | chunk size: 4096 | | | chunk size: 8192 | | | chunk size: 16384 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | window size ↓ | 1 byte | 2 bytes | 4 bytes | 1 byte | 2 bytes | 4 bytes | 1 byte | 2 bytes | 4 bytes | 1 byte | 2 bytes | 4 bytes |
| hurr quant | 32 | 8.1 4.9 (1.7×) | 14.9 9.6 (1.6×) | 29.0 18.1 (1.6×) | 6.9 3.1 (2.2×) | 14.8 8.4 (1.8×) | 28.0 17.4 (1.6×) | n/a | 11.3 4.4 (2.6×) | 26.6 13.3 (2.0×) | n/a | n/a | 16.0 7.6 (2.1×) |
| | 64 | 4.6 2.9 (1.6×) | 8.9 5.6 (1.6×) | 17.5 11.2 (1.6×) | 4.5 2.2 (2.0×) | 8.6 5.3 (1.6×) | 17.2 11.0 (1.6×) | n/a | 7.4 3.1 (2.4×) | 16.6 9.2 (1.8×) | n/a | n/a | 11.8 5.6 (2.1×) |
| | 128 | 2.5 1.6 (1.6×) | 4.9 3.1 (1.6×) | 11.0 6.7 (1.6×) | 2.4 1.3 (1.8×) | 4.7 2.9 (1.6×) | 10.1 6.3 (1.6×) | n/a | 4.3 2.0 (2.2×) | 9.5 5.7 (1.7×) | n/a | n/a | 7.4 3.7 (2.0×) |
| | 255 | 1.4 0.9 (1.6×) | 2.8 1.8 (1.6×) | 7.0 4.4 (1.6×) | 1.3 0.8 (1.7×) | 2.6 1.6 (1.6×) | 5.7 3.6 (1.6×) | n/a | 2.3 1.1 (2.1×) | 5.3 3.3 (1.6×) | n/a | n/a | 4.3 2.3 (1.9×) |
| hacc quant | 32 | 7.4 4.2 (1.8×) | 13.8 8.5 (1.6×) | 29.0 18.1 (1.6×) | 5.8 2.6 (2.2×) | 13.1 7.5 (1.7×) | 27.5 15.5 (1.8×) | n/a | 9.3 3.4 (2.7×) | 24.6 10.6 (2.3×) | n/a | n/a | 14.5 6.1 (2.4×) |
| | 64 | 4.5 2.8 (1.6×) | 8.2 5.3 (1.5×) | 19.2 11.4 (1.7×) | 4.1 2.1 (2.0×) | 8.2 5.1 (1.6×) | 19.1 11.4 (1.7×) | n/a | 6.6 2.7 (2.4×) | 17.4 8.4 (2.1×) | n/a | n/a | 11.1 5.3 (2.1×) |
| | 128 | 2.6 1.7 (1.5×) | 4.8 3.1 (1.6×) | 12.4 6.3 (2.0×) | 2.6 1.4 (1.9×) | 4.7 3.0 (1.5×) | 11.1 6.7 (1.6×) | n/a | 4.2 2.0 (2.1×) | 11.1 6.0 (1.8×) | n/a | n/a | 7.9 3.6 (2.2×) |
| | 255 | 1.5 1.0 (1.6×) | 2.7 1.8 (1.5×) | 7.4 4.4 (1.7×) | 1.5 0.8 (1.8×) | 2.7 1.7 (1.6×) | 6.7 3.9 (1.7×) | n/a | 2.5 1.2 (2.1×) | 6.0 3.5 (1.7×) | n/a | n/a | 4.9 2.5 (2.0×) |
| nyx quant | 32 | 9.5 6.0 (1.6×) | 15.7 10.1 (1.5×) | 30.1 19.1 (1.6×) | 7.5 4.0 (1.9×) | 15.8 9.1 (1.7×) | 30.3 18.8 (1.6×) | n/a | 12.4 5.6 (2.2×) | 29.2 14.7 (2.0×) | n/a | n/a | 18.1 8.4 (2.2×) |
| | 64 | 5.7 3.6 (1.6×) | 9.4 6.2 (1.5×) | 19.8 11.6 (1.7×) | 5.4 2.8 (1.9×) | 9.3 6.2 (1.5×) | 18.0 11.4 (1.6×) | n/a | 8.1 3.8 (2.2×) | 17.9 10.8 (1.7×) | n/a | n/a | 12.9 6.3 (2.0×) |
| | 128 | 3.1 1.9 (1.6×) | 5.5 3.6 (1.5×) | 11.3 7.1 (1.6×) | 3.1 1.7 (1.9×) | 5.9 3.4 (1.7×) | 10.3 6.8 (1.5×) | n/a | 5.0 2.5 (2.0×) | 10.2 6.5 (1.6×) | n/a | n/a | 8.7 4.6 (1.9×) |
| | 255 | 1.8 1.0 (1.7×) | 3.6 2.1 (1.7×) | 6.9 4.9 (1.4×) | 1.7 0.9 (1.8×) | 3.2 1.9 (1.7×) | 6.6 4.1 (1.6×) | n/a | 3.1 1.4 (2.2×) | 6.3 3.9 (1.6×) | n/a | n/a | 5.3 2.8 (1.9×) |
| tpch int32 | 32 | 7.1 3.9 (1.8×) | 12.1 8.3 (1.5×) | 25.4 14.9 (1.7×) | 5.2 2.3 (2.3×) | 11.7 6.2 (1.9×) | 21.4 13.5 (1.6×) | n/a | 7.7 3.1 (2.5×) | 19.4 9.3 (2.1×) | n/a | n/a | 10.5 5.0 (2.1×) |
| | 64 | 4.4 2.7 (1.6×) | 7.9 5.1 (1.5×) | 16.3 10.2 (1.6×) | 3.8 1.7 (2.2×) | 7.5 4.5 (1.7×) | 14.6 9.8 (1.5×) | n/a | 5.9 2.4 (2.5×) | 14.2 7.1 (2.0×) | n/a | n/a | 8.2 4.2 (2.0×) |
| | 128 | 2.4 1.6 (1.5×) | 4.8 3.0 (1.6×) | 10.2 6.3 (1.6×) | 2.2 1.1 (2.0×) | 4.5 2.8 (1.6×) | 9.2 5.6 (1.7×) | n/a | 3.8 1.7 (2.3×) | 8.4 5.0 (1.7×) | n/a | n/a | 6.4 3.1 (2.1×) |
| | 255 | 1.3 0.9 (1.6×) | 2.8 1.7 (1.6×) | 6.7 4.0 (1.7×) | 1.2 0.7 (1.8×) | 2.4 1.6 (1.5×) | 5.8 3.5 (1.7×) | n/a | 2.1 1.0 (2.1×) | 5.0 3.1 (1.6×) | n/a | n/a | 3.8 2.0 (1.9×) |
| tpch string | 32 | 7.1 4.2 (1.7×) | 12.5 8.0 (1.6×) | 22.9 13.8 (1.7×) | 5.3 2.5 (2.2×) | 11.4 6.2 (1.8×) | 21.1 12.6 (1.7×) | n/a | 8.0 3.5 (2.3×) | 19.0 8.4 (2.3×) | n/a | n/a | 10.0 4.8 (2.1×) |
| | 64 | 4.7 3.1 (1.5×) | 8.4 5.2 (1.6×) | 15.2 9.5 (1.6×) | 4.2 2.0 (2.1×) | 7.6 5.1 (1.5×) | 15.6 9.2 (1.7×) | n/a | 6.3 2.9 (2.2×) | 14.0 6.8 (2.0×) | n/a | n/a | 8.2 4.1 (2.0×) |
| | 128 | 2.4 1.7 (1.4×) | 4.8 3.3 (1.5×) | 10.7 6.0 (1.8×) | 2.6 1.3 (2.0×) | 4.7 3.1 (1.5×) | 9.4 5.7 (1.7×) | n/a | 4.0 2.0 (2.0×) | 8.2 4.8 (1.7×) | n/a | n/a | 5.7 3.5 (1.6×) |
| | 255 | 1.4 0.9 (1.5×) | 2.6 1.8 (1.4×) | 6.9 3.7 (1.8×) | 1.4 0.8 (1.8×) | 2.6 1.7 (1.5×) | 6.1 3.3 (1.8×) | n/a | 2.3 1.2 (1.9×) | 4.7 3.3 (1.4×) | n/a | n/a | 3.7 2.3 (1.6×) |
| rtm float32 | 32 | 7.3 4.2 (1.7×) | 14.3 9.0 (1.6×) | 28.4 17.6 (1.6×) | 6.1 2.8 (2.2×) | 13.9 7.5 (1.8×) | 28.6 17.2 (1.7×) | n/a | 11.2 3.9 (2.9×) | 26.6 13.1 (2.0×) | n/a | n/a | 16.2 7.4 (2.2×) |
| | 64 | 4.5 2.9 (1.6×) | 9.3 5.5 (1.7×) | 17.7 11.2 (1.6×) | 3.9 2.2 (1.8×) | 8.3 5.1 (1.6×) | 17.4 10.9 (1.6×) | n/a | 7.0 3.0 (2.3×) | 16.8 9.1 (1.8×) | n/a | n/a | 12.4 5.5 (2.2×) |
| | 128 | 2.5 1.8 (1.4×) | 4.9 3.4 (1.4×) | 10.8 6.8 (1.6×) | 2.4 1.4 (1.7×) | 4.7 3.1 (1.5×) | 10.0 6.4 (1.6×) | n/a | 4.2 1.9 (2.2×) | 9.6 5.6 (1.7×) | n/a | n/a | 7.5 3.7 (2.1×) |
| | 255 | 1.4 1.0 (1.4×) | 3.1 2.0 (1.5×) | 8.1 4.8 (1.7×) | 1.3 0.8 (1.6×) | 3.1 1.8 (1.7×) | 6.1 3.8 (1.6×) | n/a | 2.6 1.3 (2.0×) | 5.8 3.3 (1.8×) | n/a | n/a | 4.5 2.6 (1.7×) |

almost all datasets. For example, on the 2-byte tpch-int32 dataset, the compression ratio is 1.26, 1.31, 1.35, and 1.39 when the sliding window size is 32, 64, 128, and 255, respectively. Moreover, the overall compression ratio improvement by extending the sliding window size from 32 to 255 is 1.4×. On the other hand, a larger sliding window incurs more operations per thread, decreasing the compression throughput. The average speedup when we change the sliding window size from 255 to 32 is 3.9×. Compared with the relatively small increase in compression ratio, the throughput decreases dramatically as the sliding window size doubles. However, we find GPULZ highly stable in throughput across different datasets under the same configuration (i.e., $C$, $W$, and $S$). For example, the throughput of $C = 2048$, $W = 32$, and $S = 2$ is 9.57 GB/s, 8.49 GB/s, 10.14 GB/s, 8.3 GB/s, 7.96 GB/s, and 9 GB/s on A4000 on {hurr, hacc, nyx}-quant, tpch-{int32, string}, and rtm datasets, respectively.

Finally, we discuss the impact of the symbol length $S$. As mentioned, the multi-byte symbol length can introduce a potential compression ratio improvement and increase the compression throughput due to longer matches and fewer symbols to process. Table 1 shows that the compression ratio improvement is not determined as we expected. It has different patterns for different datasets. For example, on the three uint16 quantization-code datasets, the compression ratio reaches the peak at $S = 2$, which is the same as the length of uint16. However, on the int32 tpch-int32 dataset, the compression ratio is optimal at $S = 1$, which is different from the length of int32. This is because the number of repeated patterns is relatively smaller in the tpch-int32 dataset, as indicated by the low compression ratio. Thus, using a 1-byte symbol (i.e., $S = 1$) may detect more byte-level repeated patterns and achieve a higher compression ratio than using a 4-byte symbol. On the utf-8 tpch-string dataset and the float32 rtm dataset, the best compression ratio is achieved at $S = 1$ and $S = 4$, respectively, which is the same as the unit length of their data types.

Regarding throughput, the impact of the symbol length is more obvious; that is, longer symbol results in higher throughput. The average throughput improvement is 4.5× when we change $S$ from 1 to 4. Combined with the above observation regarding compression ratio, we find that $S = 2$ has both a higher compression ratio and throughput than $S = 1$ in some cases. For example, on the hurr-quantization dataset with any $W$ and $C$, $S = 2$ can always lead to a better compression ratio and throughput than $S = 1$. Note that this observation can be generalized to all LZ compressors.

## 4.3 Evaluation on Compression Ratio

Next, we compare the compression ratio of GPULZ with CULZSS and nvCOMP's LZ4, as shown in Figure 8. Note that in the figure, we use "gpulz" to denote the default configuration ($C = 2048$, $S = 2$, and $W = 128$) and "gpulz-best" to denote the best compression ratio from all settings. The figure shows that compared with CULZSS, GPULZ achieves a similar compression ratio on all datasets because

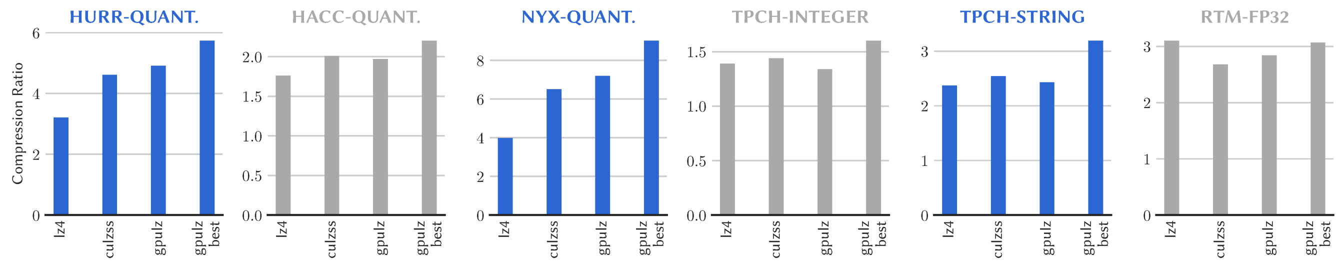



**Figure 8: Compression ratio of different GPU compressors.**

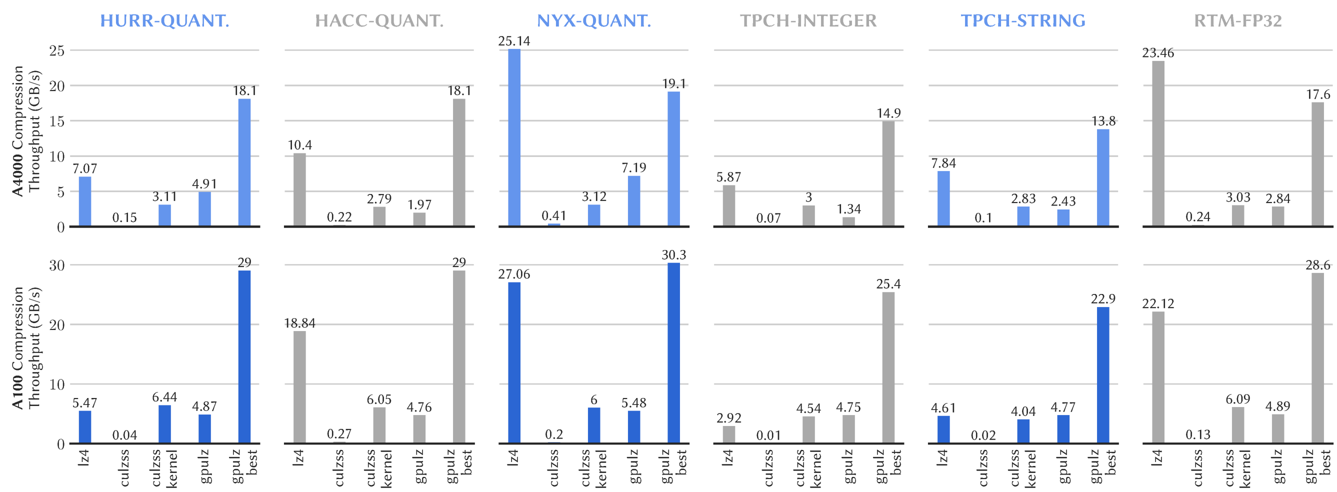



**Figure 9: Compression throughput of different GPU compressors on A100 and A4000.**

the compression ratio is highly dependent on the sliding window size. In our default configuration, we use $W = 128$ as the same as the CULZSS. In the best cases (overall configurations), GPULZ has an improvement of 1.4× on compression ratio. Compared with nvCOMP's LZ4, GPULZ achieves an average compression ratio improvement of 1.23×. Specifically, on the hurr-quantization and nyx-quantization datasets, GPULZ has the highest compression ratio improvements, which are 1.53× and 1.8×, respectively. In the best cases (of all configurations), GPULZ has an improvement of 1.42× on compression ratio thanks to our fine-tuned parameters.

## 4.4 Evaluation on Compression Throughput

Then, we evaluate the performance of GPULZ, with its scalability on A100 and A4000 shown in Figure 9. Note that "culzss" denotes the overall throughput of CULZSS, including the GPU matching kernel and the CPU encoding process. "culzss-kernel" denotes the throughput of the GPU matching kernel. "gpulz" denotes our method with the default setting ($C = 2048$, $S = 2$, and $W = 128$). "gpulz-best" denotes the best compression throughput from all settings. We note that the settings for the best case are generally C=2048, W=32, S=4, except for the nyx-quantization and rtm datasets on A100, which achieve the best performance with C=4096, W=32, S=4. This is because A100 has a larger L1 cache (192 KB/SM) than A4000 (128 KB/SM); a larger chunk size (i.e., more shared memory utilization) will not significantly affect performance. Compression throughput is potentially increased by fully utilizing the high-speed shared memory.

Compared with CULZSS, our method has an average speedup of 22.19× on all datasets with A4000. When compared with our best case, this speedup increases to 130.01×. The reason is three-fold: 1 the slow CPU encoding process, 2 the data movement overhead between CPU and GPU, and 3 the overhead of multiple times of launching the same kernel. Note that the entire process of GPULZ is almost as fast as the matching kernel of CULZSS both on A100 and A4000, thanks to our optimizations in both algorithm and implementation. Compared with nvCOMP's LZ4, GPULZ has similar compression throughputs on the hurr-quantizaiton, tpch-int32, and tpch-string datasets but slightly slower on the hacc-quantization and nyx-quantization datasets. However, since nvCOMP is not an open-source library, we can only infer the underlying reason, which may be that some field sizes in these datasets are too small. By comparison, GPULZ has more stable performance across all datasets, and our best case achieves higher throughput than nvCOMP with both A100 and A4000 platforms on almost all datasets. For example, the average speedup is 4.32× on A100.

In addition, we also implement our decompression and evaluate its throughput. Considering decompression is easy to parallel, we do not describe and compare it with other compressors in detail; instead, we only show the average decompression throughput across all datasets. Specifically, the average decompression throughput of GPULZ on all datasets is 16.4 GB/s on A4000 and 29.1 GB/s on A100.

For comparison, nvCOMP's LZ4 has an average decompression throughput of 21.1 GB/s on A4000.

### 4.5 Use-case of gpuLZ

Finally, we apply gpuLZ to cuSZ (a state-of-the-art GPU lossy compressor for scientific data) due to its high performance on the quantization-code datasets to improve the compression ratio. Note that the original cuSZ only has a Huffman encoding [16], whereas the improved cuSZ includes gpuLZ before the Huffman encoding. We evaluate the original cuSZ and the improved cuSZ on the A100 platform under the relative error bound 1e-2. Besides Hurricane, NYX, and RTM, we also include one more dataset from SDRBench, i.e., CESM (climate simulation) [5].

**Table 3: Comparison of compression ratio and throughput (GB/s) between original cuSZ and improved cuSZ (with gpuLZ) on A100 platform.**

| Dataset | cuSZ | | cuSZ w/ gpuLZ | |
|---|---|---|---|---|
| | CR | THR | CR | THR |
| CESM | 22.6 | 12.0 | 43.2 | 2.7 |
| Hurricane | 24.3 | 31.9 | 29.1 | 5.9 |
| Nyx | 30.1 | 87.2 | 74.8 | 10.4 |
| RTM | 28.6 | 49.2 | 249.8 | 7.2 |

Table 3 shows that the improved cuSZ obtains an improvement of 1.9× ∼ 8.7× in compression ratio with a slightly lower compression throughput. We note that the improved cuSZ has higher compression ratio improvements on larger error bounds and higher dimensional datasets (e.g. 3D Hurricane and RTM), since the quantization code generated by cuSZ in these cases has more spatial redundancy, thus benefiting gpuLZ. Enabling higher compression ratios is critical for many HPC applications using lossy compression (rather than lossless compression). We also note that some CPU lossy compressors with multi-threading support such as SZ [38] and ZFP [23] can also achieve compression throughputs of about 2∼4 GB/s on 32 cores [11], but their overall throughput is limited by moving uncompressed data from the GPU to the CPU; in comparison, the time of moving compressed data (with hundreds of compression ratios) with the improved cuSZ is much lower.

## 5 CONCLUSION AND FUTURE WORK

In this paper, we propose a series of optimizations for one of the most important lossless compression algorithms LZSS for multi-byte data on GPUs. Specifically, we develop a new method for multi-byte pattern matching, optimize the prefix-sum operation, and fuse multiple GPU kernels, thereby improving both compression ratio and throughput (due to lower computational time complexity, less data movement, and potentially longer matches). gpuLZ achieves up to 272.1× speedup and up to 1.4× higher compression ratio over state-of-the-art solutions.

In the future, we plan to evaluate gpuLZ on more multi-byte datasets. We will attempt to develop an analytical model for searching the optimal parameter combination for different datasets. In addition, we will integrate gpuLZ into more data-intensive applications running on different parallel and distributed systems.

## ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357. This work was also supported by the National Science Foundation under Grants OAC-2003709, OAC-2104023, OAC-2303064, OAC-2247080, and OAC-2312673. This research was also supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute.

# REFERENCES


[1] About Big Red 200 at IU. https://kb.iu.edu/d/brcc.

[2] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. Communication-avoiding qr decomposition for gpus. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 48–58. IEEE, 2011.

[3] Guy E Blelloch. Prefix sums and their applications. 1990.

[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. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, 2019.

[5] CESM Large Ensemble Data Sets. https://www.cesm.ucar.edu/projects/community-projects/LENS/data-sets.html.

[6] CUB Prefix Sum. https://nvlabs.github.io/cub/structcub_1_1_device_scan.html.

[7] CULZSS. https://github.com/adnanozsoy/CUDA_Compression.

[8] cuSZ: A CUDA-Based Error-Bounded Lossy Compressor for Scientific Data. https://github.com/szcompressor/cuSZ.

[9] Peter Deutsch. Rfc1951: Deflate compressed data format specification version 1.3, 1996.

[10] M Dmitriev, T Tonellot, HJ AlSalem, and S Di. Error-bounded lossy compression in reverse time migration. In *Sixth EAGE High Performance Computing Workshop*, volume 2022, pages 1–5. EAGE Publications BV, 2022.

[11] Griffin Dube, Jiannan Tian, Sheng Di, Dingwen Tao, Jon C. Calhoun, and Franck Cappello. Efficient error-bounded lossy compression for cpu architectures. In *30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2022, Nice, France, October 18-20, 2022*, pages 1–8. IEEE, 2022.

[12] Thomas M Evans, Andrew Siegel, Erik W Draeger, Jack Deslippe, Marianne M Francois, Timothy C Germann, William E Hart, and Daniel F Martin. A survey of software implementations used by application codes in the exascale computing project. *The International Journal of High Performance Computing Applications*, 36(1):5–12, 2022.

[13] FastLZ. https://ariya.github.io/FastLZ/.

[14] Evangelos Georganas, Jorge González-Domínguez, Edgar Solomonik, Yili Zheng, Juan Tourino, and Katherine Yelick. Communication avoiding and overlapping for numerical linear algebra. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[15] 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. *Communications of the ACM*, 60(1):97–104, 2016.

[16] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

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

[18] Sian Jin, Sheng Di, Jiannan Tian, Suren Byna, Dingwen Tao, and Franck Cappello. Improving prediction-based lossy compression dramatically via ratio-quality modeling. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2494–2507. IEEE, 2022.

[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 *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 105–115. IEEE, 2020.

[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*, pages 45–56, 2021.

[21] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. Comet: a novel memory-efficient deep learning training framework by using error-bounded lossy compression. *Proceedings of the VLDB Endowment*, 15(4):886–899, 2021.

[22] Suha N Kayum, Thierry Tonellot, Vincent Etienne, Ali Momin, Ghada Sindi, Maxim Dmitriev, and Hussain Salim. Geodrive-a high performance computing flexible platform for seismic applications. *First Break*, 38(2):97–100, 2020.

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

[24] Tong Liu, Jinzhen Wang, Qing Liu, Shakeel Alibhai, Tao Lu, and Xubin He. High-ratio lossy compression: Exploring the autoencoder to compress scientific data. *IEEE Transactions on Big Data*, 2021.

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

[26] Maleeha Najam, Usman Younis, and Raihan Ur Rasool. Multi-byte pattern matching using stride-k dfa for high speed deep packet inspection. In *2014 IEEE 17th International Conference on Computational Science and Engineering*, pages 547–553. IEEE, 2014.

[27] nvCOMP: A library for fast lossless compression/decompression on the GPU. https://github.com/NVIDIA/nvcomp.

[28] NVIDIA cooperative groups. https://developer.nvidia.com/blog/cooperative-groups/.

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

[30] Adnan Ozsoy. Culzss-bit: A bit-vector algorithm for lossless data compression on gpgpus. In *2014 International Workshop on Data Intensive Scalable Computing Systems*, pages 57–64. IEEE, 2014.

[31] Adnan Ozsoy and Martin Swany. Culzss: Lzss lossless data compression on cuda. In *2011 IEEE International Conference on Cluster Computing*, pages 403–411. IEEE, 2011.

[32] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Pipelined parallel lzss for streaming data compression on gpgpus. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 37–44. IEEE, 2012.

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

[34] Cody Rivera, Sheng Di, Jiannan Tian, Xiaodong Yu, Dingwen Tao, and Franck Cappello. Optimizing huffman decoding for error-bounded lossy compression on gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 717–727. IEEE, 2022.

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

[36] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[37] Masayuki Takeda, Satoru Miyamoto, Takuya Kida, Ayumi Shinohara, Shuichi Fukamachi, Takeshi Shinohara, and Setsuo Arikawa. Processing text files as is: Pattern matching over compressed texts, multi-byte character texts, and semi-structured texts. In *International Symposium on String Processing and Information Retrieval*, pages 170–186. Springer, 2002.

[38] 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*, pages 1129–1139, Orlando, FL, USA, 2017. IEEE.

[39] 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 *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 283–293. IEEE, 2021.

[40] 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*, pages 3–15, 2020.

[41] 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 *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 881–891. IEEE, 2021.

[42] TPCH. https://www.tpc.org/tpch/.

[43] Daoce Wang, Jesus Pulido, Pascal Grosset, Sian Jin, Jiannan Tian, James Ahrens, and Dingwen Tao. Tac: Optimizing error-bounded lossy compression for three-dimensional adaptive mesh refinement simulations. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, page 135–147, 2022.

[44] Jinzhen Wang, Tong Liu, Qing Liu, Xubin He, Huizhang Luo, and Weiming He. Compression ratio modeling and estimation across error bounds for lossy compression. *IEEE Transactions on Parallel and Distributed Systems*, 31(7):1621–1635, 2019.

[45] André Weißenberger and Bertil Schmidt. Accelerating jpeg decompression on gpus. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 121–130. IEEE, 2021.

[46] Raymond K Wong, Fengming Shi, and Nicole Lam. Full-text search on multi-byte encoded documents. In *Proceedings of the 2012 ACM symposium on Document engineering*, pages 227–236, 2012.

[47] 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 (HPCA)*, pages 284–295. IEEE, 2014.

[48] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.