High Throughput GPU-Accelerated FSST String Compression

Tim Anema
Delft University of Technology
Delft, The Netherlands
tim.anema@hotmail.nl

Joost Hoozemans
Voltron Data
USA
joosthooz@gmail.com

Zaid Al-Ars
Delft University of Technology
Delft, The Netherlands
z.al-ars@tudelft.nl

H. Peter Hofstee
IBM
Texas, USA
hofstee@us.ibm.com

ABSTRACT

Slow PCIe bandwidth represents a bottleneck for I/O-bound applications such as GPU-accelerated data analytics. Compression can improve ingestion throughput, but contemporary GPU compressors are much slower than the latest PCIe buses. The sequential nature of widely used LZ-based compression proves challenging for the GPU's SIMT-based architecture.

This paper introduces a GPU-accelerated compressor based on the FSST (*Fast Static Symbol Table*) compressor, providing a throughput of 74 GB/s on an RTX4090 while maintaining its compression ratio. The resulting compression pipeline is 3.86x faster than nvCOMP's LZ4 compressor, while providing similar compression ratios (0.84x). We achieved this by creating a memory-efficient encoding table, an encoding kernel that uses a voting mechanism to maximize memory bandwidth, and an efficient gathering pipeline using stream compaction.

Additionally, our compressor is compatible with a modified version of the GSST decompressor, which is capable of decompressing at 191 GB/s, to provide a high-throughput end-to-end (de)compressor.

VLDB Workshop Reference Format:
Tim Anema, Joost Hoozemans, Zaid Al-Ars, and H. Peter Hofstee. High Throughput GPU-Accelerated FSST String Compression. VLDB 2025 Workshop: 16th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS25).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/timanema/fsst-gpu.

1 INTRODUCTION

Modern analytical engines handle large amounts of data and are starting to leverage GPU accelerators to benefit from the rapid increase in throughput potential [11–13, 16]. With the release of NVIDIA's new Blackwell architecture, systems have access to HBM3e memory with a large bandwidth of 1TB/s per stack [24]. Even though these recent advances in memory bandwidth are impressive, ingesting data into GPU memory happens through PCIe, which is often a bottleneck in I/O-bound applications such as data analytics. Conceptually, compression could alleviate that bottleneck, but the throughput of (de)compressing data on a GPU is currently an order of magnitude slower than most other operations in analytics pipelines [11, 12, 16]. For example, joins and aggregations can achieve a throughput of 100s of GB/s, while in contrast, most compressors in NVIDIA's nvCOMP library do not reach more than 30 GB/s [23]. An important reason is that data compression often uses an LZ-based algorithm [39], which is a poor match to the GPU's SIMT model of computation [29]. Compression is a field that has been widely studied in the past [2, 6, 9, 21, 26, 30–32, 38, 39].

In the context of data analytics, *de*compression is most important for data ingestion. NVIDIA introduced the *Decompression Engine* with Blackwell, which is reported to achieve decompression speeds of 180 GB/s for Snappy on a B200 [22, 24]. In addition, other GPU decompressors have been proposed [20, 33, 36]

When considering big data query engines, there are also interesting gains to be found for compression. GPU memory is a scarce and expensive resource, creating a necessity to temporarily offload memory to host memory (or fast storage, for example, using GDS [34]). Another use case is distributing (shuffling) data between GPUs on a multi-device system or to other nodes in a cluster.

This paper introduces a novel heterogeneous GPU-CPU compressor based on the FSST (*Fast Static Symbol Table*) [6] string compression algorithm. Our compressor is compatible with a modified version of the GSST decompressor [36], which allows for a full compression and decompression cycle. We highlight the issues with running FSST on a GPU and propose mitigations. Furthermore, we show how to enhance throughput on the GPU with various optimization techniques, such as adding transposition stages.

This paper has the following contributions:

- An analysis of the FSST compression bottlenecks on the GPU
- Various GPU optimization techniques and their impact on throughput
- An optimized GPU-accelerated FSST compression implementation achieving 74GB/s throughput

The paper is organized as follows. We will touch upon related work and general GPU development background in Section 2. We will then analyze the acceleration potential of FSST and possible

Proceedings of the VLDB Endowment. ISSN 2150-8097.

inhibitors in Section 3. We will then provide a memory-efficient encoding table in Section 4, and use it in the encoding kernel implementation in Section 5. The overall compression pipeline will be discussed in Section 6, and we will evaluate its performance in Section 7. Finally, we will conclude in Section 8 and discuss some potential future work.

2 BACKGROUND

In Section 1, we have established that string (de)compression is a relevant problem for big data analytics. Most CPU compression schemes predate the use of GPUs as general-computing accelerators and offer limited acceleration potential. Nonetheless, significant work has been done to port those existing schemes to GPUs.

One example is the CULZSS algorithm [26], which has had several follow-ups [27, 28]. The initial papers implemented the LSZZ algorithm on NVIDIA GPUs, primarily by splitting data into chunks. Several derivates of this include CULZSS-bit [25], GLZSS [40], and GMATCH [21].

Other examples of CPU algorithms ported to GPUs include compressors included in the nvCOMP [22] library, such as Snappy, LZ4, (G)Deflate, and ZSTD. To the best of our knowledge, GPULZ [38] is the fastest LZ-based (LZSS) GPU compressor outside of nvCOMP with a best-case throughput of approximately 29 GB/s.

For newer systems and data formats, there is an increasing effort to emphasize the parallelization potential. An example of this is the FastLanes format [1], which is partially implemented on GPUs [2]. Some more recent (numerical) compressors include Bitcomp [22], SPspeed/SPratio [4], DietGPU [18], and Ndzip [19]. While these compressors focus on numerical data, they can (mostly) also be applied to string data, but at the cost of a low compression ratio.

Compression acceleration can also be achieved with hardware other than GPUs, such as FPGAs [7, 8] and NVIDIA's data processing units (DPUs) with hardware compression [37].

2.1 FSST

FSST is essentially a dictionary coder that replaces frequently occurring strings (symbols) with a length of one to eight bytes with smaller single-byte symbols. The compression process involves creating a symbol table for every block and then replacing matching entries in the block with their corresponding codes. Bytes not matched by any symbol in the table will be escaped with a special character.

Figure 1 shows an example of the FSST compression process. During encoding, FSST transforms the input data stream to a smaller data stream using the symbol table, or encoding table, for every block. It scans the input stream and identifies the longest matching symbol, it will then append the corresponding code to the output stream. When no match is found, a special escape character will be added in addition to the first byte, indicating to the decompressor that the next byte should be interpreted as data instead of a code. The encoded stream, together with metadata such as the symbol table, forms the output data of the compression algorithm.

Decompression is the reverse operation, where every byte is expanded to one or more bytes while taking special care of escape characters.

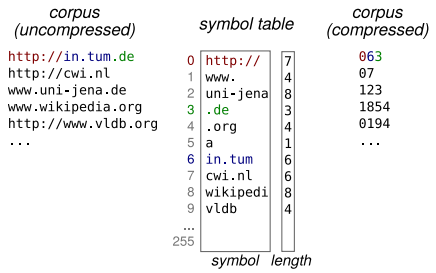


Figure 1: An example of FSST compression. The uncompressed data is encoded to a (smaller) format using a static dictionary. Source: [6]

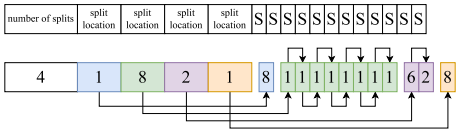


Figure 2: The split format GSST uses. Every block is divided into splits, which individual threads will process. Source: [36]

The use of a static symbol table enables random access to compressed data, without needing to decompress an entire data block. This feature is particularly useful in the context of databases. Additionally, the use of a static table introduces an opportunity for acceleration, which is the focus of this paper.

2.2 GSST

GSST [36] provides a partial solution to high-throughput string compression. The authors provide a high-throughput decompressor that introduces some changes to the FSST data format. GSST achieves high throughput using additional block-level metadata and a tiling-based approach to distribute work over multiple threads. By applying tiling, GSST creates parallelism within the block level, which allows it to decompress blocks in SIMT fashion.

The main problem is that the location where each thread should output its decompressed data is unknown. GSST relies on the compressor providing metadata detailing the structure of a block. The decompressor can then use this information in the file header to deduce where every thread should output its data. The file header following their *splits format* can be seen in Figure 2.

Overall, GSST achieves considerable throughput while maintaining the high compression ratio that the FSST table generation algorithm provides by limiting the amount of information it needs from a compressor to reconstruct the original output structure. However, the original version of GSST does not include a high-throughput compressor, has been tested with limited datasets, and does not provide any source code. For that reason, we aim to keep our compressor mostly compatible with the GSST format so that we can create a more complete software package in the future. We will discuss this further in Section 6.4.

2.3 GPU development

A Graphical Processing Unit (GPU) is a special processor originating in graphics processing, such as shaders. A GPU follows the *Single instruction, Multiple threads* (SIMT) paradigm, a combination of *Single instruction, Multiple data* (SIMD) and multithreading. This execution model is suitable for algorithms that can be massively parallelized and run on general-purpose GPUs (GPGPUs).

At the core of GPUs lie many small cores, which are grouped in *Streaming Processors* (SMs), each with its own schedulers, register files, and caches. The SMs can execute multiple threads simultaneously, achieving high throughput through parallelism. Threads running on an SM are grouped into warps, which run in lockstep. This means all threads execute the same instructions, potentially leading to inefficiencies if there is divergence between threads in the same warp.

NVIDIA introduced the CUDA API to use the available compute on GPUs in 2007. CUDA includes drivers, compilers, development tools, and libraries, enabling the use of NVIDIA GPUs for general-purpose computing via languages such as C++. While ROCm is available for AMD GPUs, this paper only focuses on NVIDIA platforms.

A CUDA kernel is executed by many threads grouped together in thread blocks. The thread blocks form a kernel grid. Threads within a block are executed on the same SM, and a grid is divided over many SMs. Threads within a block are grouped in blocks of 32 threads called warps. A block cannot be migrated to a different SM, but a single SM can execute multiple blocks. A GPU contains many SMs, so underutilized SMs can be used to execute different kernels.

One effect of this architecture is that all threads within a block are guaranteed to use the same L1 memory, which enables its use as shared memory. Some algorithms use collective communication operations, such as parallel reductions and scans. Shared memory is used as a communication layer for this purpose. When communication is confined to threads within the same warp, warp-level primitives provide a more efficient mechanism

CUDA has three categories of warp-level primitives: synchronized data exchange, active mask query, and thread synchronization. With synchronized data exchange, threads can exchange data directly through registers and use voting functions. This allows threads within a warp to perform a reduction fully in the register file, for example. Another example is accelerating stream compaction using ballots [5, 17].

3 ACCELERATION POTENTIAL OF FSST

FSST generates a symbol table based on its bottom-up approach and then encodes the input data in a more compact format. With its AVX512-based encoder kernel, FSST encodes up to 24 strings in parallel using an encoding table consisting of hashtables and an additional lookup table for short symbols.

There are two main steps in the process: table generation and encoding. Table generation can be parallelized as there are many tables to be generated, but the process of generating a single table is highly sequential. Furthermore, the divergence between processes is high, and the process uses data structures unfit for a GPU, such as a priority queue. However, table generation only needs a small sample of the data to work with, so modifying this to run in parallel on the CPU will already yield high throughput. The encoding stage operates on all data and, therefore, must be executed on the GPU itself. To achieve parallelism, we can divide the data into tiles and encode each tile in a separate thread, a common technique often called tiling or chunking [1, 2, 31, 36].

For that reason, our accelerated compression pipeline will focus on GPU-accelerated encoding combined with multi-threaded table generation on the CPU. A heterogeneous design like this is best suited to the FSST compression algorithm. For that reason, we will shift our focus to potential blockers for a GPU-accelerated encoding kernel.

One issue with encoding is that the encoding table does not fit in shared memory because of the significant size of the lookup table used for shortcodes. This lookup table is around 130kB in size, while the hash table uses an additional 16kB of memory, totaling 146kB of shared memory usage. This means the table has to be stored in global memory, which is not suited for random accesses like those bound to happen in a lookup table.

Another issue is the alignment of input data (and output, for that matter). String data is essentially a sequence of 8-bit values, which is unnatural for GPUs that use 32-bit registers. This means that every operation on 8-bit values that is not bit-packed to 32-bit registers effectively wastes bandwidth. FSST string matching uses 64-bit values to match up to eight characters, which would map to eight 8-bit loads from memory in a naive implementation.

Finally, since we use tiling to create parallelism, our input data tiles, and therefore also the output data tiles, will be in consecutive blocks in memory. Consequently, threads within a warp will not work with consecutive memory addresses from global memory, and no memory coalescing can occur with reading or writing. This drastically lowers the effective memory bandwidth and, therefore, our overall compression throughput.

4 MEMORY-EFFICIENT ENCODING TABLE

We will first address the size of the encoding table. First, we will investigate how the encoding table is used and where potential gains are. Based on these observations, we will introduce our own encoding table, which is more memory efficient and is structured in a way that is efficient for GPUs.

4.1 Data properties

The encoding table consists of two main lookup structures: the hashtable and the shortcodes matrix. The hashtable is used for symbols with a length between three and eight, while the shortcodes matrix is used to efficiently encode symbols that consist of one or two characters. Both lookup structures are very sparse; the hashtable stores up to 1024 symbols with a memory footprint of sixteen bytes each, while the shortcodes matrix can theoretically store up to 65536 symbols that take up 2 bytes each. In reality, their usage will depend on the actual dataset, but it will be much lower for compressible data. Especially in the context of textual data, since many combinations are not present in natural language.

To investigate a realistic data structure usage, we will examine the resulting encoding tables generated by FSST for three datasets: TPC-H [35], GDelt [10], and DBText [6]. To be more specific, we will use textual data from the TPC-H *lineitem* and *customer* tables,

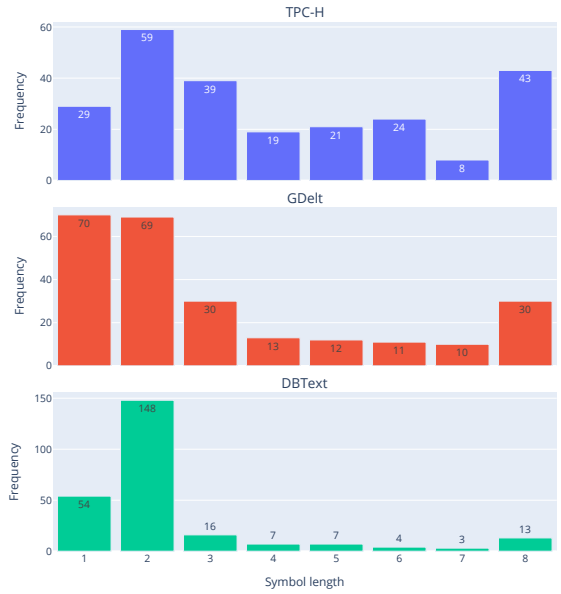


Figure 3: This histogram shows the spread of symbols regarding their length for three datasets: TPC-H, GDelt, and DBText. In general, we can see that the DBText corpus heavily uses short symbols, while the other datasets also use longer symbols more often.

location data from GDelt, and the machine-readable datasets from the DBText corpus used by the original FSST authors.

Figure 3 shows the average lengths of symbols generated for the three datasets. We can see that the hashtable is responsible for a significant portion of the symbols. In the case of TPC-H and GDelt, the hashtable stores 64 percent and 43 percent of all symbols, respectively. We can also observe that machine-readable data in the DBText set, such as hex data, almost exclusively uses the shortcodes data structure.

The shortcodes lookup table effectively works as a 2D matrix. Retrieving the code and length of a given symbol is achieved by accessing the location that corresponds to the two characters; the first character is used to identify the *row*, and the second character is used to identify the *column*. This means a lookup consists of a single memory access into a very sparse matrix.

For this reason, we do not only specify the usage of the shortcodes data structure in terms of cells used, but rather in the maximum and average usage of rows and columns within a row. The number of rows tells us something about how many symbols, with a length of two characters, start with the same character. Similarly, the number of columns within a row tells us something about how many combinations of symbols with the same starting character exist.

Table 1 shows the usage of the shortcodes data structure for the three datasets, in terms of the metrics described above. Note that we only look at symbols with a length of two characters. We can see that while the overall matrix is very sparse, the actual data is relatively dense. The number of rows is relatively small compared to the potential number of rows, which makes sense considering most characters are not used in purely textual data. Furthermore, we can see that the (average) number of symbols that start with the same character, so the average number of entries (columns) in the same row, is relatively low.

Table 1: The usage of the lookup table in terms of row and column usage. A row is used when there is a 2-byte symbol starting with the character corresponding to the row. The number of columns described in this table refers to the columns used within the same row; in other words, the number of 2-byte symbols that start with the same character.

Dataset	Max/avg rows	Max/avg columns
TPC-H	26/16.7	15/3.7
GDelt	36/29.1	12/2.4
DBText	38/26.8	19/6.5

For textual data, the hashtable contains 130 out of 1024 symbols on average, and the shortcodes table contains 13148[1] out of 65536 possible combinations on average. Note that all symbols that consist of a single character use 256 entries in the lookup table. For machine-readable data, such as that found in DBText, the hashtable contains 50 symbols, and the shortcodes table contains 13972 entries.

4.2 Modifying the hashtable

The size of the hashtable directly influences the number of hash collisions, as the size is used in a modulo operation. For this reason, the size cannot easily be lowered to more closely fit the observed usage. We can, however, introduce indirection to the hash lookup. This means that one table is used to store the actual data, while a (more memory-efficient) table is used to store hash locations. The number of possible data entries can then be modified without affecting the number of entries in the hashtable, and as a result, the number of hash collisions.

Another minor modification we can perform has to do with the memory organization of the actual symbol structure. In the original implementation, a hashtable entry consists of a 64-bit number representing the symbol data and a 32-bit number to store metadata such as the code and length. This allows the encoding kernel to perform direct 64-bit comparisons. However, this also forces the compiler to align the structure to 8-byte boundaries, which requires four bytes of padding. A GPU does not perform direct 64-bit comparisons, but uses two 32-bit comparisons. For that reason, we split the 64-bit number into two 32-bit numbers representing the high and low sides. Consequently, the structure can now be aligned to four bytes, resulting in less padding.

Overall, this changes the memory requirement from $1024 * 16$ to $1024 + 12 * X$ at the cost of an additional lookup, where X is the size of the secondary data table. This parameter balances the compression ratio and, indirectly, performance. We will investigate the effect of this parameter in Section 7.1.

[1] $((29 * 256 + 59) + (70 * 256 + 69) + (54 * 256 + 148))/3$

4.3 Efficient short symbols

We have already established that the shortcode structure is essentially a sparse matrix. Furthermore, we have observed that most rows are not used and that the number of entries in a single row is also relatively low when only storing symbols with a length of two bytes. For this reason, we will store single-byte symbols in a separate data structure and only use the shortcode table for symbols of length two.

4.3.1 ELL matrix. One data structure that could more efficiently represent this data pattern is an ELL matrix based on the sparse matrix package in ELLPACK [14]. The original matrix can be changed to a NxK matrix, where K is the new number of columns, and all non-zero elements within a row are compacted. While the ELL format leads to a significant reduction in size, the matrix is still sparse, storing more than 200 empty rows. Additionally, a GPU uses 32 banks to address shared memory, meaning a single bank will serve eight rows of this matrix, likely leading to bank conflicts as the characters used in textual data are in close proximity.

4.3.2 Match table. We address these limitations with our own matching table. The main idea behind the matching table is that we translate the lookup table to a format that allows the GPU to perform a series of computations to get the final result. We achieve this by creating a series of masks and then applying the masks to all codes for a particular row. The masking function uses the fact that $-(A == B)$ for unsigned numbers returns all zeros (`0x00`) when $A \neq B$ and all ones (`0xFF`) when $A = B$.

We can select the row from the first character in a two-byte symbol XY using a small lookup table. This row then consists of several symbol-code pairs (SC pairs): a tuple containing a symbol (Y) that can be used to create a mask and the code corresponding to the combination of the row character with the symbol in the SC pair. When the row has been selected, the GPU uses all SC pairs in that row to generate the masks for all pairs and then applies the mask to the respective codes. All results are then OR'ed to generate the final code from that, which works because there is a maximum of one match per row. Listing 1 shows the lookup algorithm, the buildup algorithm, and the required memory structures for the match table.

The underlying SC pairs are represented in 32-bit words. Every word contains two SC pairs. The reason we use a 32-bit number is twofold: shared memory uses 32-bit words, both in addressing and servicing. Additionally, GPUs use 32-bit registers, so anything more than that will be split into 32-bit words anyway. This means we can represent K pairs in $K * 2$ bytes. We then use R rows, which must be a multiple of 32, to create a RxK matrix and store it in a column-major format. When R is a multiple of 32, there are no bank conflicts, and we reduce the memory usage even further to $R * K * 2 + 256$ bytes.

Note that the parameters R and K directly map to the row and column usage described in Section 4.1, and will influence the final compression ratio and, indirectly, performance. We have slightly modified the original FSST table generation algorithm to respect the additional constraints defined by these parameters and pick the next best option if a constraint would be violated. We will investigate the effects of these parameters in Section 7.1.

```
struct SymbolMatch { // Represents two symbol-code pairs
    uint32_t val_sc_pairs;

    SymbolMatch(uint8_t s1, uint8_t c1, uint8_t s2, uint8_t c2) :
            val_sc_pairs(s1 << 24 | c1 << 16 | s2 << 8 | c2) {}

    uint8_t get_val_if_equal(uint8_t b, uint8_t c, uint8_t val) {
        return -(b == c) & val; // Returns val, if b == c
    }

    // Returns code if symbol matches any symbol, otherwise 0
    uint8_t match(uint8_t symbol) {
        return get_val_if_equal(symbol, val_sc_pairs >> 24,
                            val_sc_pairs >> 16) |
            get_val_if_equal(symbol, val_sc_pairs >> 8,
                        val_sc_pairs);
    }
};

struct SymbolMatchTable {
    SymbolMatch matches[rows * matchesPerRow]; // R * K
    uint8_t row_indices[256]{};

    SymbolMatchTable(Symbol shortCodes[65536]) {
        memset(row_indices, 255, 256); // Escape (255) by default
        uint16_t values[rows][matchesPerRow * 2] = {};
        uint8_t usedRows = 0; // assert(usedRows < rows)
        for (uint16_t a = 0; a < 256; a++) {
            bool matches = false;
            int col = 0; // assert(col < matchesPerRow * 2)

            for (uint16_t b = 0; b < 256; b++) {
                if (Symbol ts = shortCodes[a | b << 8];
                        ts.code() != 255) {
                    matches = true;
                    // We need to maintain escape == 0, so +1
                    values[usedRows][col] = b << 8 | ts.code() + 1;
                    col += 1;
                }
            }

            // If any 2-byte symbol is found in this row, save it
            if (matches) {
                row_indices[a] = usedRows;
                usedRows += 1;
            }
        }

        // And now construct all the symbol-code pairs structs
        for (uint8_t row = 0; row < usedRows; row++) {
            for (int i = 0; i < matchesPerRow; i++) {
                uint16_t sc1 = values[row][i * 2];
                uint16_t sc2 = values[row][i * 2 + 1];

                matches[i * rows + row] =
                    SymbolMatch(sc1 >> 8, sc1, sc2 >> 8, sc2);
            }
        }
    }

    uint8_t lookup(uint8_t x, uint8_t y) {
        const uint8_t row = row_indices[x];
        if (row == 255) {
            return 255; // No row found == escape for 2-byte lookup
        }

        uint8_t result = 0;
        for (int i = 0; i < matchesPerRow; i++) {
            SymbolMatch match = matches[rows * i + row];
            result |= match.match(y); // OR entire row
        }

        return result - 1; // Restore to original code
    }
};
```

Listing 1: All the required memory structures and algorithms for the match table. It is constructed from FSST structures and then used in our GPU encoding kernel.

5 ENCODING KERNEL DESIGN

The main challenge of accelerating the overall compression pipeline lies in efficient encoding. For one, we need to create parallelism within a single FSST block to make effective use of the GPU's massive parallelism. Furthermore, we need to mitigate the issues mentioned in Section 3, besides the encoding table size.

In this section, we will describe a basic compression pipeline and define the interfaces of the encoding kernel. We will describe how we can mitigate the issue of memory alignment and how we can achieve coalesced memory operations despite working with non-contiguous tiles.

5.1 Applying tiling

After the tables have been created, the encoding stage will start. Encoding is done on a block level, i.e., every FSST block can be encoded separately. This is the first level of parallelism and maps fairly naturally to a CUDA thread block. To create parallelism within a (thread) block, we utilize the tiling technique. We will split the data within a block into multiple tiles, which map to a single thread. This means a single thread works on a small contiguous block of memory, which is part of the original data block, and all threads in the thread block work in parallel to encode a single data block.

The size of a tile has an effect on both the compression ratio and the compression throughput. To create parallelism and indirectly improve throughput, a smaller tile size is ideal. However, symbols that overlap tile borders will not be recognized as a single symbol, but instead will be split into two or more smaller symbols. Furthermore, a table block size that is too small will not be able to capture repeating patterns that can be compressed. For that reason, table generation prefers a bigger block size. To uncouple these conflicting requirements, we use the concept of *super tables*. This means multiple data blocks will use the same encoding table. This allows us to modify the data block size to better suit the GPU, while continuing to use a (larger) block size for table generation.

5.2 Inter-block dependencies

Compression of data inherently suffers from several sequential dependencies, which prevent parallel execution. Since we use a *static* symbol table, the only relevant dependency is determining the output location for every block. At the start of compression, it is not yet known what the resulting compressed size of each block will be, so it is not possible to calculate where each block should start depositing its output. This dependency forces a sequential execution order between blocks.

This can be mitigated by the use of padding characters. We pad the output blocks to their worst-case size. This ensures the output location is fixed for all blocks. Padding ensures there is no overlap between blocks and removes the inter-block dependency, allowing for parallel execution.

However, the use of padding necessitates an additional post-processing stage that removes said padding. This defines the basic structure of our compression pipeline: we begin by generating tables, proceed with the encoding kernel, and conclude with data compaction during post-processing. We will go into more depth about the post-processing stage in Section 6, but we can already define the interface for the encoding kernel: it encodes the given data blocks using a dedicated thread block into fixed locations in global memory.

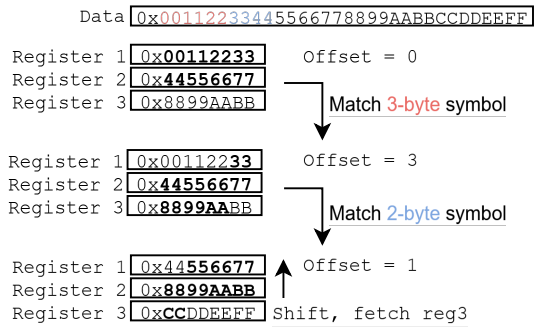


Figure 4: The process of using a sliding window to build a view of the active data, which can be used by the encoding kernel to directly match on. In this example, we show how data moves through the registers as the data in shared memory is processed. Bold numbers are used to show what part of the data is part of the current view.

5.3 Sliding window

The main encoding loop consists of reading data from global memory, encoding it, and writing it to global memory. Because of the repeated random access, we use temporary buffers in shared memory, which have limited space. For this reason, every thread performs multiple encoding cycles, which consist of reading a small chunk from global to shared memory, encoding it, and storing the result in shared memory (and flushing when required). This loop is repeated until the entire tile has been processed.

As mentioned before, a naive implementation performing byte-level operations leads to many bank conflicts and underutilizes the shared memory banks, which are capable of 32 bits per clock cycle. We mitigate this by requesting 32 bits, or four characters, at a time from shared memory, and we also organize the input buffer as a column-major matrix. This means we view the input data for a thread block as a NxM matrix, where N represents the number of threads (or tiles) within a thread block and M the number of 4-byte integers representing the data of a single tile. Shared memory will then contain a $X * N$ matrix, where X represents the chunk size. All data for a single tile will be stored in a column in this matrix, completely eliminating bank conflicts.

This greatly simplifies the encoding cycles, as we now deal with 32-bit words, but also introduces a problem: a symbol can span multiple words and might not consume a full 32-bit word. In order to evaluate multiple (partial) words, we introduce the sliding window.

The sliding window uses three 32-bit registers and keeps track of the reading offset to create a view of the next eight bytes. The effect of the sliding window can be seen in Figure 4. In Listing 2, we show how to create a view. We also keep track of the spillover from the previous encoding cycle, as a symbol might overlap chunk borders. When a register is fully encoded, indicated by the offset, we shift the registers once and fetch the next 32-bit number.

```
uint64_t create_view(uint32_t first_word, uint32_t second_word,
                uint32_t third_word, uint8_t offset, uint8_t len) {
    uint8_t b_from_first = min(len, 4 - offset);
    uint8_t b_from_second = min(len - b_from_first, 4);
    uint8_t b_from_third = min(len-(b_from_first+b_from_second), offset);

    uint64_t first_data = get_first_n(
                            first_word >> offset * 8, b_from_first);
    uint64_t second_data = get_first_n(second_word, b_from_second);
    uint64_t third_data = get_first_n(third_word, b_from_third);

    return first_data | second_data << b_from_first * 8 |
        third_data << (b_from_first + b_from_second) * 8;
}
```

Listing 2: Sliding window view creation using three registers and an offset

```
void pack_results_local(uint32_t result[out_buf_size][thread_count],
                  uint32_t offset, uint32_t val) {
        uint32_t shift = (offset & 3) * 8; // n-byte within word
        uint8_t block_index = offset / 4; // Identify block
        uint32_t val_mask = val << shift;
        uint32_t clean_mask = ~(0xFF << shift);

        uint32_t current = result[block_index][threadIdx.x];

        result[block_index][threadIdx.x] = current & clean_mask
                                           | val_mask;
    }
```

Listing 3: The output packing process

5.4 Output packing

The sliding window addresses the issue of memory alignment on the input side of the encoding loop, but we have a similar problem with our output data. Every match iteration of an encoding cycle produces one or two bytes, depending on whether the symbol needs an escape character. This is not naturally aligned to 4-byte boundaries, so we need to perform output packing. The process in Listing 3 allows us to set individual bytes in a 32-bit number, which allows us to use an efficient array of 32-bit numbers as if it were an array of 8-bit numbers.

5.5 Ensuring coalesced writes

Up until now, we have defined our tiling approach, kernel interfaces, and main encoding loop, including the sliding window and output packing. However, we have yet to define a solution for possibly the two biggest challenges: inter-tile dependencies and memory coalescing. Just as is the case with blocks, the output lengths of tiles are not known beforehand and have a sequential dependency. Furthermore, the effective memory bandwidth has a significant impact on our overall performance, which means memory coalescing is necessary.

We will address both issues at the same time with the final part of the encoding kernel: collaborative output writing. In order to achieve coalesced writes, we will use a transposed output format. This means the output data can be seen as a YxN matrix with 32-bit words, where N is the number of threads within a thread block and Y is the number of output words per thread. To achieve coalesced memory transactions, all threads within a warp have to perform

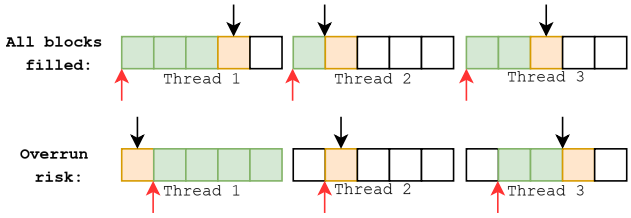


Figure 5: Threads keep track of their own local buffer head (marked with black arrows, on byte level), and their working block (marked orange) and filled blocks (marked green). All threads keep track of the active block in the warp (marked with red arrows, on block level). Threads in a warp will decide to flush in two scenarios: when all threads have filled the currently active block with data, or when a thread can potentially overrun the buffer in the next encoding iteration.

writes in the same row at the same time, hence the *collaborative* part.

We achieve this using a voting system within warps using the `ballot` functionality[2], and a thread-local circular buffer. The overall process is illustrated in Figure 5. Every thread has its own circular output buffer and keeps track of its local head and the currently active block. The local head is used in the output packing process, and is specifically for that thread and refers to a *byte* location. The currently active block is shared by all threads within a warp and refers to the 4-byte word that is the next block to be flushed.

After every iteration in the encoding cycle, threads will hold a vote on whether to initiate a flush or not. If any thread risks overrunning its buffer, all threads will add padding to their local buffer if needed and trigger a flush. A flush will also be triggered if all threads have filled the currently active block, which is the ideal scenario. After the last encoding cycle has completed, a warp will continue flushing its buffers until all threads within a warp have fully written their data. Additionally, all warps within a block will communicate such that they perform the same number of overall flushes to create a valid output matrix.

This method ensures all write transactions are coalesced and also eliminates the sequential inter-thread dependency. Imbalances in compression output between threads as a result of different local compression ratios are no longer an issue due to this voting process.

6 COMPRESSION PIPELINE

In this section, we will describe our full pipeline in more detail and provide several optimizations that take full advantage of the capabilities of modern GPUs. We will also discuss our compatibility with the existing GSST decompressor.

6.1 Gathering data

As mentioned before, our pipeline consists of three steps: table generation, encoding, and post-processing. The post-processing step involves removing padding between data blocks, i.e., gathering the results from every thread block.

[2]https://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-vote-functions

We can employ one of two techniques to gather the resulting data from thread blocks. We could perform stream compaction on the entire data stream, removing the special padding symbol between blocks. This would be the best option if the padding were interleaved throughout the data. However, our balloting scheme outputs dense data, i.e., the inter-block padding is not interleaved but at the end of the output block instead. This means using direct memory copies also becomes an option. Instead of performing stream compaction on the entire data stream, the CPU would trigger a device-to-device memory copy for every block, which can be significantly faster.

6.2 Improving output format

The compression pipeline is now complete, but still has two issues. Both issues are caused by the transposed format. The first issue is that we (partially) lose FSST's ability to perform random access decompression. Since consecutive bytes are not guaranteed to belong to the same tile anymore, random access decompression would become significantly more complex. The second issue is that the compression ratio will be lower than that of FSST. This is because of the special padding introduced by the collaborative output writing when a thread forces a flush because of a potential buffer overflow. This padding cannot be removed without creating an invalid matrix with rows of different lengths. Even though we consider the output matrix to be filled with 32-bit numbers, this does not matter for the underlying memory. Removing a single byte will cause the decompressor to interpret the data incorrectly.

We can fix the first issue by performing a transposition operation on the output data of a block. This orients the output data in a NxY format, which is more in line with the output format of FSST and the input format of the GSST decompressor. Since this transposition is on the block level, we can use dynamic parallelism to achieve pipelining. This means we can use idle resources on the GPU, which are likely to be there at the end of the encoding process, to transpose the output data in parallel with encoding other data blocks.

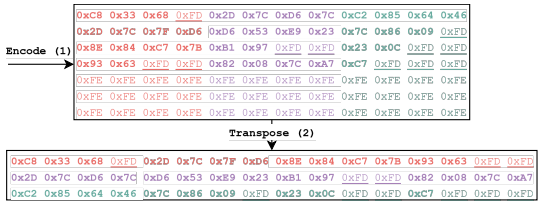
In addition to this, we can now fix the second issue by performing stream compaction on the transposed data to remove the interleaved padding. Since the encoded data for a single tile is now in contiguous memory, we no longer need to maintain identical output lengths for all tiles.

These improvements are expected to give a high compression ratio and transform the output format such that it is compatible with the GSST compressor. We will investigate the performance characteristics of the pipeline stages in Section 7.3.

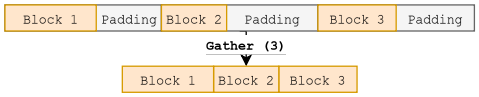
6.3 Optimized pipeline

Our final pipeline now consists of five distinct stages. We first create the encoding tables on the CPU, which we use to encode our input data on the GPU. We then transpose the output data of every individual data block to undo the effect of our coalesced writes. Once all data has been transposed, we gather the resulting data into a single contiguous block of memory using device-to-device memory copies. Finally, we perform stream compaction to filter out interleaved padding. The pipeline is shown in Figure 6.

Memory usage is an important aspect of compressors, which is sometimes overlooked. This is especially the case on GPUs, where memory is still a scarce resource. To minimize the required amount



(a) After table generation and encoding, the relevant data is transposed such that all data from a single tile is in contiguous memory. Note that we omitted the padded data in the transposed data for the sake of brevity.



(b) After all blocks have been encoded and transposed, we gather all data into a contiguous block of memory by using device-to-device memory copies. This eliminates the intra-block padding.

(c) We perform stream compaction on the entire data stream to eliminate interleaved padding, which is a result of the balloting system.

Figure 6: A simplified overview of our GPU-accelerated compression pipeline. All data belonging to the same tile has the same color. Note that the first two stages of encoding and transposition operate on the block level, and the final two stages of gathering and compaction operate on the entire data stream.

of memory to compress the data, we carefully use a temporary buffer and make use of the fact that we have multiple sequential memory transformations. Figure 7 shows how we use the temporary buffer with the memory transformations to swap data between buffers. We encode the input data to a temporary buffer, which we then transpose to the output buffer. Since the temporary buffer is now unused, we use it as the target buffer for our gathering stage. After the gather operation has completed, the output buffer is no longer used, so we directly perform our stream compaction from the temporary buffer to the output buffer. Additionally, we copy our generated headers to the output buffer during compaction. This ensures that we only need a single additional buffer to compress the data, reducing our overall memory usage. We will compare our memory usage to state-of-the-art compressors in Section 7.4.

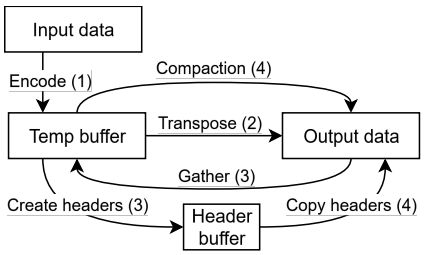


Figure 7: The data flow through the temporary and destination buffers in our pipeline. The overall memory usage is low because we reuse the temporary and destination buffers for multiple operations.

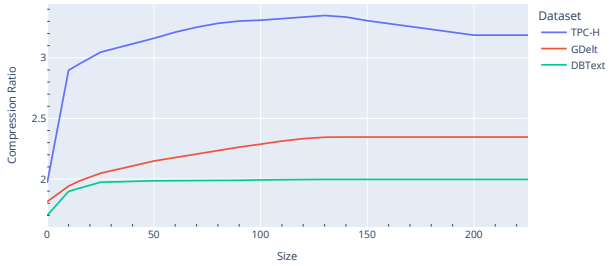


Figure 8: The effect of varying the number of entries in the hashtable on the resulting compression ratio. It is clear that the hash table can be smaller without sacrificing significant accuracy.

6.4 Ensuring compatibility with GSST

As we mentioned in Section 2.2, GSST [36] introduced a GPU decompressor but lacks a high-throughput compressor. We will discuss the technical details of integration in this section.

GSST works by applying tiling to the FSST algorithm and providing some metadata about the tiles, or splits, as the authors of GSST call them. The tiles have a constant uncompressed length, and the header is slightly modified to include the compressed length of each tile. This allows the decompressor to identify the exact starting locations of each tile. This work division matches our tiling approach. To make our compressor compatible with the GSST decompressor, every thread will have to write its output length, excluding padding, to the block header.

The GSST decompressor also has to be slightly modified, as we write all headers to the start of the file as opposed to the start of each data block. This is because we perform stream compaction on the entire data stream, which requires that all data is in contiguous memory. However, this should not be a problem because the relevant table can be retrieved fairly easily as long as the decompressor keeps track of which data block it is decompressing.

7 EVALUATION

In this section, we will evaluate the performance of our compression pipeline and compare it to the state-of-the-art. We will use the same datasets as analyzed in Section 4.1, and perform our tests on a system with an RTX 4090 and a Ryzen 9 9950X (16 hardware cores, 32 threads). We use CUDA 12.8 and the NVIDIA driver 570.133.20, in combination with *nvidia-smi* to gather usage data. All code was compiled in release mode with the highest optimization settings.

We will compare our performance in terms of compression throughput and compression ratio with the nvCOMP library from NVIDIA, GPULZ [38], and compressors generated with the LC framework [3]. For GPULZ, we use three configurations: fast, average, and max-compression, which match the configurations based on the original authors' parameter sweep of `(C=4096, W=32, S=4)`, `(C=4096, W=128, S=2)`, and `(C=4096, W=255, S=1)`, respectively. For LC, we generate compressors with one, two, and three stages. The throughput measurements are performed on data in GPU memory.

7.1 Encoding table performance

In Section 4.2 and 4.3, we introduced the modified hashtable and the new match table, respectively. Both have the goal to encode the same, or at least close to the same, amount of information while using less memory.

Figure 8 shows the effects of reducing the hashtable size. We can see that a size of 128 is sufficient for all datasets to reach their compression ratio. Datasets like DBText that do not contain many long symbols will require even less.

To determine the effects of a maximum number of rows and a maximum number of entries within a row, we performed a parameter sweep using these two parameters. As a baseline, we used the average usage by the regular FSST algorithm, which can be found in Section 4.1. The results of this experiment can be found in Figure 9.

Remember that in the match table format, the number of allowed rows must be a multiple of 32. Based on our experiments, we can say that 32 rows will be enough. The maximum number of entries in a single row must be a multiple of two, and the average usage for all datasets is between 2.4 and 6.5. When limiting the number of rows to 32, using more than eight columns results in a negligible increase in compression ratio for the textual datasets. DBText is the exception, since it heavily uses the shortcodes structure. Using eight columns as a baseline results in similar compression ratios as FSST, while only suffering an acceptable 5 percent decrease for machine-readable data.

These parameters indirectly affect the throughput of the encoding kernel by changing how much shared memory is needed. This influences the occupancy of our encoding kernel, which can potentially change the overall performance. When using the parameters above, we use $1024 + 12 * 128 = 2560$ bytes for the hashtable and $32 * 8 * 2 + 256 = 768$ bytes for the lookup table, a reduction of 84 percent and 99 percent compared to FSST, respectively.

Figure 10 shows the effect on overall pipeline throughput as a result of higher shared memory usage when modifying the number of columns. When combined with the effect on the compression ratio shown in Figure 9, these results suggest that a throughput reduction of approximately 3 percent across all datasets leads to an

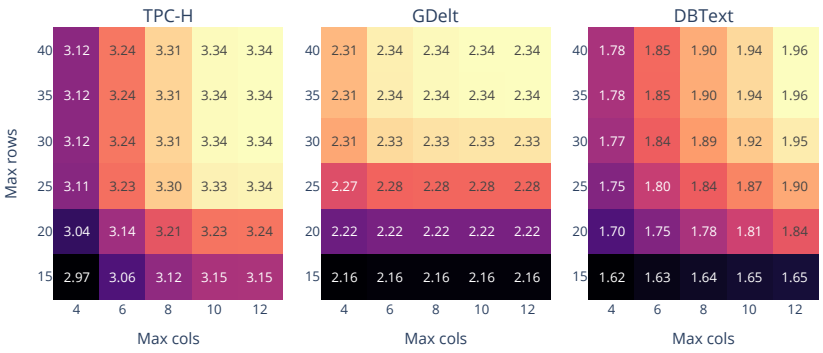


Figure 9: The effect of varying the number of rows and columns in the lookup table on the resulting compression ratio.

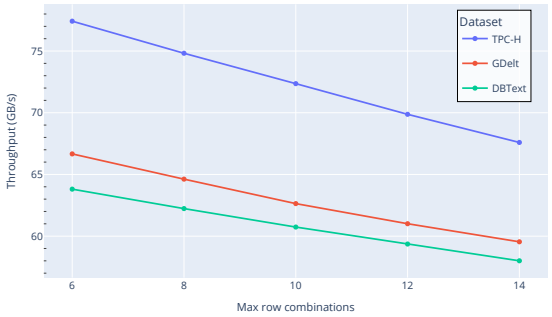


Figure 10: The effects on overall throughput of changing the maximum number of columns, as a consequence of lower occupancy.

approximate 2 percent increase in compression ratio for machine-readable data.

7.2 Accelerated compression throughput and ratio

The goal of this compressor is to accelerate the original FSST algorithm beyond what a multi-threaded CPU application can achieve and at least match PCIe throughput, while maintaining FSST's excellent compression ratio on string data. We performed a parameter sweep to determine the optimal work division and throughput, which we will elaborate on more in Section 7.3.

Figure 11 compares the achieved compression throughput and ratio of our compression pipeline to the state-of-the-art. We use 2GB files for all our datasets to ensure there is enough work to process, while ensuring that none of the compressors run out of memory.

We can make some observations from these results. In terms of compression, we outperform ANS, Bitcomp, Cascaded, and GPULZ consistently for all datasets. For TPC-H and DBText, we achieve slightly higher compression ratios than LZ4 and Snappy, while they have a higher compression ratio for the GDelt location data.

When considering overall compression throughput, we outperform GPULZ and all nvCOMP compressors with the exception of ANS, Bitcomp, Cascaded, and all compressors generated with the LC framework. This ranges from a 2.8x increase when compared to Snappy to a 7.9x increase when compared to ZSTD.

Overall, our compressor is part of the Pareto front for every dataset, meaning we push the state-of-the-art further towards ideal compression.

We also added the results for the original FSST paper to the graph to compare overall compression ratios. We achieve nearly identical compression ratios, except for the machine-readable data as explained in Section 7.1, while achieving a speedup of 50.27x. Even when compared to a multithreaded CPU implementation, we achieve a 7.43x speedup.

7.3 Performance analysis per stage

Our pipeline consists of several stages, most notably the encoding and the compaction stage, which consists of gathering data from all thread blocks and then filtering out interleaved padding. Remember that we apply tiling to achieve parallelism, which influences the amount of data per thread and therefore has a significant effect on the overall throughput.

First, more data per thread results in fewer data blocks (and thread blocks) overall. This is beneficial for the compaction stage, since fewer blocks mean fewer, but larger, device-to-device copies. This is also the case when increasing the number of warps per thread block, as the compaction stage initiates a single copy per thread block.

On the contrary, the encoding stage needs a certain number of thread blocks to operate at full throughput. When the number of active thread blocks is too low, the occupancy of the kernel is low, and several Streaming Multiprocessors will be idle.

Figure 11: We compare our proposed compression pipeline to the nvCOMP compressors, GPULZ, compressors generated with the LC framework with a different number of stages, and the original FSST algorithm, regarding compression ratio and throughput. All benchmarks were completed on the same machine (RTX 4090 with a Ryzen 9 9950X) and used the same 2GB files. Our compressor, marked with an orange star, is a Pareto point in all datasets.

Overall, the tile size influences both the occupancy and the efficiency of the compaction stage, so we expect to see opposite behaviour in terms of performance between these two stages. Figure 12 shows the throughputs of the two stages and the combined overall throughput. This figure confirms our reasoning.

This means that, ideally, the tile size dynamically grows with the input file size to always have the highest overall throughput. Furthermore, more warps per thread block have a positive effect on the compaction stage, but a negative effect on the encoding stage due to increased shared memory pressure. It might be possible to avoid the negative effects and further increase performance by using cooperative groups[3].

7.4 GPU memory consumption

As we mentioned in Section 6.3, we also considered memory usage to be an important aspect of a GPU compressor, as memory is a scarce resource and excessive usage can significantly reduce overall performance and the ability to handle large datasets [15].

To measure the memory usage for every compressor, we ran the provided benchmark code for each and measured the memory consumption for the process using `nvidia-smi`. We then subtract the size of the input and output buffer to get the memory used by the compressor itself.

[3]https://docs.nvidia.com/cuda/cuda-c-programming-guide/#cooperative-groups

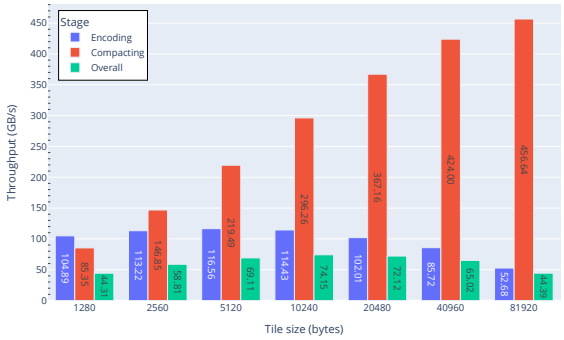


Figure 12: The throughput for the two major stages (encoding and compaction) and how they are influenced by the tile size.

Our compressor re-uses buffers several times to minimize memory consumption, and the results of that effort can be found in Figure 13. We use significantly less memory than all other compressors, as we only require a single additional buffer in addition to some working memory for metadata and temporary header storage.

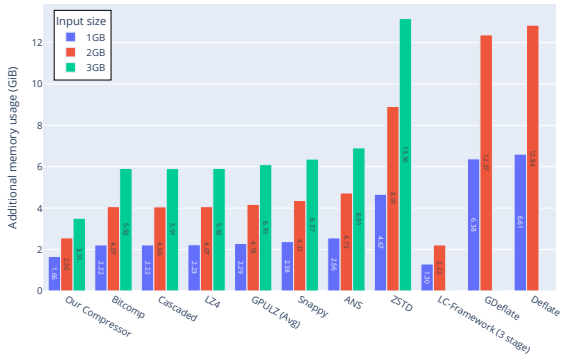


Figure 13: Memory usage, measured with `nvidida-smi`, excluding input and output buffers of our compression pipeline, compared to other state-of-the-art compressors. The compressors are sorted based on their memory usage, and empty columns indicate that a compressor ran out of memory or failed to compress the file.

8 CONCLUSION AND FUTURE WORK

In this paper, we introduce a GPU-accelerated compressor based on the FSST table generation algorithm. By optimizing both the encoding kernel and the overall compression pipeline, we efficiently exploit the massive parallelism of GPUs. Our results show that we achieve compression ratios comparable to LZ-based algorithms, while significantly improving throughput over existing GPU implementations. While some GPU-native compressors like ANS, Bitcomp, and other floating-point compressors like SPspeed still achieve considerably higher throughputs, we offer a higher compression ratio for textual data.

This positions our compressor as a Pareto optimal compressor that can be used in GPU-accelerated database systems and other areas where large amounts of textual data need to be efficiently compressed with competitive compression ratios.

In the future, we would like to fully incorporate the GSST decompressor to provide full end-to-end measurements and perform a complete evaluation.

Also, as mentioned in Section 7.3, it would be ideal to scale the tile size with input size. In this version of the pipeline, a static size is used, which limits the performance portability for different file sizes. Furthermore, cooperative groups might prove to be useful to further increase the throughput of the compaction stage, without negatively affecting the encoding performance.

REFERENCES

[1] Azim Afroozeh and Peter Boncz. 2023. The fastlanes compression layout: Decoding> 100 billion integers per second with scalar code. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2132–2144.

[2] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–11.

[3] Noushin Azami, Alex Fallin, Brandon Burtchell, Andrew Rodriguez, Benila Jerald, Yiqian Liu, and Martin Burtscher. 2024. *LC Git Repository*. https://github.com/burtscher/LC-framework

[4] Noushin Azami, Alex Fallin, and Martin Burtscher. 2025. Efficient Lossless Compression of Scientific Floating-Point Data on CPUs and GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 395–409.

[5] Markus Billeter, Ola Olsson, and Ulf Assarsson. 2009. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the conference on high performance graphics 2009*. 159–166.

[6] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[7] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. 2021. Fpga acceleration of zstd compression algorithm. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 188–191.

[8] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H Peter Hofstee. 2019. Refine and recycle: A method to increase decompression parallelism. In *2019 IEEE 30Th international conference on application-specific systems, architectures and processors (ASAP)*, Vol. 2160. IEEE, 272–280.

[9] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. 2015. Fast LZW compression using a GPU. In *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 303–308.

[10] GDELT. [n.d.]. *The GDelt project*. https://www.gdeltproject.org/

[11] Jens Glaser, Felipe Aramburú, William Malpica, Benjamín Hernández, Matthew Baker, and Rodrigo Aramburú. 2021. Scaling SQL to the Supercomputer for Interactive Analysis of Simulation Data. In *Smoky Mountains Computational Sciences and Engineering Conference*. Springer, 327–339.

[12] Jens Glaser, Josh V Vermaas, David M Rogers, Jeff Larkin, Scott LeGrand, Swen Boehm, Matthew B Baker, Aaron Scheinberg, Andreas F Tillack, Mathialakan Thavappiragasam, et al. 2021. High-throughput virtual laboratory for drug discovery using massive datasets. *The International Journal of High Performance Computing Applications* 35, 5 (2021), 452–468.

[13] Maya Gokhale, Jonathan Cohen, Andy Yoo, W Marcus Miller, Arpith Jacob, Craig Ulmer, and Roger Pearce. 2008. Hardware technologies for high-performance data-intensive computing. *Computer* 41, 4 (2008), 60–68.

[14] ELLPACK Group. 1985. *ELLPACK - Software for Solving Elliptic Problems*. https://www.cs.purdue.edu/ellpack/ellpack.html

[15] Laiq Hasan, Marijn Kentie, and Zaid Al-Ars. 2011. DOPA: GPU-based protein alignment using database and memory access optimizations. *BMC research notes* 4 (2011), 1–11.

[16] Benjamín Hernández, Suhas Somnath, Junqi Yin, Hao Lu, Joe Eaton, Peter Entschev, John Kirkham, and Zahra Ronaghi. 2020. Performance evaluation of python based data analytics frameworks in summit: Early experiences. In *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI: 17th Smoky Mountains Computational Sciences and Engineering Conference, SMC 2020, Oak Ridge, TN, USA, August 26-28, 2020, Revised Selected Papers 17*. Springer, 366–380.

[17] David Meirion Hughes, Ik Soo Lim, Mark W Jones, Aaron Knoll, and Ben Spencer. 2013. Ink-compact: In-kernel stream compaction and its application to multi-kernel data visualization on general-purpose gpus. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 178–188.

[18] Jeff Johnson. 2022. *DietGPU: GPU-based lossless compression for numerical data*. https://github.com/facebookresearch/dietgpu

[19] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. Ndzip-gpu: Efficient lossless compression of scientific floating-point data on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[20] Fangzheng Lin, Kasidis Arunruangsirilert, Heming Sun, and Jiro Katto. 2023. Recoil: Parallel rans decoding with decoder-adaptive scalability. In *Proceedings of the 52nd International Conference on Parallel Processing*. 31–40.

[21] Li Lu and Bei Hua. 2019. G-Match: a fast GPU-friendly data compression algorithm. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 788–795.

[22] NVIDIA. [n.d.]. *nvCOMP library*. Retrieved March 17, 2025 from https://developer.nvidia.com/nvcomp

[23] NVIDIA. 2024. *nvCOMP performance*. Retrieved March 18, 2025 from https://web.archive.org/web/20240225035645/https://developer.nvidia.com/nvcomp

[24] NVIDIA. 2024. *Nvidia Blackwell Architecture Technical Overview*. Retrieved March 17, 2025 from https://resources.nvidia.com/en-us-blackwell-architecture

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

[26] Adnan Ozsoy and Martin Swany. 2011. CULZSS: LZSS lossless data compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*. IEEE, 403–411.

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

[28] Adnan Ozsoy, Martin Swany, and Arun Chauhan. 2014. Optimizing LZSS compression on GPGPUs. *Future Generation Computer Systems* 30 (2014), 170–178.

[29] Jeongmin Park, Zaid Qureshi, Vikram Mailthody, Andrew Gacek, Shunfan Shao, Mohammad AlMasri, Isaac Gelado, Jinjun Xiong, Chris Newburn, I-hsin Chung, et al. 2023. CODAG: Characterizing and Optimizing Decompression Algorithms for GPUs. *arXiv preprint arXiv:2307.03760* (2023).

[30] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. 2012. *Parallel lossless data compression on the GPU.* IEEE.

[31] Anil Shanbhag, Bobbi W Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based lightweight integer compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data*. 1390–1403.

[32] K Shyni and Manoj Kumar KV. 2013. Lossless LZW data compression algorithm on CUDA. (2013).

[33] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A Ross. 2016. Massively-parallel lossless data decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 242–247.

[34] Adam Thompson and CJ Newburn. 2019. *GPUDirect Storage: A Direct Path Between Storage and GPU Memory.* Retrieved March 17, 2025 from https://developer.nvidia.com/blog/gpudirect-storage

[35] TPC. [n.d.]. *TPC-H Version 2 and Version 3.* https://www.tpc.org/tpch/

[36] Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. 2025. GSST: Parallel string decompression at 191 GB/s on GPU. In *Proceedings Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (Rotterdam, Netherlands). ACM.

[37] Zheng Wang, Chenxi Wang, and Lei Wang. 2023. Dpubench: An application-driven scalable benchmark suite for comprehensive dpu evaluation. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 3, 2 (2023), 100120.

[38] 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 *Proceedings of the 37th International Conference on Supercomputing*. 348–359.

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

[40] Yuan Zu and Bei Hua. 2014. GLZSS: LZSS lossless data compression can be faster. In *Proceedings of Workshop on General Purpose Processing Using GPUs*. 46–53.