GSST

Parallel string decompression at 191 GB/s on GPU

Vonk, Robin; Hoozemans, Joost; Al-Ars, Zaid

DOI
10.1145/3759441.3759450

Publication date
2025

Document Version
Final published version

Published in
Operating Systems Review (ACM)

GSST: Parallel string decompression at 191 GB/s on GPU

Robin Vonk
Delft University of Technology
Delft, The Netherlands
research@robinvonk.com

Joost Hoozemans
Voltron Data
worldwide remote, US
joosthooz@gmail.com

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

Abstract

Most of the commonly used compression standards make use of some form of the LZ algorithm. Decompressing this type of data is not a good match for the Single-Instruction, Multiple Thread (SIMT) model of computation used by GPUs, resulting in low throughput and poor utilization of the GPU parallel compute capabilities. In this paper, we introduce GSST, a GPU-optimized version of the FSST compression algorithm, which targets string compression. The optimizations proposed in this paper make the algorithm particularly suitable for GPUs, which allows it to achieve a significantly better tradeoff for decompression throughput vs compression ratio as compared to the state of the art. Our results show that the new algorithm pushes the Pareto curve closer towards the ideal region, completely dominating LZ-based compressors in the nvCOMP library (LZ4, Snappy, GDeflate). GSST provides a compression ratio of 2.74x and achieves a throughput of 191 GB/s on an A100 GPU.

ACM Reference Format:
Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. 2025. GSST: Parallel string decompression at 191 GB/s on GPU. In *5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3719330.3721228

1 Introduction

As the throughput gap between storage media and computational device speeds narrows down, it becomes increasingly important to ensure the high throughput of compression algorithms, as indicated by a number of recent studies [7, 8]. In this evolving landscape, achieving a high compression ratio (CR) is becoming less critical for overcoming the bottlenecks caused by slow interconnects. Instead, the focus has shifted to the decompression process, which can now take longer than transferring data to the computation device.

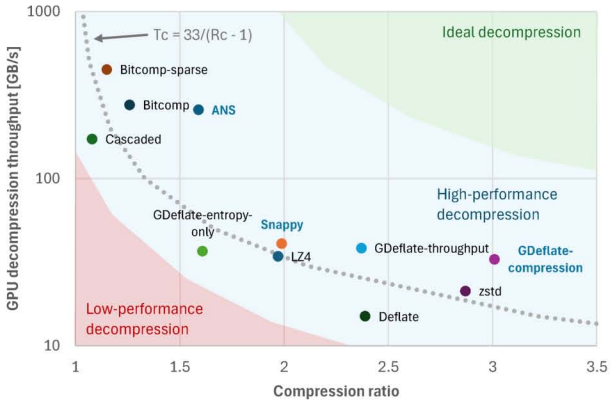


Figure 1. Decompression throughput and compression ratio of GPU compression algorithms included in the nvCOMP library using the measurements on the nvCOMP website [10] on the A100 GPU for the Silesia dataset

High throughput and parallel compression algorithms have become a hot research topic in the high-performance computing (HPC) community. Traditionally, compression algorithms have been designed to maximize CRs and optimize single-core throughput. However, with the stagnating single-core performance improvements, hardware manufacturers are now enhancing performance through multicore processors and accelerators such as GPUs. This shift necessitates the development of algorithms that can leverage these new hardware capabilities.

To benefit from the rapid increase in the throughput achievable on GPU accelerators, multiple compression algorithms have been ported and optimized to GPUs, achieving various throughput-CR tradeoff points.

Figure 1 plots the benchmark results of GPU compression algorithms included in the nvCOMP library using the measurements on the nvCOMP website [10] running on the NVIDIA A100 GPU for the Silesia dataset [13]. Achieving both high throughput and high CR at the same time has been a formidable challenge on GPUs, especially for string-based datasets. Decompression of the commonly used LZ-style compression algorithms (used for example in Deflate,

CHEOPS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1529-7/25/03
https://doi.org/10.1145/3719330.3721228

Snappy, LZ4, and zstd) proves to be a poor match for GPUs, because of their sequential nature [12].

This paper presents a GPU accelerated string (de)compression algorithm called GSST, based on the FSST (Fast Static Symbol Table) compression format [3]. This paper discusses a number of modifications we made to the FSST algorithm to benefit from the abundant parallelism GPUs provide. The optimizations proposed to the algorithm include the addition of sub-block metadata, that allows thread parallel decoding of the sub-blocks (a technique similar to the two-level parallelism used in GDeflate [15]). Then, we present a number of optimizations in the implementation, including:

- Thread balancing during the compression stage, to minimize thread imbalances during decompression
- The use of local scratchpad storage for symbol table and input/output data blocks
- Asynchronous data movement between the main memory and local scratchpads

Results show that the proposed GSST algorithm significantly pushes the state-of-the-art in GPU decompression performance.

This paper is organized as follows. Section 2 introduces the FSST compression layout, while Section 3 details the GSST decompression implementation and its modifications compared to FSST. In Section 5, we describe the experimental setup used for benchmarking, followed by the performance results presented in Section 6, where we compare GSST with FSST and the nvCOMP algorithms.

2 FSST compression

The FSST compression format works by replacing frequently occurring substrings (symbols) of lengths 1 to 8 bytes with fixed-size 1-byte codes. This process involves creating a static symbol table based on the input data, where each code maps to a corresponding symbol. This symbol table is limited in size to 255 symbols, and so can be indexed using a single byte.

Figure 2 shows the compression process of FSST. During compression, FSST scans the input string, identifies the longest matching symbol from the table, and substitutes them with their respective codes. When there is no symbol describing a part of the input string, a special escape code is used to store individual bytes. An escape code indicates that the next byte should be interpreted as data, and not as a code. Finally, the symbol table is stored together with the sequence of codes.

In decompression, FSST performs a lookup to convert each 1-byte code back to its original substring using the static symbol table. The static nature of the symbol table enables FSST to support random access to individual compressed strings, allowing efficient decompression without needing to process entire data blocks. This makes FSST particularly suitable for database systems where rapid access to specific string attributes is essential.

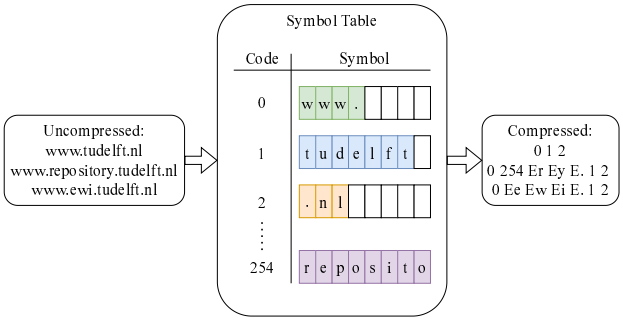


Figure 2. Demonstration of FSST compression. During the compression, the input data is scanned for repeated patterns. These are inserted in the symbol table. During the compression process, data is replaced by (1-byte) codes from the symbol table. Data patterns that cannot be found in the symbol table, are prepended by an escape code (here shown as **E**). The escape code allows storing individual raw bytes.

3 The GSST compression format

3.1 Tiling parallelism

A commonly exploited method of achieving parallelism during compression and decompression, is by dividing the data into blocks and compressing each block of data independently. This method is referred to as tiling, chunking, or block compression [1, 14]. Increasing the number of blocks enhances parallelism but reduces block size, which can negatively affect the compression ratio. Since each block is compressed independently, the algorithm cannot exploit repeated data patterns that extend across multiple blocks.

Our proposed GSST algorithm increases the parallelism of FSST by adding block-based compression. Our implementation uses OpenMP on the CPU and CUDA on the GPU, supports user-configurable block sizes and is able to handle edge cases such as escape sequences on a block boundary.

3.2 Splits parallelism

Decompression of data suffers from two sequential dependencies, preventing its parallel execution: 1. the output of compressed data depends on other decompressed data, and 2. the position of decompressed data depends on the decompressed size of the data before it. Because of the *static* symbol table, the FSST format does not suffer from the first limitation; all symbols can be decompressed independently of each other. Only the second limitation needs to be solved to achieve full parallel decompression. As the symbols can have different lengths, additional information is required to know where the output data corresponding to a certain offset in the compressed input data must be stored.

GSST stores additional metadata during compression, that contains information about the uncompressed size of *subsections inside a block* (called splits), allowing the decompression to determine the corresponding output locations. In a sense, this adds another level of hierarchy to the block-level parallelism, with the difference being that each *block* has its own symbol table while all the *splits* inside a block make use of the same symbol table (and can therefore employ SIMT parallelism). The amount of splits in a block determines the level of thread-level parallelism. More splits can result in higher throughput, but each split requires a value in the added metadata. Using a large number of splits with a relatively small block size can negatively affect the achieved compression ratio.

3.3 Splits layout

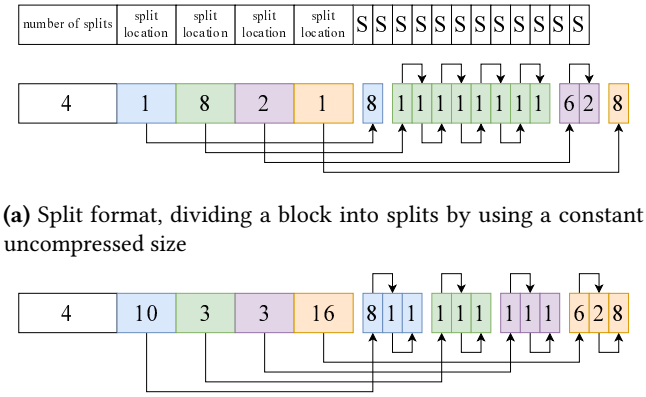
Here, we define the layout of the metadata for carrying the sub-block split information. Splitting the symbols in a block can be done in two ways:

1. Constant uncompressed size, variable split compressed size: As illustrated in Figure 3a, this format stores the offsets/locations in the compressed data (= sizes of the compressed splits) where a constant amount of data has been compressed. During decompression, each thread reads a location from the header and starts decompressing at that location. Because of the constant output size of each compressed split, each thread can calculate the output location of their data by multiplying the split index by the constant uncompressed size.
2. Variable uncompressed size, constant split compressed size: As illustrated in Figure 3b, this format divides the compressed data into constant sized splits. The uncompressed size of each split is stored in the header. During decompression, each thread can start decompressing at a multiple of the constant compressed size and writes the resulting data at the offset/location from the header.

We use a length encoding approach for storing split locations in the header, similar to the strategy described in [9]. By leveraging the fact that each split location begins after the previous one, only the difference between consecutive splits needs to be stored. This method reduces the storage requirements for metadata. However, it necessitates pre-processing, specifically summing the preceding locations, before decompression can begin.

4 Memory alignment optimization

To fully make use of the high bandwidth of GPU memory, using aligned memory accesses is highly desirable. There are two memory operations during decompression that will benefit the most from aligned accesses.



(a) Split format, dividing a block into splits by using a constant uncompressed size

(b) Split format, dividing a block into splits by using a constant compressed size

Figure 3. Two formats for ordering symbols in a compressed block. The arrow direction (→) refers to thread execution order. A format consists of a header with metadata followed by a sequence of symbols. The format describes the order in which the symbols are stored. In this example, the symbols are divided over four splits, storing four values as metadata in the header. The number in a symbol depicts the uncompressed length of that symbol. Format 3a stores the compressed length of each split, and each split has uncompressed size 8. Format 3b stores the uncompressed length of each split, and each split has compressed size 3.

1. Reading compressed data from global memory to shared memory (L1 cache)
2. Writing decompressed data from shared memory to global memory

Decompression of a split starts with aligning both the input and output pointers in global memory. The memory accesses are shown in Figure 4. First, the input pointer is aligned by using unaligned reads until the first aligned address in the input buffer is reached. Then this data is decompressed and written to global memory using unaligned writes until the first aligned address in the output buffer is reached. This leads to two possible scenarios:

1. Aligning the input buffer generated enough output data to align the output pointer. In this case, copy the amount of data needed to align the output pointer to global memory.
2. Aligning the input buffer did not generate enough data to align the output pointer. In this case, first decompress all the data from aligning the input and write it to global memory. Then read the first aligned input block, decompress it, and write data to global memory until the output is aligned.

In both cases, we can now move the remaining output data in shared memory to the beginning of the buffer, and continue decompressing until the output buffer is full. The full

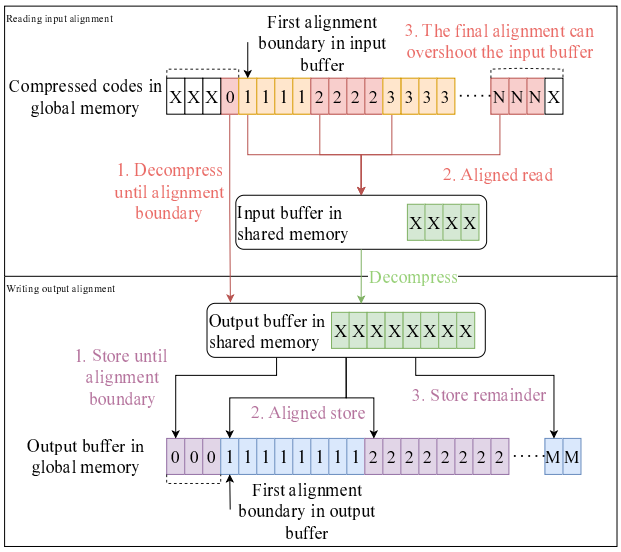


Figure 4. Achieving aligned transfers in an unaligned compressed buffer on a 4 byte-boundary for the compressed input data and on an 8 byte-boundary for the decompressed data by (1) reading, decompressing and writing individual bytes until the first aligned input and output addresses are reached. (2) Then, aligned read and write operations are used. (3) Finally, overshoot of the input data needs to be handled, and the remainder data in the output buffer needs to be stored.

buffer can now be moved to global memory using aligned instructions.

5 Experimental setup

The experiments are executed on a dual-socket machine with two Intel Xeon Platinum 8380 CPUs. The memory is 32x 128GB running at 3200MHz, with a total of 4TB. The node also has 8 NVIDIA A100 GPUs with 80GB device memory. In addition, the node has an 8 lane network interface running at 200GbE and a Samsung PM9A3 storage system with 8x 15TB storage.

We compare the performance of GSST against the nvCOMP library from NVIDIA. nvCOMP version 3.0.6 has support for the classical compression formats LZ4, Snappy, Deflate, and zstd. In addition, ANS, Bitcomp, Cascaded, and GDeflate are provided with GPU-optimized formats. To identify the optimal configuration for the algorithms, a sweep is conducted with various chunk sizes. In the results section below (Section 6), we report the highest throughput achieved by each algorithm for the most optimal chunk size. The throughput measurements are performed based on data that resides in GPU memory. In other words, we are not taking I/O performance and PCIe transfer times into consideration. The dataset is initially generated by TPC-H's dbgen tool. From the lineitem table, we extracted the column named

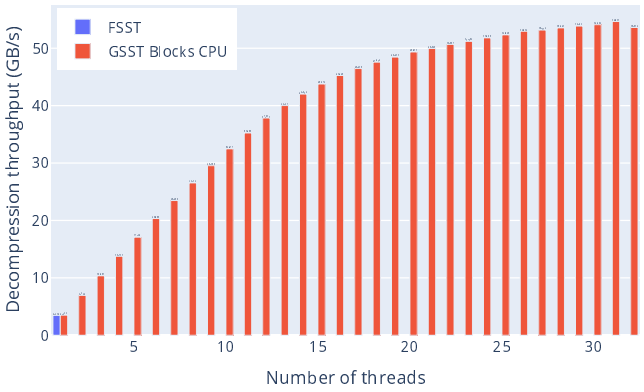


Figure 5. Decompression throughput of FSST and the CPU implementation of GSST block-level parallelism. The benchmark involves decompressing a 1.5 GB text file, generated by dbgen, which has been compressed into 1024 blocks. The benchmark measures decompression speed of decompressing data from an in-memory input buffer with compressed data to an in-memory output buffer.

"comment". To ensure sustainable throughput, we ensure our dataset size is 10GB.

6 Results

6.1 CPU performance

The first implementation of GSST utilizes block-level parallelism on the CPU with OpenMP. Figure 5 compares the decompression throughput to FSST. As we increase the number of threads, decompression throughput grows by about 3 GB/s up to 14 threads. Beyond that point, the performance each thread delivers begins to decline, and after 32 threads, adding more threads actually reduces overall throughput. Looking at compression ratios, FSST achieves 2.71, where GSST reaches 2.74 when using 1024 blocks. Despite GSST adding extra metadata, it still has a slightly better ratio. This difference can be attributed to the amount of data compressed per symbol table. FSST processes 4 MB of uncompressed data per symbol table, while GSST uses about 1.46 MB in this configuration. Depending on the dataset, using a different amount of data per symbol table can improve the compression ratio.

6.2 GPU decompression throughput and compression ratio

We begin by examining the progressive optimization of our GSST decompression kernels on the GPU. Figure 6 presents the achieved decompression throughput for five key implementations, each building incrementally on the one before it. The initial *GSST Blocks Kernel* suffers from low GPU utilization due to limited threads per block. Switching to the *GSST Splits Kernel* improves utilization by using all threads, but global memory stalls limit compute throughput. Introducing shared memory shifts the bottleneck to shared memory stalls

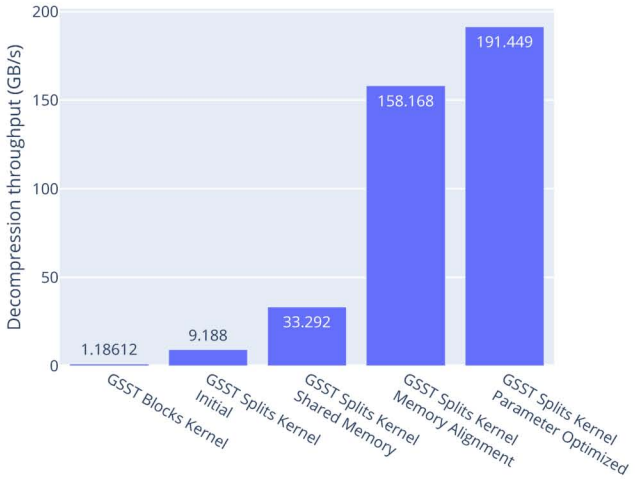


Figure 6. Decompression throughput achieved by each GSST GPU implementation.
GSST Blocks Kernel introduces block parallelism, as explained in Section 3.1.
GSST Splits Kernel Initial is the first implementation utilizing the split format, described in Section 3.2.
GSST Splits Kernel Shared Memory enhances this by incorporating shared memory for data staging.
GSST Splits Kernel Memory Alignment adds aligned memory transfers, as covered in Section 4.
GSST Splits Kernel Parameter Optimized is the same implementation as the memory-aligned splits but using the fastest parameters found by sweeping parameters.

and reveals misalignment issues, where each byte written triggered a 32-byte cache-line transaction. Aligning memory accesses greatly increases memory and compute throughput, with the largest bottleneck being stalls waiting for shared memory. Finally, tuning the parameters (number of blocks, number of splits, number of bytes per alignment, and shared memory buffer size) reduced these stalls and achieved the highest decompression throughput.

Figure 7 compares the decompression throughput and compression ratio of GSST against the nvCOMP algorithms. From these results, the following observations can be provided:

1. GSST significantly outperforms traditional CPU-based formats in nvCOMP, such as LZ4, Snappy, and zstd, when it comes to decompression throughput. **GSST achieves a decompression throughput of 191GB/s, which is only surpassed by ANS and Bitcomp in nvCOMP**. While these two algorithms have a 49% and 46% higher decompression throughput compared to GSST, respectively, GSST offers a 49% and 151% higher compression ratio compared to ANS and Bitcomp.

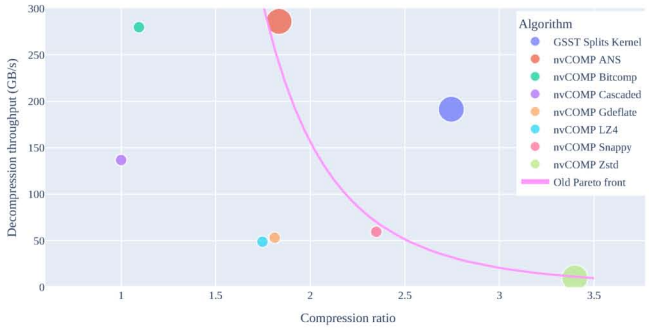


Figure 7. Comparison of decompression throughput and compression ratio of GSST against nvCOMP algorithms on an NVIDIA A100, using the 10GB TPC-H text dataset. The decompression throughput measures the time it takes for compressed data in GPU memory to be decompressed to GPU memory. The large dots represent the new Pareto front. GSST shifts the Pareto front, offering a favorable balance of both decompression throughput and compression ratio.

2. Compression Ratio: When compressing string data, **GSST achieves a compression ratio of 2.74, outperforming all nvCOMP algorithms except zstd**. Although zstd surpasses GSST with a 23% higher compression ratio, GSST makes up for this with its decompression throughput, which is 18.6x faster than zstd.

Overall, GSST is well-positioned as an optimal solution, offering a strong trade-off between compression ratio and decompression throughput for string data. Note that bitcomp and ANS may still be able to perform better on numerical data, but that is not the focus of GSST.

6.3 GPU memory usage

Memory usage is an important factor when evaluating compression algorithms, especially on GPUs, where excessive memory consumption can slow down processing, limit throughput, and prevent large datasets from being handled effectively [6]. This makes memory usage an important metric for assessing how well compression algorithms perform on large files.

In this analysis, memory usage was measured for GSST and all nvCOMP compression algorithms using the 10GB text file. The memory usage during decompression is recorded using NVIDIA Nsight Systems with the *-cuda-memory-usage=true* parameter. The exact values are extracted from the SQLite export of the profile.

The memory usage measurements are shown in Figure 8. A remarkable observation was the high memory consumption of nvCOMP algorithms, particularly due to a large buffer allocation when creating the nvCOMP compression manager. Regardless of the algorithm, the manager allocated a buffer twice the size of the input data (i.e. 20GB for our 10GB input),

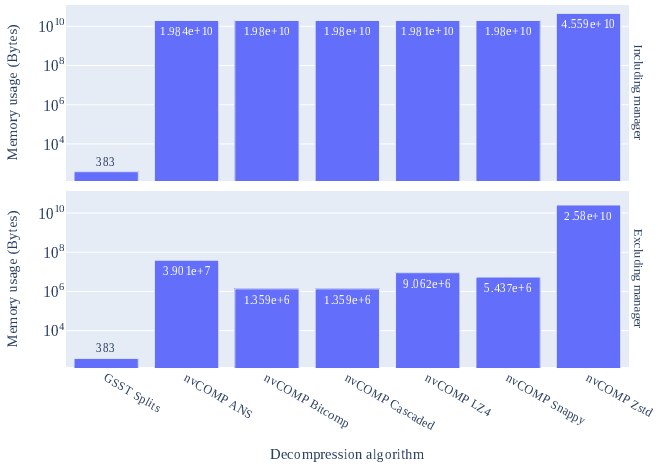


Figure 8. Peak GPU memory usage of GSST compared to the algorithms in the nvCOMP library when decompressing the 10GB TPC-H text dataset on the A100 80GB, as reported by NVIDIA Nsight Systems. This only includes the memory usage from initializing the nvCOMP compression manager and calling the decompression algorithm. This does not include the input buffer with compressed data and output buffer for the decompressed data. Gdeflate is not included, as the benchmark crashed when launched by Nsight Systems. The nvCOMP manager in the high-level API allocates a buffer of twice the input size before any compression or decompression is done. The top graph shows the memory usage including the memory allocated by the nvCOMP manager, and the bottom graph without it.

before any compression or decompression is started. This overhead contributed to significantly higher memory usage. While this benchmark used nvCOMP's high-level API, it is possible that using the low-level API could reduce these large memory allocations.

GSST demonstrates a significant advantage over nvCOMP in terms of memory efficiency. By keeping the needed data, such as the symbol tables, in shared memory, the algorithm requires almost no global memory. As stated before, this is mainly the result of nvCOMP using excessive amounts of global memory. As nvCOMP is closed source, it is unclear why it allocates such large amounts of memory. While the low-level API might help reduce memory consumption, it is unlikely to bring memory usage down to the same levels as GSST.

7 Related work

Most traditional compression algorithms predate GPU accelerators and lack parallel-friendly designs, whereas newer big data formats increasingly emphasize parallelization by design. A SIMD-friendly integer encoding layout introduced in [1] facilitates parallel decoding of schemes like delta and run-length encoding, which was later implemented on GPUs [2]. Alongside software-based optimizations, the push for higher decompression throughput has spurred the development of dedicated hardware accelerators for popular encoding formats, such as NVIDIA's data processing units (DPUs) with built-in decompression [16] and upcoming GPU generations featuring specialized decompression engines [11], as well as those developed on FPGAs [4, 5].

8 Conclusion

This paper discusses a new algorithm for string compression based on FSST that targets fast decompression throughput on GPUs. By making use of a pre-defined static symbol table and adding metadata to expose additional parallelism, our proposed GSST algorithm allows efficient use of the GPUs SIMT model of computation. Results show that the algorithm performs significantly better in terms of both decompression throughput and compression ratio compared to most LZ-based algorithms that have GPU implementations. That means that those algorithms are rendered obsolete when working with string data on GPUs.

This new GPU compression landscape has only 3 Pareto-optimal trade-off points for string decompression on GPUs:

- zstd provides the highest compression ratio at 3.40x but the decompression throughput is limited to 9.8GB/s.
- ANS provides the highest decompression throughput, reaching 286GB/s at a compression ratio of 1.83x.
- GSST provides a balanced tradeoff with a compression ratio of 2.74x and a throughput of 191GB/s.

GSST achieves a high compression ratio with high decompression throughput, allowing for efficient data transfer and processing on GPUs. It also delivers the lowest memory usage by minimizing global memory overhead and utilizing shared memory, making it highly effective for handling large datasets with minimal resource consumption.

References

[1] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (May 2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[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* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3662010.3663450

[3] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (01 07 2020), 2649–2661. https://doi.org/10.14778/3407790.3407851

[4] 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)* (2021-06-01). IEEE Computer Society, 188–191. https://doi.org/10.1109/IPDPSW52791.2021.00035

[5] 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)* (2019-07), Vol. 2160-052X. 272–280. https://doi.org/10.1109/ASAP.2019.00017 ISSN: 2160-052X.

[6] Laiq Hasan, Marijn Kentie, and Zaid Al-Ars. 2011. DOPA: GPU-based protein alignment using database and memory access optimizations. 4, 1 (2011), 261. https://doi.org/10.1186/1756-0500-4-261

[7] Byungseok Kim, Jaeho Kim, and Sam H. Noh. 2017. Managing Array of SSDs When the Storage Device Is No Longer the Performance Bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotstorage17/program/presentation/kim

[8] Fritz Kruger. 2016. CPU Bandwidth – The Worrisome 2020 Trend. (23 March 2016). https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/

[9] Lennart Noordsij, Steven van der Vlugt, Mohamed A. Bamakhrama, Zaid Al-Ars, and Peter Lindstrom. 2020. Parallelization of Variable Rate Decompression through Metadata. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 245–252. https://doi.org/10.1109/PDP50117.2020.00045

[10] NVIDIA. 2024. nvCOMP library. https://developer.nvidia.com/nvcomp

[11] NVIDIA. 2024. NVIDIA Blackwell Architecture Technical Brief. https://resources.nvidia.com/en-us-blackwell-architecture

[12] Jeongmin Park, Zaid Qureshi, Vikram Mailthody, Andrew Gacek, Shunfan Shao, Mohammad AlMasri, Isaac Gelado, Jinjun Xiong, Chris Newburn, I hsin Chung, Michael Garland, Nikolay Sakharnykh, and Wen mei Hwu. 2023. CODAG: Characterizing and Optimizing Decompression Algorithms for GPUs. arXiv:2307.03760 [cs.DC]

[13] Sebastian Deorowicz. 2024. Silesia Compression Corpus. https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

[14] 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* (New York, NY, USA, 2022-06-11) *(SIGMOD '22)*. Association for Computing Machinery, 1390–1403. https://doi.org/10.1145/3514221.3526132

[15] Yury Uralsky. 2024. *Accelerating Load Times for DirectX Games and Apps with GDeflate for DirectStorage*. https://developer.nvidia.com/blog/accelerating-load-times-for-directx-games-and-apps-with-gdeflate-for-directstorage/

[16] 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. https://doi.org/10.1016/j.tbench.2023.100120