GSST

High Throughput Parallel String Decompression on GPU

Robin Vonk

TU Delft

GSST
High Throughput Parallel String Decompression on GPU

by

Robin Vonk

Project duration:	October 2023 – September 2024	
Thesis committee:	Dr.ir. Z. Al-Ars,	TU Delft, supervisor
	Dr.ir. R. Hai,	TU Delft
	Dr.ir. J. Hoozemans,	Voltron Data
	Dr.ir. J. Peltenburg,	Voltron Data

This thesis was developed with the assistance of QuillBot, DeepL, GPT-4o, and LanguageTool for drafting, paraphrasing, and grammar checks. However, all interpretations, conclusions, and arguments are my own, and I take full responsibility for the content of this work.

Abstract

This thesis describes how the throughput of data ingestion on GPUs can be increased by using data compression. This is done through two main contributions. First, a high-level model is presented to assess the impact of compression on ingestion throughput. Second, a novel decompression algorithm called GSST (GPU Static Symbol Table) is developed, optimized for GPU parallelism. GSST achieves state-of-the-art performance, striking an effective balance between compression ratio and decompression throughput.

The work done in this thesis has contributed to the submission of two scientific papers:

1. GSST: Parallel string decompression at 150 GB/s on GPU [1] (to be updated to 191 GB/s upon submission).
2. Benchmarking GPU Direct Storage for High-Performance Filesystems: Impact & Future Trends [2]

The ingestion throughput model is introduced to quantify the impact of data compression on data ingestion on a GPU. This model offers insight into how the compression ratio and decompression throughput influence overall data ingestion performance. This model shows that, as storage devices become faster, the decompression algorithm must also increase its throughput to keep up, while the compression ratio becomes less influential on the ingestion throughput.

GSST is the solution proposed to increase data ingestion throughput on GPUs. GSST adapts from the FSST (Fast Static Symbol Table) algorithm to the parallel architecture of GPUs. GSST's performance is driven by six performance optimizations. Three format optimizations, block parallelism, split parallelism, and coalesced memory access, increase parallelism and throughput by changing the way data is stored. Additionally, three memory management techniques are implemented to effectively utilize the memory throughput of a GPU. These are the use of shared memory, using aligned memory accesses, and utilizing asynchronous data transfers.

Using the ingestion throughput model, GSST is evaluated against the state-of-the-art GPU compression algorithms from nvCOMP. The results reveal that GSST achieves a decompression throughput of 191 GB/s with a compression ratio of 2.74 on an A100. While nvCOMP's ANS and Bitcomp outperform GSST in decompression throughput, they offer lower compression ratios. Similarly, Zstd achieves a higher compression with significantly lower decompression throughput, positioning GSST as a good balance of decompression throughput and compression ratio.

The data ingestion model demonstrates that GSST offers the highest ingestion throughput among the tested compression algorithms when ingesting data over a connection with a throughput between 0.8 GB/s and 87 GB/s. This means GSST is ideally suited for use with top-of-the-line networking equipment and even provides headroom for future improvements in connection throughput. Additionally, GSST is extremely memory-efficient, using significantly less GPU memory than all of nvCOMP's compression algorithms. In some cases, GSST uses 3,500 times less memory, and in the best scenarios, over 67 million times less.

By leveraging these format and memory management optimizations, GSST provides a powerful, efficient solution for industries using large-scale data systems such as high-performance computing and data analytics.

The GSST source code will be made available on GitHub [3].

Contents

1 Introduction

In today's digital age, the amount of data generated is increasing at a rate never seen before. As more sensors and humans are connecting to the internet, the amount of data increases with it [4]. With the increase in data comes the need for more data storage and processing throughput.

New applications such as query engines [5], machine learning models [6], and blockchain nodes [7] utilize GPUs to be able to process these large amounts of data. The increase in data and the processing of it comes with major advantages for society. In healthcare, GPUs increase the speed and reduce the cost of diagnostics [8], cancer research [9], and new drug discovery [10]. For our climate, GPUs are used to accurately understand carbon emissions and effectively combat them [11]. In research, GPUs help discover new materials [12], simulate physics [13], reconstruct DNA [14], and much more.

The processing throughput of GPUs has been steadily improving with each new generation, leading to new possibilities [15]. However, the technology for storing and transferring data to these GPUs hasn't advanced as quickly. This mismatch creates a bottleneck, where the GPU has to wait for data to arrive, slowing down its performance. Compression techniques are being used to help close this gap by reducing the amount of data that needs to be moved, allowing the GPU to work more efficiently [16].

1.1. Context

1.1.1. Compression

Compression is the process of reducing the size of data or files by eliminating redundancies or using more efficient encoding. It allows information to be stored or transmitted using less space or bandwidth. The goal of compression is to trade in computation time to reduce data size. This makes compression a tool to alleviate strain on another weaker component in the system, for example to optimize storage space or speed up data transfer.

Lossless compression is the technique used to reduce the size of data without losing any information, ensuring that the original data can be perfectly reconstructed from the compressed version. Although it typically results in less size reduction compared to *lossy compression*, it guarantees that no data is lost. The focus of this thesis is limited to lossless data compression.

1.1.2. Computation Bottlenecks

In recent years, processor performance has significantly improved, continuing to follow the trend predicted by Moore's law, which has been notable in both CPUs and GPUs [17]. However, the performance of memory and storage devices connected to these processors has not kept up, creating a gap that can constrain overall system performance. This gap is called the memory wall. As a result, processors often spend considerable time idle while waiting for data to be retrieved from memory or storage.

One approach to mitigate the impact of the memory wall is through data compression [18]. Compression techniques can reduce the volume of data that needs to be transferred between memory and processors, thereby effectively increasing bandwidth and decreasing latency. By compressing data before storing it in memory or on storage devices, systems can accelerate data processing and retrieval, alleviating some bottlenecks associated with slower memory speeds.

The impact of the memory wall has begun to diminish in recent years due to several trends [19], [20]. Advances in storage technologies, such as NVMe SSDs and persistent memory, are helping to narrow the performance gap. Additionally, the rise of parallel processing allows systems to access multiple storage devices simultaneously, distributing the workload and reducing latency.

As these trends continue, there is growing interest in exploring compression algorithms that prioritize speed, as they may better align with the faster storage technologies now emerging. The idea is that as storage devices become quicker, compression algorithms also need to be faster to keep up, ensuring that data processing remains efficient. Additionally, in parallel and distributed systems, there is a need for compression techniques that can operate effectively in parallel.

1.1.3. (Big) Data Query Engines

Database systems and query engines play an important role in data management and storage efficiency. These systems are designed to handle large amounts of data and execute complex queries to retrieve information. However, as data volumes grow, the performance of these systems can become a bottleneck. Inefficient data retrieval processes can lead to increased computational load and higher energy consumption.

An important process in a database system is the *Data ingestion*. It is the collecting, importing, and processing of data from various sources. For the purposes of this thesis, data ingestion is specifically defined as the process of reading and decompressing data.

Modern database systems employ various techniques to improve performance, including indexing, partitioning, and caching. However, these methods alone may not suffice as data continues to grow. Correctly integrating data compression techniques within database systems can greatly increase their efficiency [21], [22]. Compressed data occupies less storage space and requires less bandwidth for transmission, allowing databases to process queries with lower resource consumption.

1.2. Challenges

1.2.1. GPUs in Data Processing

In recent years, Graphics Processing Units (GPUs) have evolved beyond their original purpose of rendering graphics, emerging as alternatives to Central Processing Units (CPUs) for a variety of data processing tasks. Initially designed for handling complex visual computations, GPUs are now recognized for their superior capability in executing parallel processing workloads. This parallelism makes GPUs particularly advantageous for data-intensive applications, such as machine learning, scientific simulations, and data analytics.

Compared to CPUs, GPUs offer several distinct advantages in terms of processing throughput and energy efficiency. Their architecture is optimized for executing multiple operations concurrently, which enables significant speedups for tasks that can be parallelized, such as compute-bound applications as machine learning. GPUs typically have higher memory bandwidth, making them also effective for memory-bound applications such as data analytics[23].

New advancements have expanded the scope of GPU usage from merely acting as a CPU accelerator to functioning as an independent processing unit itself. Applications such as Theseus [5], BlazingDB [24], Bitcoin miners [7], and machine learning models [6] now use the GPU as the primary processing device, with the CPU playing a secondary, coordinating role. Innovations such as GPU Direct Storage (GDS) [25], NVLink [26], Compute Express Link (CXL) [27], and Open Coherent Accelerator Processor Interface (OpenCAPI) [28] further reduce the need for GPU-CPU interaction, decreasing the GPU's dependency on the CPU.

The increasing pervasiveness of the GPU comes with new needs and opportunities. GPUs offer significant computation throughput and have the potential to be more energy efficient. However, to fully leverage this potential, traditional algorithms must be redesigned. Since the single-core performance of the GPU is much lower than that of the CPU, parallelism is needed to gain an advantage.

1.2.2. GPU Data Ingestion Bottleneck

A common challenge in GPU processing is the data transfer bottleneck. A recent study by Rosenfeld on the current state and challenges of query engines on GPUs highlights that improving data transfer speeds or reducing the amount of data transferred can have a much bigger impact on performance than simply increasing the computational throughput of GPUs [29]. The issue arises from slow connections between the GPU and other system components, which prevent the full utilization of the GPU’s computational capabilities.

This bottleneck occurs because GPUs rely on a continuous flow of data to maintain their high processing speed. While GPUs are designed to handle large amounts of parallel computations quickly, they often end up sitting idle, waiting for data to be transferred from other parts of the system, such as the CPU or main memory. The main cause of this inefficiency is the limited bandwidth of interconnects, like PCIe (Peripheral Component Interconnect Express), that handle the data exchange between the GPU and the rest of the system. As a result, even though GPUs are capable of processing data at incredible speeds, the slow transfer rates are a major performance bottleneck.

To address this issue, one effective solution is to optimize data transfer by either reducing the amount of data being moved or increasing transfer speed. The study emphasizes compression as a valuable technique for improving query performance in this context. Compression provides two main benefits:

1. It reduces the amount of data that needs to be transferred and stored, trading some computation time for decompressing data in exchange for faster overall transfer speeds.

2. Certain operations can be performed directly on compressed data, which can further reduce the time spent on computation.

When data is compressed before being transferred, the GPU can use its bandwidth more efficiently, which helps reduce the delays caused by a slow connection. However, as storage devices and interconnects become faster, the need for highly efficient compression to address this issue decreases.

Figure 1.1 illustrates how the time is spent across the three stages of data ingestion into a GPU. It highlights that with modern, high-speed storage devices, data can now be transferred faster than the GPU can decompress and decode it. Compression still speeds up data transfer, as demonstrated when comparing the transfer time of uncompressed data with data compressed using Snappy. However, the time it takes to decompress with Snappy ends up being longer than the time saved by compressing the data, which leads to an overall increase in transfer time.

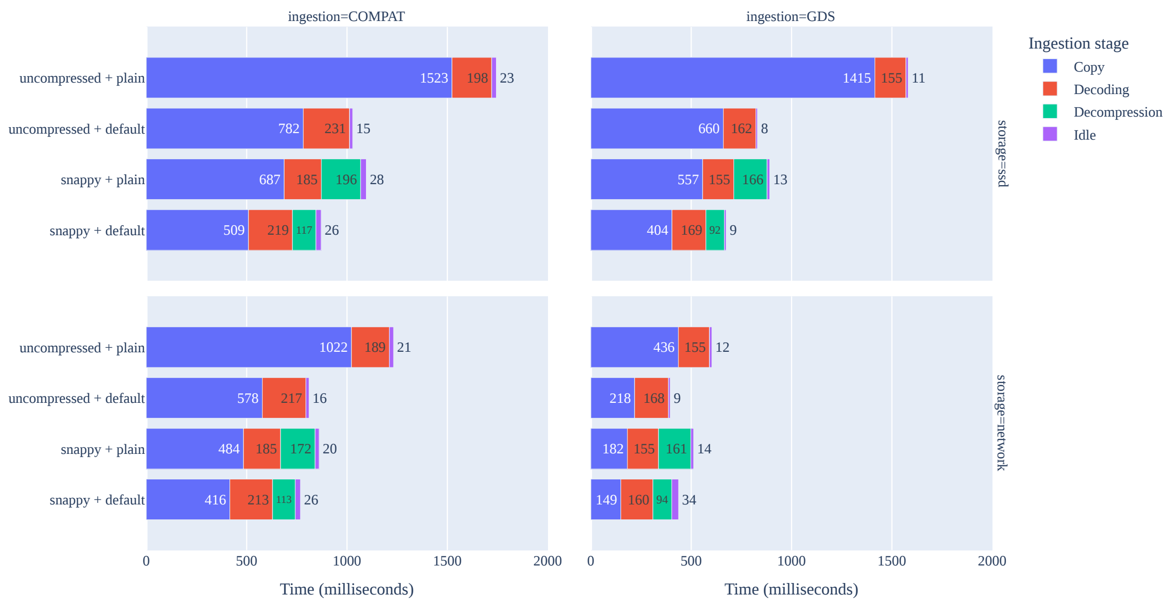
This shift suggests that the bottleneck is now moving to the decompression process, emphasizing the need for fast, high-throughput compression algorithms that prioritize throughput over achieving the highest possible compression ratios.

1.2.3. Compression on GPUs

Data compression on GPUs has a history that goes back to the evolution of computer graphics. Initially, GPUs were designed for rendering graphics, focusing on processing large volumes of image data. Early graphics required efficient methods to handle textures, images, and 3D models without overloading memory bandwidth. As the quality of these visuals increased, so did the need for compressing this data. Early efforts in data compression on GPUs centered around techniques like DirectX Texture Compression (DXT), developed in the late 1990s [32]. DXT helps to reduce texture size while maintaining quality, allowing games and applications to run more efficiently without requiring significant amounts of video memory.

In the early 2000s, GPUs began to evolve beyond graphics rendering, becoming more general-purpose computing devices, thanks to architectures like NVIDIA’s CUDA and AMD’s GPGPU technology. As GPU architectures became more flexible and programmable, developers began to offload compression algorithms traditionally handled by CPUs to GPUs. Compression techniques like JPEG, video codecs, and even lossless compression schemes started to leverage the parallel processing throughput of GPUs [33].

In recent years, GPUs have expanded into a broader range of general-purpose computing tasks, as introduced in Section 1.2.1. Beyond traditional parallelizable tasks, GPUs are now employed for diverse data-intensive applications like neural networks, databases, and various forms of data analytics [34]. This brings the need for new compression algorithms specialized to these new data types, as traditional GPU compression methods often fail to work efficiently on these data types. For example, [35] reports that "decompression can consume approximately 91% of total GPU time on a modern data analytics pipeline". This highlights the urgency for compression formats and algorithms to evolve with both GPU architectures and the used data types to address the new types of data.



Source: [2]

Figure 1.1: Comparison of time distributions when loading data from a Parquet file into GPU memory using the *read_parquet* function in CuDF [30]. This uses the test setup described in Table 5.1. The data comes from two types of storage: network storage, which is much faster, and SSD storage. In the COMPAT mode, the data passes through the CPU memory, whereas in GDS mode, it bypasses the CPU memory altogether. The tests are conducted with and without Snappy compression, and with both plain and default encoding, as outlined in the Parquet format specification [31]. The results highlight how, as storage throughput improves, the bottleneck shifts from data transfer to the processes of decompressing and decoding the data.

1.3. Research Questions

This thesis addresses the main question:

How can compression increase data ingestion throughput in GPU processing?

This question will be answered in two steps, first with an exploration, followed by a design. The exploration will answer the following sub-questions:

1. What are the bottlenecks in data ingestion on a GPU?
   Identifying and modelling the limitations and inefficiencies in the current data ingestion processes.
2. Which CPU compression formats have the potential for high performance on GPUs?
   Evaluating and analyzing compression formats and techniques to identify those that can be effectively ported to GPU architectures for enhanced performance.

After the exploration, the focus will shift towards the decompression on the GPU, designing a new implementation of a decompression algorithm with the right trade-offs to speed up data ingestion. The following sub-questions will be answered:

3. What strategies can be used to effectively distribute data across the parallel processing units in the GPU?
   Investigating methods for splitting and distributing data to ensure utilization of all parallel processing units.
4. How can memory access patterns be optimized to reduce memory latency during decompression?
   Investigating techniques to improve memory access patterns that align with the GPU memory hierarchy.
5. What are optimal parameters for the GPU decompression algorithms?
   Measuring throughput and compression ratio, and tuning algorithm parameters for optimal performance on GPU hardware.

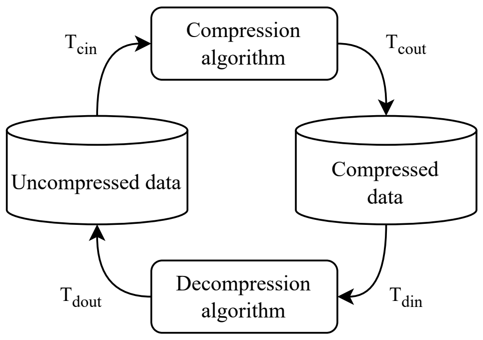


Figure 1.2: Diagram of the compression throughput cycle, showing how data moves through both the compression and decompression algorithm. The throughput refers to the amount of data processed by the algorithm per unit of time, both as input and output. However, during compression and decompression, the data size changes, meaning the input and output throughput are typically different.

1.4. Evaluation

A compression algorithm compresses a file of uncompressed size S_u to a compressed size S_c. The ratio between these two sizes is a factor called the compression ratio, calculated using Equation (1.1). The time from calling the compression algorithm until the algorithm finishes is the compression time t_c. Likewise, the time from calling the decompression until it is done is the decompression time t_d. A widely used metric for comparing (de)compression algorithms is the throughput or rate. This measure quantifies the amount of data that can be (de)compressed per second. As the (de)compression algorithm changes the size of the data, there are two throughputs per algorithm. There is a throughput of data going into the algorithm and the throughput of data going out. The relation between these throughputs is the CR. The mathematical symbols used for these throughputs are shown in Table 1.1

	Compression	Decompression
Input throughput	T_{cin}	T_{din}
Output throughput	T_{cout}	T_{dout}

Table 1.1: Four symbols are used for (de)compression throughput of data going into the algorithm and going out.

$$CR = \frac{S_u}{S_c} \tag{1.1}$$

Equations (1.2) and (1.3) are used to calculate the throughput of a compression algorithm. These formulae are preferred, as they only take into account how fast an algorithm compresses or decompresses a buffer of a given size. Contrary to using the compressed data size (S_c), which would be influenced by the CR as defined in Equations (1.4) and (1.5)

$$T_{cin} = \frac{S_u}{t_c} \tag{1.2}$$

$$T_{dout} = \frac{S_u}{t_d} \tag{1.3}$$

$$\begin{aligned} T_{cout} &= \frac{S_c}{t_c} \\ &= \frac{S_u}{CR \cdot t_c} \quad \text{(substitude Equation (1.1))} \\ &= \frac{T_{cin}}{CR} \end{aligned} \tag{1.4}$$

$$\begin{aligned} T_{din} &= \frac{S_c}{t_d} \\ &= \frac{S_u}{CR \cdot t_d} \quad \text{(substitude Equation (1.1))} \\ &= \frac{T_{dout}}{CR} \end{aligned} \tag{1.5}$$

In Chapter 3, these formulae will be further extended to a model, used to quantify the impact of compression algorithms.

This thesis investigates how compression can increase data ingestion throughput in GPU processing. Since string data is estimated to make around 80-90% of all data, the evaluation will focus on compression of string data [36], [37]. By targeting this data type, the research effectively aims to address how compression can enhance the efficiency of data ingestion in GPU processing. TPC-H format data, generated using dbgen, will be used to test the algorithms in a large-scale processing environment [38].

1.5. Outline

First, in Chapter 2, the state of the art in data compression on CPU and GPU will be discussed. Second, a model is defined which is used to quantify the performance of compression on data ingestion in Chapter 3. After understanding how compression works, its bottlenecks, and how it can be improved, new formats will be introduced in Chapter 4. In Chapter 5 the testing setup and parameters of the algorithm are discussed. Chapter 6 presents the benchmark results and compares them to the state of the art. Finally, the results are concluded, and future work is described in Chapter 7.

2 Related Work

2.1. Compression Techniques

Compression can be achieved using different approaches and techniques. These can be categorized into lightweight compression (sometimes referred to as encoding) and heavyweight compression.

2.1.1. Lightweight Compression

Lightweight compression algorithms are designed to efficiently reduce the size of data while using minimal computational resources. These algorithms prioritize speed and simplicity, making them suitable for devices with limited processing power and memory. They achieve compression by identifying and encoding patterns in the data, reducing redundancy without heavily taxing system resources. Lightweight compression is often used in embedded systems, mobile devices, and real-time applications where performance and resource efficiency are critical.

- Run Length Encoding (RLE): Run-Length Encoding compresses data by replacing sequences of the same data value, known as runs, with a single data value followed by a count of how many times it is repeated. This is particularly useful for images or data streams where repetition occurs frequently.
  Example:
  - Original data: AAAAAABBBBCCCC
  - Compressed data: A6B4C4
- Delta Encoding: Delta Encoding compresses data by storing the difference (or delta) between consecutive elements rather than storing each value in its entirety. This is particularly useful when dealing with sequences of data that change gradually, such as time-series data or numerical sequences.
  Example:
  - Original data: 33333 33334 33335 33332
  - Compressed data: 33333 1 1 -3
- Frame of Reference (FOR): Frame of Reference compression works by selecting a base value, typically the minimum or first value in a sequence, and then encoding each subsequent value as an offset relative to that base value. This method is useful for compressing sequences of numbers that vary within a limited range.
  Example:
  - Original data: 33333 33334 33335 33332
  - Compressed data: 33333 1 2 -1

- Bit packing: Bit Packing compresses data by combining multiple smaller values into a single larger value, using the minimum number of bits necessary to represent the data. This technique is especially useful when dealing with datasets containing small integers that can be packed together into fewer bits.
  Example:
  - Original data: 0000000000000001 0000000000000010 0000000000000011 0000000000000100 (each represented as 16-bit integers)
  - Compressed data: (8-bit integers) 00000001 00000010 00000011 00000100
    or packing 2 values in 1 byte: 000010010 00110100
- Dictionary Encoding (DICT): Dictionary Encoding replaces repeated occurrences of data patterns with shorter codes. A dictionary is created where common patterns are stored once, and references to these patterns are used throughout the data. This approach is widely used in text compression, where common words or phrases can be replaced with smaller tokens.
  Example:
  - Original data: apple banana apple orange banana apple
  - Compressed data: 1 2 1 3 2 1 (where 1 = apple, 2 = banana, and 3 = orange)

2.1.2. Heavyweight Compression

Heavyweight compression algorithms are more complicated techniques that can significantly reduce data size. They use complex data formats to achieve high CR, often requiring more computational resources and time.

- Huffman Coding [39]: Huffman coding is a type of **entropy encoding** used in lossless data compression. It assigns variable-length codes to a sequence of input characters, with shorter codes assigned to more frequent characters. This algorithm is used in various compression formats, including DEFLATE, which is employed in ZIP files and the PNG image format.

  Example:
  For the input "AAAAABBC", Huffman coding would assign shorter codes to 'A' (such as 0), slightly longer codes to 'B' (perhaps 10), and the longest to 'C' (maybe 11), resulting in a compressed binary output of 00000101011.
- Lempel-Ziv (LZ) Algorithms [40]: LZ algorithms, such as LZ77 and LZW, are widely used in general-purpose compression tools like ZIP and GIF. They work by searching for repeated sequences of data and replacing them with shorter references to previous occurrences. These algorithms are the basis for many well-established compression formats due to their balance in compression ratio and computational efficiency.

  Example:
  For the input "ABABABAC", an LZ algorithm identifies that after "AB", the rest of the string repeats earlier sequences. Instead of storing "AB" multiple times, it would store pointers to earlier instances.
- Arithmetic Coding [41]: Arithmetic coding is a more efficient form of entropy encoding than Huffman coding. Rather than assigning fixed codes to individual symbols, arithmetic coding represents the entire sequence of input data as a single number, a fraction between 0 and 1. The fraction represents a range that becomes more narrow based on the probability of each symbol in the sequence.

  Example:
  For the string "ABAC", where 'A' occurs 50% of the time, 'B' 25%, and 'C' 25%, arithmetic coding would generate a fraction representing the entire sequence of probabilities rather than coding

each symbol individually. The output fraction might look like 0.375, which could be decoded using the probabilities to recover the exact sequence.

- Prediction by Partial Matching (PPM) [42]: Prediction by Partial Matching is a statistical compression algorithm that uses a context-based model to predict the next character in a sequence. It creates a context for each symbol based on the preceding characters, and encodes the symbol using the probability of it occurring in that specific context.

  Example:
  For the input "ABBAB", PPM predicts the next character by analyzing previous character patterns. After "A" and "B", it will predict "B" as more likely based on the context, which allows it to use fewer bits for encoding repeated patterns.

- Burrows-Wheeler Transform (BWT) [43]: The Burrows-Wheeler Transform is a block-sorting algorithm that rearranges the characters of a data block to bring repeated characters together, which makes it easier to compress using simpler techniques like Run-Length Encoding (RLE) or Huffman coding. BWT is not itself a compression algorithm, but a transformation that significantly improves the performance of other compression methods.

  Example:
  For the string "banana", BWT transforms it into "annb$aa", bringing similar characters (the 'a's) together. This transformed string is better compressible using Huffman coding or run-length encoding.

- Context Mixing (CM) [44]: Context mixing combines multiple prediction models to predict the next symbol in a sequence. Each model focuses on different aspects of the data, and their outputs are combined to make more accurate predictions. It's used in advanced compression algorithms like PAQ, known for achieving some of the best compression ratios but at the cost of high computational requirements.

  Example:
  In compressing text, context mixing might combine predictions from a character-based model and a word-based model to encode the sequence "the quick brown fox" by using both previous letters and whole words as predictive contexts, improving compression efficiency.

- Asymmetric Numeral Systems (ANS) [45]: ANS is a more recent entropy encoding technique that combines the efficiency of arithmetic coding with the speed of Huffman coding. ANS works by encoding data as a single number, much like arithmetic coding, but it operates more efficiently by processing bits in blocks. ANS is widely used in modern codecs such as Zstandard (zstd), which is designed for both high compression ratios and high throughput.

2.2. Compression Algorithms Comparison

The compression techniques discussed in the previous chapter are implemented by many compression applications. Some applications are able to execute one compression technique with high throughput, while others combine multiple techniques to strike a different balance in throughput and compression ratio. Even between algorithms employing the same techniques, the performance can vary widely. For example, LZ4 and Brotli are two widely used LZ-based compressions. As shown in Figure 2.2, Brotli can achieve over 2 times the compression ratio of LZ4, while LZ4 can get over 4 times the decompression throughput of Brotli. This shows that the techniques used do not necessarily dictate the performance of the implementation. This section compares different implementations of compression techniques, and aims to extract characteristics for different techniques.

2.2.1. CPU Compression

This section gives an overview of CPU implementations of compression techniques and shows how their performance compares. Which techniques are good for achieving high compression ratios, and which for decompression throughput? First, a high level impression is gained from the result of the Large Text Compression Benchmark. Second, a more detailed information is acquired by running the

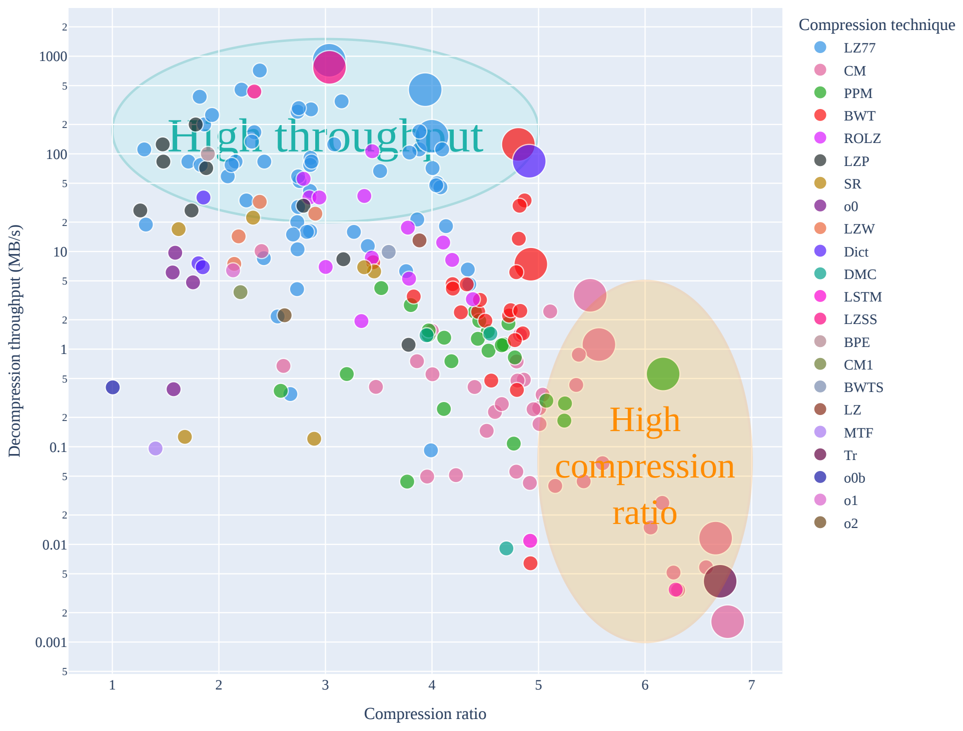


Figure 2.1: Comparison of the decompression throughput and compression ratio of compression techniques, from the results in the large text compression benchmark (On CPU) [46]. Each dot represents a different compression algorithm, and the large dots are on the Pareto front.

LZBench compression benchmark. Finally, these results are combined to get a clear overview over the state-of-the-art for text data compression.

Large Text Compression Benchmark

The large text compression benchmark challenges developers to design compression algorithms that can compress two datasets (the first 100 MB and 1 GB of Wikipedia) into the smallest possible size [46]. The algorithms are evaluated solely based on the size of the compressed file and the size of the binary, meaning the throughput of the algorithms is typically not a primary focus for the programmers. However, since both compression and decompression times are recorded, it's possible to calculate their throughput and make performance comparisons. The following points are observed from Figure 2.1:

- LZ-based algorithms typically have high decompression throughput but achieve lower compression ratios.
- BWT-based algorithms often provide a good balance between decompression throughput and compression ratio.
- CM-based algorithms achieve the highest compression ratios but suffer from very low decompression throughput.
- In general, as compression ratios increase, decompression throughput tends to decrease.

LZBench

LZBench is an open source benchmark project that implements a wide range of (mostly LZ-based) compression algorithms in a standardized test interface [47]. The project provides a convenient way to benchmark many compression algorithms in the same environment. All compression algorithms supported in LZBench are executed on the test setup described in Table 5.1, with a text file of 1.5 GB as input generated in TPC-H format using dbgen. The best performing algorithms are shown in Figure 2.2.

During the course of the research, the compression algorithm called FSST came to attention. A custom implementation of FSST was integrated into LZBench to evaluate its performance against other compression algorithms.

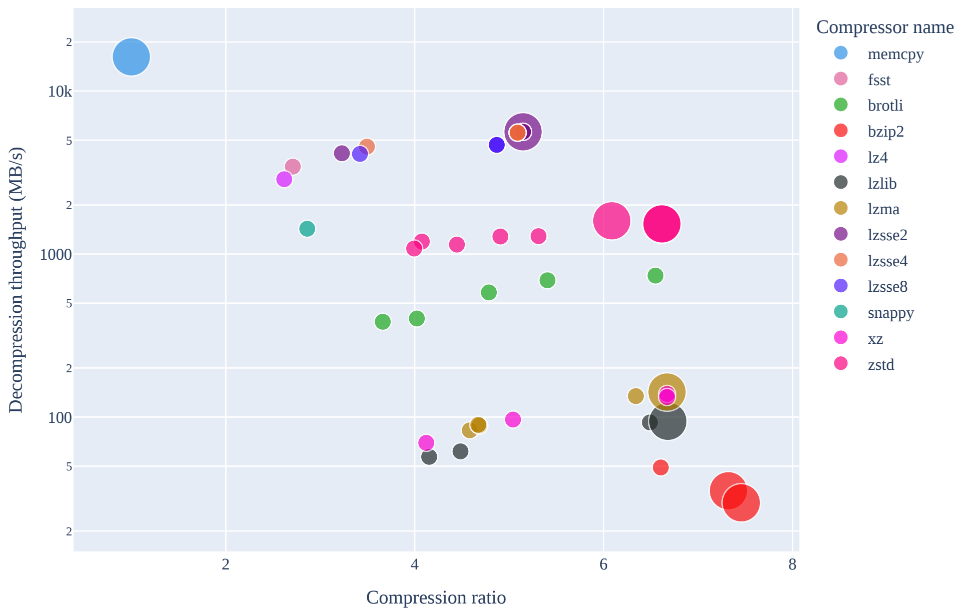


Figure 2.2: Comparison of the decompression throughput and compression ratio of CPU compression algorithms from executing LZBench on the test setup described in Table 5.1. This benchmark uses a 1.5 GB text file, generated using dbgen. The large dots are on the Pareto front. Dots of the same color are the same algorithm with different parameters.

The following observations are done based on Figure 2.2:

- lzsse achieves the highest decompression throughput with using SIMD instructions at 5.6 GB/s
- FSST achieves the highest decompression throughput without using SIMD instructions at 3.4 GB/s
- bzip2 achieves the highest compression ratio of 7.5
- zstd strikes a good balance between compression ratio and decompression throughput, with 6.6 and 1.5 GB/s.

2.2.2. GPU Compression

Parallel compression has seen significant advancements in recent years, leading to the development of various GPU-based implementations of the compression techniques discussed in Section 2.1. The performance of these implementations vary widely, as shown in Table 2.1. Algorithms like DietGPU and nvCOMP Cascaded are specifically designed for floating point data, making use of the GPU's ability to handle floating point operations more efficiently.

One thing that stood out for these results is the large difference in performance of a general-purpose compression algorithm. While Table 2.1 lists the highest decompression throughput reported by the developers, many of these algorithms experience a dramatic drop in performance when the input data type or size changes. For instance, the decompression throughput of all nvCOMP algorithms fall to less than 20% of the reported performance when switching from integer data to textures [48]. This highlights how sensitive compression algorithms are to the data they are working with.

GPU compression implementation	Techniques used	Data type	Reported Decompression Throughput
nvCOMP ANS	ANS	General Purpose	Up to 269 GB/s on A100
nvCOMP Bitcomp	Custom	Numerical	Up to 351 GB/s on A100
nvCOMP Cascaded	RLE, Delta, Bit-packing	Numerical	Up to 730 GB/s on A100
nvCOMP Deflate	LZ, Huffman	General Purpose	Up to 188 GB/s on A100
nvCOMP GDeflate	LZ, Huffman	General Purpose	Up to 221 GB/s on A100
nvCOMP LZ4	LZ	General Purpose	Up to 369 GB/s on A100
nvCOMP Snappy	LZ	General Purpose	Up to 204 GB/s on A100
nvCOMP Zstd	LZ, Huffman, ANS	General Purpose	Up to 281 GB/s on A100
DietGPU [49]	ANS	Numerical	Up to 410 GB/s on A100
ndzip-gpu [50]	Custom	Numerical	Up to 324 GB/s on A100
GPULZ [51]	LZ	General Purpose	Up to 29 GB/s on A100
Recoil [52]	ANS	General Purpose	Up to 96 GB/s on RTX 2080 Ti
GFC [53]	Custom	Numerical	Up to 121 GB/s on Quadro FX 5800
Gompresso [54]	LZ, Huffman	General Purpose	Up to 16 GB/s on Tesla K40
... ANS Decoding on GPUs [55]	ANS	General Purpose	Up to 19 GB/s on V100
... bzip2 Decompression on GPUs [56]	BWT, RLE, Huffman	General Purpose	Up to 11 GB/s on eight H100

Table 2.1: Comparison of GPU compression implementations based on their techniques used.

The implementations of these compression techniques on GPU show that the developers carefully considered the design of the implementations. No straightforward ways to significantly increase the throughput of these implementations is found. The performance achieved highlights the challenge of implementing these complex compression techniques on the parallel architecture of a GPU. To address the need for higher throughput, this thesis will shift focus to CPU-based compression algorithms. It will explore other compression techniques can be parallelized and look for methods that can deliver the desired high decompression throughput when adapted to a GPU architectures.

2.3. Comparing Algorithms for Parallel Decompression: Why FSST Stands Out

The parallelization of data decompression is challenging as a result of the inherent sequential dependencies in the compressed data. These dependencies mean that parts of the data rely on other parts of the data, making it difficult to process the data in parallel. For instance, if a byte of data at position *X* must be decompressed before the byte at *X+1* can be decompressed, it forces the decompression to proceed sequentially from the start to the end, limiting the potential for parallel processing.

As shown in Figure 2.3, the data dependencies involved in decompression can be divided into four main categories:

1. **Compressed to compressed**: Compressed data requires other compressed data to be available before it can be decompressed. An example of this is ANS, which compresses data into a single, highly accurate value that is decoded into multiple output bytes. Because the compressed data is interdependent as a single number, it becomes challenging to start decompressing in the middle of the data, making parallelization difficult.

2. **Compressed to decompressed**: Compressed data needs the decompressed version of other data to proceed. This is typical of the LZ compression algorithm, where compressed data contains references to previously seen data. When these references are encountered during decompression, the referenced data must already be decompressed; otherwise, the process cannot continue.

3. **Decompressed to compressed**: This scenario, where decompressed data relies on still-compressed data, is uncommon.

4. **Decompressed to decompressed**: Decompressed data depends on other decompressed data. The challenge here arises from the fact that compressed data can have variable decompressed lengths. If decompression starts partway through the data, the final location for output data is unknown until the preceding data has been decompressed.

As these categories demonstrate, most traditional compression algorithms have sequential data dependencies in their format, making them less suited for parallelization and require additional methods to function efficiently on GPU architectures. A method commonly used under the implementations in Section 2.2.2 is tile-based compression, as described in [57]. This approach divides the input data

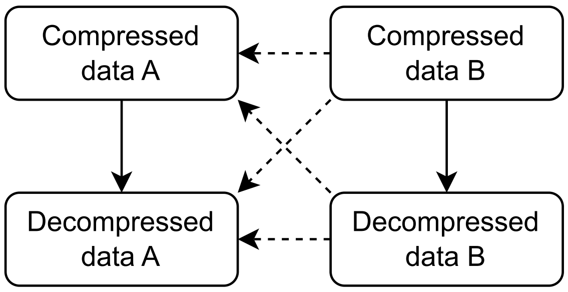


Figure 2.3: Diagram showing the data dependencies in a (compressed) buffer that complicate parallel decompression. A compressed data buffer consisting of compressed part A and compressed part B can have four sequential data dependencies, which prevent decompressing part B in parallel with part A.
Compressed B to compressed A: Decompressing compressed data in B depends on the compressed data in A (Example: Decoding ranges in ANS [58]).
Compressed B to decompressed A: Decompressing compressed data in B depends on the decompressed data in A (Example: The back references in all LZ-based compressions).
Decompressed B to compressed A: Decompressed data B is only usable with information from compressed data A (Example: None).
Decompressed B to decompressed A: Decompressed data B is only usable with information from decompressed data A (Example: The decompressed length of A determines the start of decompressed B).

into independent chunks that can be compressed and decompressed separately, allowing for parallel execution. This technique can be applied to most compression algorithms.

The comparison of GPU-based compression techniques in Section 2.2.2, shows that most common compression techniques have already been implemented on GPU, and leave little room for significant improvements. However, the comparison of CPU compression algorithms in Section 2.2 shows that a compression algorithm using a new compression technique stands out. FSST achieves the highest decompression throughput of the algorithms that don't use SIMD, making it a compelling candidate for further exploration.

One of the key advantages of the format used by FSST, which will be introduced in Section 2.4, is that it only has the *decompressed to decompressed* sequential dependency. This means that, unlike other traditional compression algorithms, FSST does not rely on either compressed data or previously decompressed data. The primary challenge it faces is the variable decompressed length of data chunks, which will become a challenge when attempting to parallelize decompression. However, this type of dependency is less restrictive compared to those seen in other algorithms like ANS or LZ, where earlier data is need before subsequent data can be accessed. Based on these observations, this thesis will focus on analyzing and optimizing FSST compression for an efficient GPU implementation.

2.4. FSST Compression

FSST (Fast Static Symbol Table) is a string compression algorithm designed to balance throughput, compression ratio, and random access capabilities, making it well-suited for use in databases [59]. FSST is particularly effective for compressing and decompressing strings.

The core idea behind FSST is to replace frequently occurring substrings with short, fixed-size 1-byte codes, as shown in Figure 2.4. This is done in two steps. First, the algorithm builds a static symbol table. This table serves as a dictionary, mapping frequently found substrings (which can be between 1 and 8 bytes long) to 1-byte codes. This static symbol table remains consistent across a block of compressed data, allowing for efficient random access. Building the symbol table starts by scanning the input data and identifying the most common substrings. These substrings are then added to the symbol table. In the second step, FSST performs the compression by replacing each occurrence of these frequent substrings with their corresponding 1-byte code. When the compression algorithm encounters a substring not present in the symbol table, it uses the *escape code* followed by the raw byte, ensuring that any input data can be accurately represented, even if it hasn't been captured in the symbol table.

Once compressed, the data consists of a sequence of these 1-byte codes.

1. A **code** is a fixed-size 1-byte reference to an entry in a lookup table containing symbols, called the symbol table.

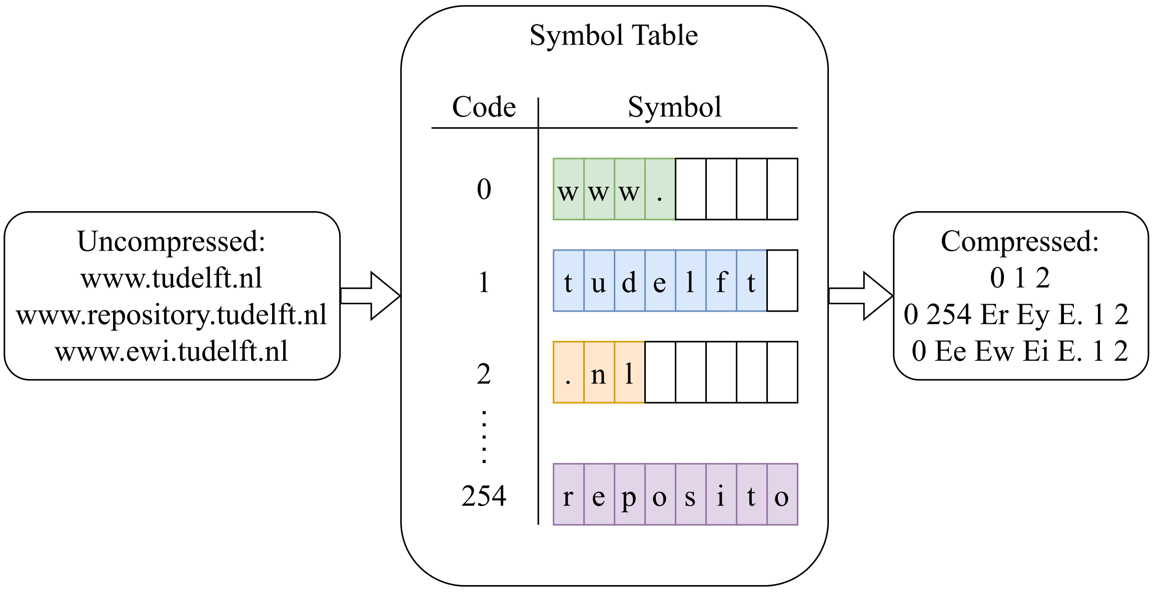


Figure 2.4: Demonstration of FSST compression. During the compression, the input data is scanned for repeated patterns. These are inserted in the symbol table. During the compression process, data is replaced by (1-byte) codes from the symbol table. Data patterns that can't be found in the symbol table, are prepended by an escape code (here shown as **E**). The escape code allows storing individual raw bytes.

2. A **symbol** is an entry in a symbol table, with a size ranging from 1 to 8 bytes.
3. A **symbol table** is a lookup table containing 255 symbols, indexed using codes.
4. A **escape code** is a reserved code, used to indicate that the byte following the escape code is not a code but a byte that should be copied directly to the output.
5. A **escaped character** is the byte preceded by an escape code.

FSST Decompression

Decompression in FSST is straightforward due to its static symbol table. Each 1-byte code in the compressed data directly corresponds to a predefined substring in the symbol table. The decompression simply looks up each code in the table and retrieves the original substring. If an escape code is encountered, it means the next byte is raw data that hasn't been compressed and should be copied as is.

Because the symbol table is static, FSST can decompress individual (sub)strings without needing to process an entire block of data. This random access capability makes FSST highly efficient in applications where only portions of the data need to be accessed or queried, such as in databases. This property also provides opportunity for the parallel decompression of FSST compressed data. When dividing the whole buffer up into equal sized substrings, parallel decompression can be achieved to the level of the number of divisions.

2.5. GPU Hardware Architecture

GPUs are designed to handle many parallel operations simultaneously, making them ideal for tasks that require processing vast amounts of data quickly. Unlike CPUs, which are optimized for sequential processing, GPUs excel at parallel processing. This distinction makes GPUs highly effective for applications that can be broken down into smaller, concurrent tasks.

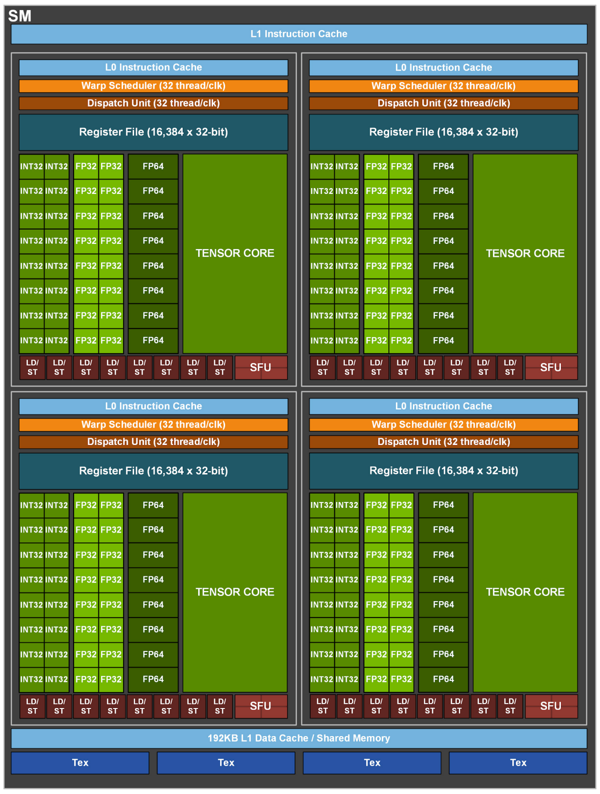
At the core of a GPU are numerous simple processing units, often referred to as cores or shader processors. A GPU's architecture is organized into multiple groups of cores called Streaming Multiprocessors (SMs). Each SM can execute hundreds of threads simultaneously, leveraging parallelism to achieve high throughput. Within an SM, threads are organized into warps, which are the smallest unit of execution. Warps execute instructions in lockstep, meaning all threads in a warp follow the same execution path, which simplifies control flow but can lead to inefficiencies if threads diverge.

The memory hierarchy of a GPU, visualized in Figure 2.6, is an important aspect of its architecture. The global memory, which is the largest and slowest, is accessible by all cores but has higher latency. The L2 cache, sitting between the global memory and the SMs, acts as an intermediate storage, reducing latency and improving data access efficiency for frequently used data. L1 cache or shared memory, located within each SM, is much faster and allows threads within the same block to share data efficiently. Registers, the fastest and smallest memory units, are used for storing temporary variables during execution. Additionally, GPUs have specialized memory types like texture memory and constant memory, optimized for specific data access patterns to enhance performance.

Interconnects on a GPU, such as PCIe and NVLink, are responsible for data movement between different components. High bandwidth and low latency interconnects are essential for maintaining performance, particularly in data-intensive applications where quick data transfer between the CPU and GPU or between multiple GPUs is required.

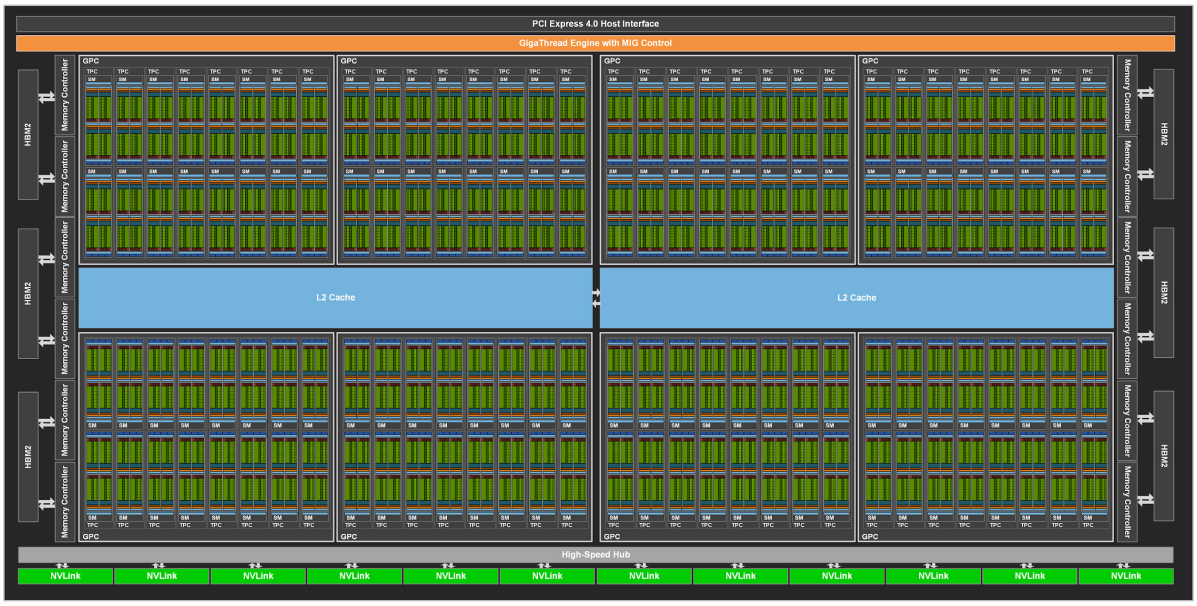
The primary advantage of GPUs lies in their ability to perform massive parallel computations. To harness this power effectively, programmers must structure their code to exploit parallelism. A critical challenge is thread divergence, which occurs when threads within a warp follow different execution paths, leading to performance reduction. Achieving high occupancy, which refers to the ratio of active warps to the maximum number of warps an SM can support, is essential for maximizing GPU utilization.

Efficient memory management is another vital aspect of programming for GPUs. Coalesced memory access, where threads access contiguous memory locations, minimizes memory latency and maximizes bandwidth utilization.


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Source: [60]

Figure 2.5: NVIDIA GA100 Streaming Multiprocessor (SM)


PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

Source: [60]

Figure 2.6: NVIDIA GA100 Chip design

3 Ingestion Throughput Analysis

This chapter focuses on identifying and analyzing bottlenecks in GPU data ingestion. Section 1.1.2 introduced decompression throughput as a new limiting factor in the data ingestion process for GPUs. In this chapter, a model is presented to quantify the effect that different compression algorithms have on ingestion throughput, providing insight into which metrics are important.

3.1. Ingestion Model

Data ingestion is the process of loading data from storage into memory, making it ready for processing. Before technologies like GPU Direct Storage (GDS) were available, this process involved a two-step transfer: data first had to pass through CPU memory before being moved to GPU memory. As illustrated in Figure 3.1a, this method could cause bottlenecks due to the use of CPU memory bandwidth and required the use of the interconnect twice, applying twice the load. With the introduction of GDS, this extra step is removed, allowing data to be transferred directly from storage to GPU memory. This bypasses the CPU, eliminating bandwidth limitations and simplifying the ingestion process, as shown in Figure 3.1b.

Our paper titled 'Benchmarking GPU Direct Storage for High-Performance Filesystems: Impact & Future Trends' presents a comprehensive analysis of data ingestion from parquet files to GPU [2]. In this paper, we present a model where we abstract data ingestion into three stages:

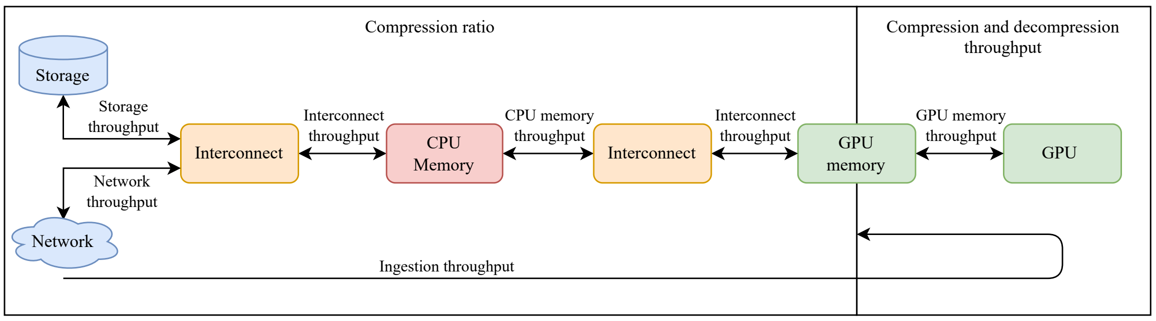
1. Data transfer (IO): Transferring the data from storage to GPU memory over an interconnect.
2. Data decompression: Decompressing data from GPU memory to GPU memory.
3. Data decoding: Restructuring the data into the desired computation model.

Two adaptations are applied to this model to make it fit the objective of this thesis. (1) The decoding stage is not measured in this thesis, and will not be included. (2) The model is reformulated under the assumption that data transferring and decompression can not happen at the same time (Section 3.2) and under the assumption that data transfer and decompression can happen at the same time on block level (Section 3.3).

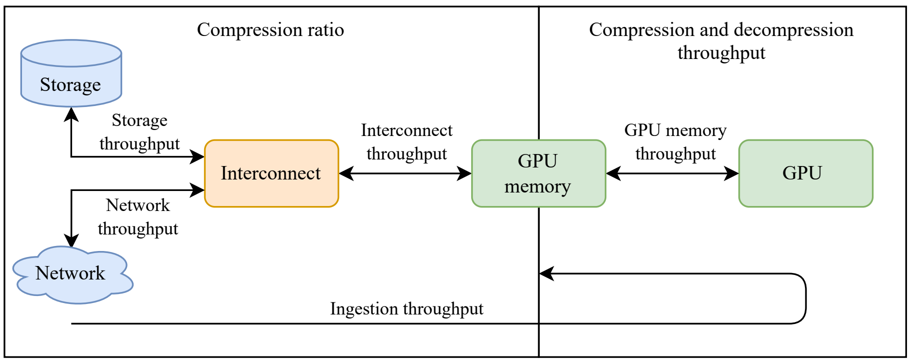
These formulas show how the total throughput of ingestion can be determined from the specs of the system and decompression algorithm.

3.2. Ingestion Throughput

When it is not possible to overlap data transfer and decompression, the entire compressed buffer is treated as a single block. In this situation, all of the compressed data is first transferred to the GPU memory. Once the transfer is complete, decompression can begin. This process is depicted in Figure 3.2a. The total time required to ingest data in this case consists of two parts: the time it takes to transfer the compressed data over the interconnect, and the time it takes to decompress the data once it's on the GPU. These factors are represented in Equation (3.1).

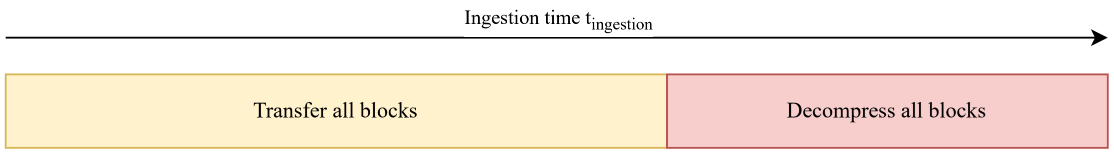


(a) Data ingestion model without GPU direct memory

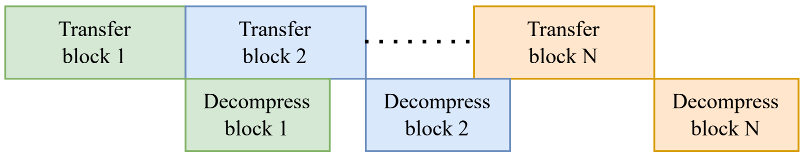


(b) Data ingestion model utilizing GPU Direct Memory [61], which will be the model used throughout the thesis.

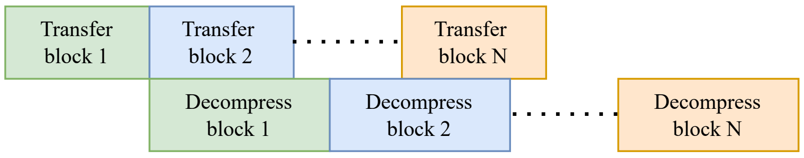
Figure 3.1: Two models for data ingestion into a GPU, showing how compression ratio and (de)compression throughput affect the stages of the process. Increasing the compression ratio helps speed up the transfer of data from storage to GPU memory, while increasing (de)compression throughput improves the extraction of data in the GPU's global memory.



(a) No overlapping of data transfer and decompression



(b) Overlapping data transfer with decompression when decompression is faster than transfer



(c) Overlapping data transfer with decompression when transfer is faster than decompression

Figure 3.2: Timelines showing the data ingestion time ($t_{ingestion}$). This time consists of transferring compressed data over an interconnect and decompressing it.

The time it takes to move the data over the interconnect (t_{io}) depends on the amount of compressed data (S_c) that needs to be moved and the speed at which the interconnect can move the data (T_{io}) (Equation (3.2)). The time it takes to decompress data (t_d) depends on the throughput of the decompression algorithm (T_{dout}) and the uncompressed size of the data (S_u), as defined in Section 1.4 (Equation (3.3)).

Ingestion throughput is defined as the amount of data processed per unit of time. The overall ingestion throughput ($T_{ingestion}$) is calculated based on the total time ($t_{ingestion}$) needed to ingest a given amount of uncompressed data (S_u). From this, a formula can be derived to calculate ingestion throughput using the compression ratio, decompression throughput, and interconnect throughput.

A key result of this formula is that the input data size becomes irrelevant for calculating ingestion throughput. Instead, the focus shifts to throughput values and the compression ratio. The following conclusions can be drawn from this:

1. **Compression Ratio and Interconnect Throughput**: The compression ratio is closely tied to the throughput of the interconnect, which is typically a fixed factor in most systems. This means the compression ratio can be adjusted to compensate for limitations in interconnect throughput.

2. **Equal Decompression and Transfer Throughput**: When the decompression throughput and transfer throughput are equal, the ingestion throughput will be half of their shared value. This makes sense because data ingestion is made up of two separate processes (transfer and decompression), and when they take the same amount of time, the overall ingestion time is the sum of both.

3. **Balancing Decompression and Transfer Throughput**: To achieve maximum ingestion throughput, there needs to be a balance between the decompression throughput (T_{dout}) and the effective transfer throughput ($CR \cdot T_{io}$). If one of these values is significantly larger than the other, it will limit the overall ingestion throughput. Therefore, improving the smaller of the two will result in higher ingestion throughput. As the interconnect throughput increases, the compression ratio doesn't need to be as high anymore. When the interconnect throughput increases too much, a low decompression throughput will cause the ingestion throughput to be kept back.

In summary, this model provides a clear mathematical framework for understanding how data ingestion throughput is influenced by both decompression throughput, data transfer throughput, and the compression ratio. It shows the importance of maintaining a balance between these two components to maximize ingestion throughput. This model is used to determine the impact of design decisions on the total ingestion throughput of the system.

$$\begin{aligned} t_{ingestion} &= t_{io} + t_d \\ &= \frac{S_c}{T_{io}} + \frac{S_u}{T_{dout}} \end{aligned} \tag{3.1}$$

$$t_{io} = \frac{S_c}{T_{io}} \tag{3.2}$$

$$t_d = \frac{S_u}{T_{dout}} \tag{3.3}$$

$$
\begin{aligned}
T_{ingestion} &= \frac{S_u}{t_{ingestion}} \\
&= \frac{S_u}{t_{io} + t_d} \quad \text{(substitude Equation (3.1))} \\
&= \frac{S_u}{\frac{S_c}{T_{io}} + \frac{S_u}{T_{dout}}} \quad \text{(substitude Equations (3.2) and (3.3))} \\
&= \frac{S_u}{\frac{\frac{S_u}{CR}}{T_{io}} + \frac{S_u}{T_{dout}}} \quad \text{(substitude Equation (1.1))} \\
&= \frac{S_u}{\frac{S_u}{CR \cdot T_{io}} + \frac{S_u}{T_{dout}}} \\
&= \frac{1}{\frac{1}{CR \cdot T_{io}} + \frac{1}{T_{dout}}} \\
&= \frac{1}{\frac{T_{dout}}{CR \cdot T_{io} \cdot T_{dout}} + \frac{CR \cdot T_{io}}{CR \cdot T_{io} \cdot T_{dout}}} \\
&= \frac{1}{\frac{T_{dout} + CR \cdot T_{io}}{CR \cdot T_{io} \cdot T_{dout}}} \\
&= \frac{CR \cdot T_{io} \cdot T_{dout}}{T_{dout} + CR \cdot T_{io}}
\end{aligned} \tag{3.4}
$$

3.3. Overlapping Data Transfer and Decompression

Splitting the compressed data into independent blocks allows the data ingestion to partially overlap data transfer and decompression. When a full block is received, the compression algorithm can already start decompressing it, in parallel with the transfer of the next block. This situation

The model has the following assumptions:

- The transfer time of all blocks is equal
- The decompression time of all blocks is equal
- The uncompressed size of the blocks is approximate equal

Under these assumptions, there are two possible situations for the ingestion time based on the decompression time ($t_{d\ block}$) and transfer time of a block ($t_{io\ block}$):

1. When the transfer time of a block is larger than the decompression time, the ingestion time is limited by the total transfer time of the blocks, plus the decompression time of the last block (Figure 3.2b).
$$t_{io\ total} + t_{d\ block\ N}$$
2. When the decompression time of a block is larger than the transfer time, the ingestion time is limited by the total decompression time of the blocks, plus the transfer time of the first block (Figure 3.2c).
$$t_{d\ total} + t_{io\ block\ 1}$$

Similar to the derivation in Equation (3.4), it is possible to derive Equations (3.5) and (3.6) for the ingestion throughput in both situations. The full derivations are shown in Appendix A.

$$T_{ingestion\ Figure\ 3.2b} = \frac{CR \cdot T_{io} \cdot N_{blocks} \cdot T_{dout}}{N_{blocks} \cdot T_{dout} + CR \cdot T_{io}} \tag{3.5}$$

$$T_{ingestion\ Figure\ 3.2c} = \frac{T_{dout} \cdot CR \cdot N_{blocks} \cdot T_{io}}{CR \cdot N_{blocks} \cdot T_{io} + T_{dout}} \tag{3.6}$$

The structure of these equations is very similar to the case without overlapping, as seen in Equation (3.4). The key difference lies in how the number of blocks is positioned within the formulas. More importantly, its position shifts when the decompression time of a block exceeds the transfer time. The same conclusions drawn in the section about no overlapping between transfer and decompression still apply here, but with the added impact of the number of blocks factored in:

1. **When the transfer time of a block is larger than the decompression time** (Figure 3.2b): The number of blocks can be used to compensate for the decompression throughput. Increasing the number of blocks increases the ingestion throughput, as long as $N_{blocks} \cdot T_{dout}$ and $CR \cdot T_{io}$ stay in balance, ensuring neither becomes significantly larger than the other.

2. **When the decompression time of a block is larger than the transfer time** (Figure 3.2c): The number of blocks can be used to compensate for the compression ratio or interconnect throughput. Increasing the number of blocks increases the ingestion throughput, as long as $CR \cdot N_{blocks} \cdot T_{io}$ and T_{dout} stay in balance, ensuring neither becomes significantly larger than the other.

This analysis shows that by overlapping data transfer with decompression, there is an additional level of control over data ingestion throughput through the number of blocks. However, this version of the model has not been tested and is not included in the benchmarks or results of this thesis.

4 GSST Compression Design

This chapter covers the design choices behind the GSST data formats and decompression algorithms, building upon the latest version of the FSST decompression algorithm. The aim is to improve both parallelism and throughput through targeted optimizations.

Optimizations are made in two areas: the data format and memory management. The data format is optimized to allow greater parallelism, enabling the decompression process to handle multiple tasks concurrently, which results in higher throughput. Additionally, memory management optimizations are applied to better align the implementations with the GPU memory architecture.

Each implementation is first developed as an OpenMP multithreaded C++ CPU solution for both compression and decompression, ensuring the correctness of the compression format. Following this, only the decompression is implemented on the GPU as a CUDA kernel, which is then used for performance benchmarking throughout the thesis.

4.1. Compression Data Format

A data layout describes how data is stored in memory or disk. It describes how complex data structures are reduced (serialized) to a byte stream. Standardizing the data format used by a compression algorithm is important as it makes the compression programming language independent, allows checking the correctness of compressed data, and recovering it when mistakes were made.

State-of-the-art compression algorithms, such as Zstandard (Zstd) [62], LZ4 [63], and Brotli [64], define their file formats in two hierarchical levels: the frame format and the block format. The block format outlines how data is compressed and stored at the most granular level, while the frame format describes how multiple blocks are organized together. This structure is illustrated in Figure 4.1a.

GSST compression makes use of a similar format. An input buffer is compressed to a single frame that contains a variable number of blocks (N_{block}). Each block starts with a block header that includes information that is needed during decompression. By taking advantage of this format and incorporating additional metadata in the headers, more parallelism is introduced in order to increase throughput.

4.2. Data Format Optimization

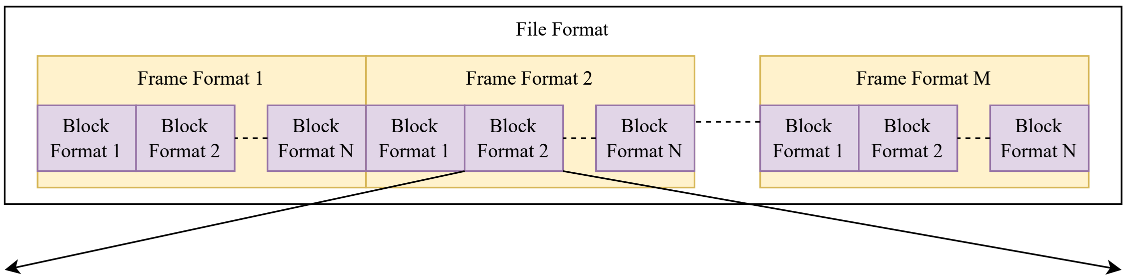
4.2.1. Block Parallelism Format

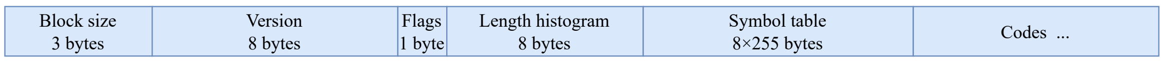
The FSST project includes a command-line binary that compresses files using a specific block-based format, referred to as the FSST format, as shown in Figure 4.1b. The compression algorithm processes uncompressed data in 4 MB blocks, compressing each block independently. Each block is given a header containing information such as the compressed block size, algorithm version, flags, and symbol table. The GSST format builds upon this FSST block structure.

The independence of the blocks in the FSST format creates an opportunity for parallel processing. Similar to the tile-based technique described by [57] and the strategies introduced in [65], the independent blocks of data can be decompressed in parallel. However, to achieve full parallelism, a one piece of information is missing: the uncompressed size of each block. Without this, the output location for the

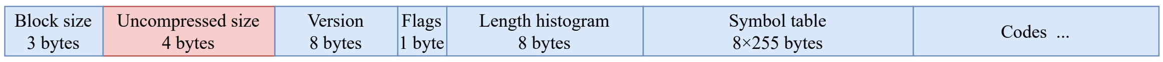
Figure 4.1: The file formats used by FSST, GSST blocks, and GSST splits. Each new format adds extra information to the block header to increase parallelism, and with it the throughput.

(a) The file format describes how data is structured and stored in a compressed buffer. It contains all information needed for the data to be fully decompressed.

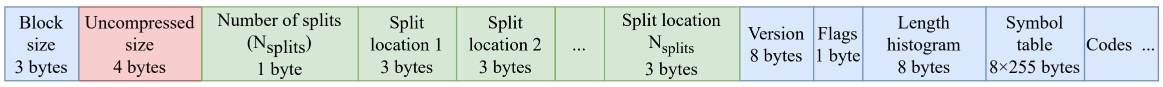




(b) Block format used by the FSST compression binary



(c) Block format used by GSST blocks compression



(d) Block format used by GSST splits and coalesced compression

next block (Block X+1) can't be determined until the current block (Block X) has been fully decompressed. This creates a sequential bottleneck in the decompression process, preventing true parallelism.

There are two ways to access the uncompressed size during decompression:

1. Use a constant uncompressed block size, meaning all blocks will decompress to exactly the same size.
2. Store the uncompressed size with each block.

Storing the uncompressed size with each block increases the size of the compressed file, which reduces the compression ratio. Using a constant uncompressed block size avoids this issue, but sacrifices flexibility. This would mean that any input size would need to use the same block size, limiting the ability to adjust the block size (and with it the number of blocks) for different input sizes. Being able to change the number of blocks is an important parameter in achieving optimal parallelism.

Equation (4.1) shows how the amount of storage overhead can be determined by adding the uncompressed size to the block header. Especially on large buffers, this amount of overhead is not expected to be impactful. However, this is a reason why the number of blocks should stay limited. Overall, it will be more effective to store the uncompressed size in the header of the block, as shown in the block format in Figure 4.1c,

$$S_{\text{overhead uncompressed block size}} = 4 \cdot N_{block} \tag{4.1}$$

With the uncompressed size added to the header, the format allows all blocks to be decompressed independently and in parallel. It provides parallelism equal to the number of blocks that are used. The amount of work done by a single execution unit is equal to the size of the block, which is related to the input size and the number of blocks.

The block format provides a new parameter for the compression algorithm, namely the number of blocks (N_{blocks}). The number of uncompressed bytes in a block (S_{block}) is calculated from the uncompressed size of the input data (S_u) and the number of blocks (N_{blocks}) using Equation (4.2). This formula makes sure that all blocks get the same size of input data, with the maximum difference being 1.

$$S_{block} = \left\lfloor \frac{S_u + N_{blocks} - 1}{N_{blocks}} \right\rfloor \tag{4.2}$$

Distributing the blocks over the parallel architecture of the GPU can be challenging. As this format does not allow multiple threads to work on the same block, each core will need its own block of compressed data. In order to make all cores in a recent GPU participate in the decompression, the data will always need to be compressed into 6912 (A100) or 16896 (H100) blocks. This has three major disadvantages. (1) Especially for small input buffers, this can impact the CR, as the data per block decreases, but the size of the header stays the same. (2) On top of that, decompressing 128 blocks at the same time in a single SM will overload its caches. Each core needs quick access to its own symbol table of 2 KB, resulting in a required cache size of 256 KB. (3) Finally, letting all threads in a thread block work on a different data block will increase thread divergence, as different data block decompress at different speeds, different threads will be in different stages of the decompression.

To address these challenges, the implementation allows for reducing the number of active threads within a thread block, improving performance under these conditions.

Appendix B shows the decompression throughput achieved by the CPU implementation of block parallelism GSST.

Handling escape codes in blocks format

In block format, each block is compressed separately, and each has its own symbol table. Because of this separation, there are no conflicts with escape codes in block parallelism.

4.2.2. Split Parallelism Format

The block parallelism approach, described in Section 4.2.1, allows for the distribution of data blocks across the streaming multiprocessors (SMs) in a GPU. However, it can be challenging to further distribute data across the individual cores within an SM without creating an excessive number of blocks or exceeding the cache capacity. The splits parallelism format overcomes this challenge by introducing additional parallelism within a compressed block. Modern GPUs have 64 or 128 cores per SM, so the goal of the splits parallelism format is to enable a single block to be processed in parallel by at least 128 threads. This allows for the efficient distribution of data across the cores, as illustrated in Figure 4.3

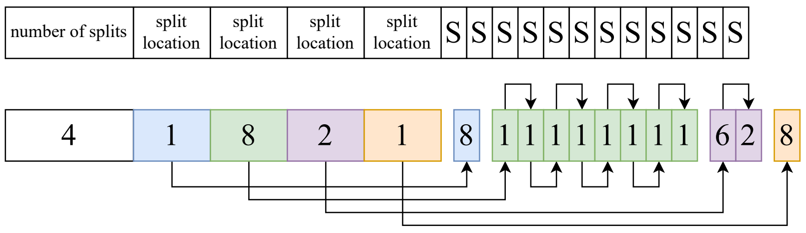
Introducing parallelism within a block faces challenges similar to block parallelism. To decompress a block in parallel, the sequential dependencies within the block must be resolved, as described in Section 2.3. By utilizing the FSST format, the only sequential dependency to manage is the distance of the output data between two splits. If a thread wants to begin decompressing in the middle of a block, it can simply use a copy of the symbol table and start converting codes into symbols partway through the block. However, there is one problem: the decompressed data's output position depends on the varying output lengths of the preceding data. One solution is to buffer the data and then copy it to the correct output location once the preceding data has been decompressed. However, this introduces a sequential step in the process, and adds unnecessary overhead from copying data.

A better solution is to store the uncompressed sizes of different parts of the block in the block header. With this information, each thread can start decompressing at specific points within the block's codes section and know exactly where the preceding split's data will end. This allows each thread to directly write its output data to the correct location, eliminating the need for extra copying. This split-related metadata is stored in the block header, as illustrated in Figure 4.2a.

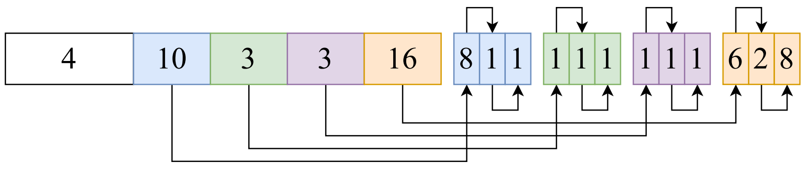
There are two ways to store the uncompressed sizes of splits in the block header, as shown in Figures 4.2a and 4.2b. While both formats solve the same problem of distributing data over multiple threads in a block, their implementation is quite different:

1. **Constant uncompressed size** (Figure 4.2a): In this format, a fixed value (in this case 8 bytes) is used for the uncompressed size of each split. The block header stores the number of codes used to compress each of these 8-byte chunks. Each thread calculates its output location by multiplying its split number by 8, as each split always outputs exactly 8 bytes. The thread then determines where to start decompressing by summing the sizes of the previous splits. For example, decompression of split 3 starts at code location 1 + 8 = 9, and it needs to decompress 2 codes to output exactly 8 bytes.
   This approach ensures that every thread writes the same amount of data to the output buffer. However, the number of bytes each thread reads can vary significantly, which can lead to inefficiencies. Some threads may process their data faster than others, causing thread divergence.

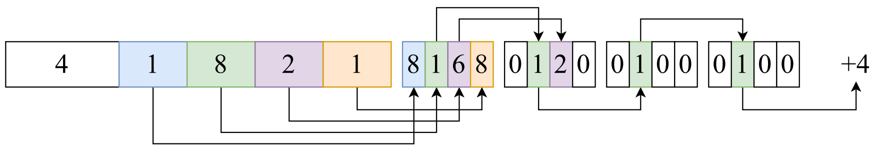
2. **Constant number of codes** (Figure 4.2b): In this format, a fixed number of codes (in this case



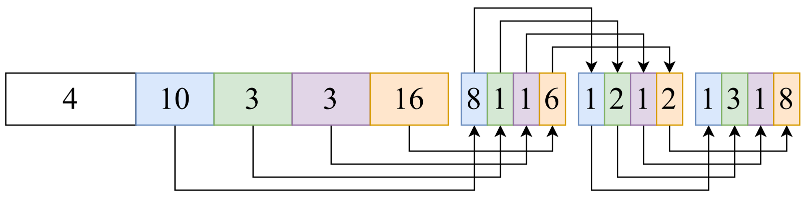
(a) Parallel decompression when a block is divided into splits by using a constant uncompressed size. This format stores the compressed length of each split in the block header, and each split has uncompressed size 8.



(b) Parallel decompression when a block is divided into splits by using a constant compressed size. This format stores the uncompressed length of each split in the block header, and each split consists of 3 codes. This code ordering format is used for GSST splits compression.



(c) Parallel decompression when a block is divided into coalesced splits by using a constant uncompressed size. This format stores the compressed length of each split in the block header, and each split has uncompressed size 8.



(d) Parallel decompression when a block is divided into coalesced splits by using a constant compressed size. This format stores the uncompressed length of each split in the block header, and each split consists of 3 codes. This code ordering format is used for GSST coalesce compression.

⟶ Thread execution order

Figure 4.2: Four formats considered for ordering the codes into splits in a compressed block. The block uses the same file format as illustrated in Figure 4.2a. Each arrow path represents the execution order of a thread. The format describes the order in which the codes are stored. In this example, the codes are divided over four splits, storing four values as metadata in the header. The number in a code depicts the uncompressed length of that code.

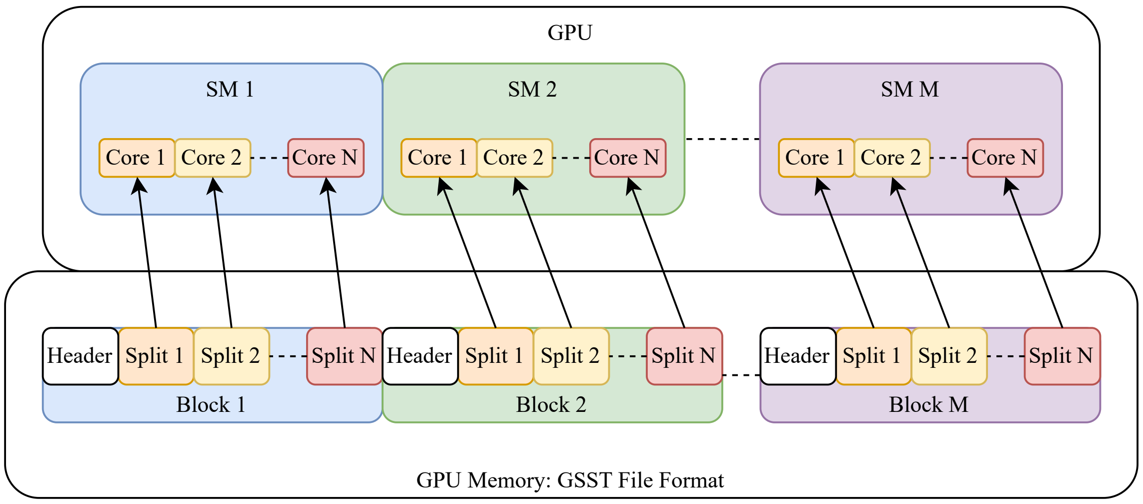


Figure 4.3: Mapping between the splits in the data format and the cores in the GPU. The compressed data is stored in global GPU memory, according to the file format described in Figure 4.2a.

3 codes) is assigned to each split, and the block header stores the uncompressed size of each split. Each thread knows where to start decompressing by multiplying its split number by 3, as every split always has exactly 3 codes. To find the correct output location, each thread sums the uncompressed sizes of the preceding splits. For instance, thread 3 starts at code location $2 \cdot 3 = 6$ and decompresses 3 codes. It writes its output starting at byte offset $10 + 3 = 13$.
This format ensures that every thread processes the same number of codes, but it may cause some threads to output much more data than others. For instance, a thread that mostly handles codes corresponding to 8-byte symbols would output significantly more data than one that processes only escaped codes.

The constant uncompressed size format (from Figure 4.2b) is chosen for the implementation. In general, decompression algorithms are expected to write more data than they read, which is why it's more important to balance the write operations than the read operations.

This format introduces a new parameter, the number of splits (N_{splits}). The size of each split (S_{split}) is calculated from the total uncompressed size of the block (S_{block}) and the number of splits (N_{splits}) using Equation Equation (4.3). This ensures that the input data is evenly divided among the splits, with at most a difference of one byte between them.

$$S_{split} = \left\lfloor \frac{S_{block} + N_{splits} - 1}{N_{splits}} \right\rfloor \tag{4.3}$$

The extra storage required for storing the uncompressed split sizes is calculated using Equation (4.4). This overhead increases with both the number of blocks and the number of splits, and is larger than the overhead for storing the block size (as calculated in Equation (4.1)).

$$S_{overhead\ uncompressed\ split\ sizes} = 3 \cdot N_{splits} \cdot N_{blocks} \tag{4.4}$$

Handling escape characters in splits format

When the compression algorithm is dividing a block into splits, it should take the escape codes into account to prevent it from generating corrupt data. If the data is split between an escape code and its escaped character, two decompression hazards are caused:

- A thread will end its compressed buffer with an escape code without being able to access the escaped character, missing one byte in its output buffer, or outputting a random value from memory.
- A thread will start it's buffer with an escaped character, without knowing it is an escaped character, causing it to be interpreted as a symbol lookup, resulting in a longer, incorrect output.

To prevent these issues, each split is compressed independently. While the symbol table is constructed using the entire block of data, the actual replacement of data with codes is performed on a per-split basis. This ensures that a split never ends with an escape code, avoiding these decompression hazards.

4.2.3. Coalesced Memory Access Format

To make the best use of GPU memory structure, it is recommended to access data according to a coalesced memory access pattern [66]. Doing this allows the GPU to stripe data over its data banks, making consecutive memory accesses completely parallel instead of sequential. However, the splits format discussed in Section 4.2.2 does not adhere to coalesced memory access. In the splits format, each thread works on its own portion of the input data, meaning that adjacent threads do not access adjacent memory locations.

Inspired by the transposed data layouts used in FOR, Delta, and RLE compression formats from [67], a new layout is proposed for better coalescing memory accesses in GPU memory when reading data from compressed GSST blocks. The goal is to adapt the data format to match the GPU's memory architecture, better utilizing memory throughput.

This new layout still uses the splits format as its base but reorganizes the codes to ensure that the first code accessed by each thread is stored in adjacent memory locations. Instead of each thread working on separate portions, the codes are rearranged so that threads access their first code consecutively, followed by all threads accessing the second consecutively, and so on until N_{splits}. In other words, the codes section of the block starts with the first code from split 1, then the first code of split 2, and so on until N_{splits}, before moving to the second code from each split, as illustrated in Figure 4.4.

Like the splits format, this coalesced format can divide the codes into splits in two ways:

1. **Constant uncompressed size**(Figure 4.2c): In this format, the codes use the same division over splits as in Figure 4.2a. Calculating the offset of the output is done by multiplying the split number by the constant uncompressed size (same as for Figure 4.2a). The variable compressed size causes splits to contain different numbers of codes. Because of this, the need for filler data is introduced. In the example, the first split has 1 code, and the second split has 8 codes. After coalescing the codes, there arises an imbalance between these two splits, as split 1 has no more codes to provide for the second coalesced block of codes, while split 2 has 7 more to output. This leads to the need for padding codes on the places where split 1 can't provide any more codes. Needless to say, it is very inefficient to apply the coalescing on this format.

2. **Constant number of codes** (Figure 4.2d): In this format, the codes use the same division over splits as in Figure 4.2b. Outputting the data per split works the same as for the split format. The constant compressed size of this format makes sure that all splits have (almost) the same number of codes. This results in a nice even distribution of the codes into the coalesced blocks of codes, resulting in a number of coalesced blocks equal to number of codes in the block, divided by the number of splits, as shown in Equation (4.5). Because of this, only the last coalesced block may require filler codes, the others will always be fully filled with codes containing data.

$$\begin{aligned} N_{\textit{coalesced blocks of codes}} &= N_{\textit{codes in longest split}} \\ &= \left\lceil \frac{N_{\textit{codes in block}}}{N_{splits}} \right\rceil \quad \textit{(When using "constant number of codes")} \end{aligned} \tag{4.5}$$

It's clear that using a constant number of codes per split is more effective for coalesced memory access, so this version will be implemented and used throughout the remainder of the thesis.

For both formats, once the codes are coalesced, the codes for a split are located at positions calculated by the formula:

$$codelocation(x) = split\ number + x \cdot N_{splits} \tag{4.6}$$

where x continues until the end of the codes section.

It's worth noting that while this reordering better matched the compression format with the GPU memory architecture, it also results in the loss of some advanced features of GSST, such as operating directly on compressed data, which is a key feature inherited from FSST. These features, fall outside the scope of this thesis.

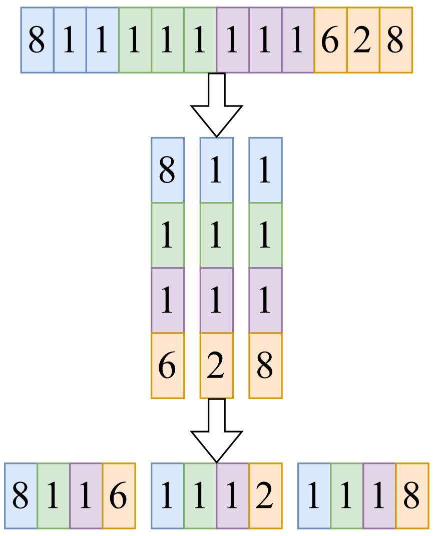


Figure 4.4: Conversion of a buffer with 12 codes and 4 splits into the coalesced format.

Handling escape characters in coalesced format

Reordering the codes in the compressed buffer comes with new decompression hazards. The hazards occur when dividing the block into splits. The situation can occur that the compressed data is split between an escape code and its escaped byte. The normal decompression algorithm will not work, as one thread its input ends with an escape code and can't find the escaped byte, the following thread's input buffer starts with the escaped byte, without knowing it is escaped. For the coalesced format, three solutions are considered:

1. Append the escaped byte to the split with the escape code.
2. Prepend the escape code to the split with the escaped byte.
3. Allow separating an escape symbol and its escaped character, and adapt the decompression algorithm to detect this.

The first two solutions, which involve moving data between splits, were not preferred as they would overly complicate the compression. Fortunately, an easier solution was found. The decompression algorithm can be adapted to handle split escape code efficiently. Before starting decompression, each thread checks if the last code of the preceding split is an escape code. If it is, the first code in its own split is treated as an escaped character. If not, the first code is processed normally. Additionally, if a thread encounters an escape code as the final code in its split, it can simply ignore it because it has already been handled by the preceding split.

4.3. Memory Management Optimizations

Decompression algorithms are typically memory-bound operations. This combined with the high computational throughput of GPUs, makes that using the memory bandwidth optimally is at most important for these algorithms. This section will discuss what measures have been taken to squeeze maximum performance out of the memory of the GPU.

4.3.1. Shared Memory

Shared memory refers to the L1 cache memory that can be used as a scratchpad memory, allowing the programmer to directly control it. Data in shared memory is accessible by all threads within the same thread block. It is designed to reduce the need for threads to access global memory, which is orders of magnitude slower. By using shared memory, threads can collaborate and share data efficiently, leading to significant performance improvements.

In GSST, the block decompression starts with parsing the block header. Because of the small size of the header, it is possible to fully store its contents in shared memory. A single thread reads the header data from global memory and constructs a data structure in shared memory, which includes the symbol table. The symbol table is very frequently accessed by all threads within the thread block, so keeping it in shared memory will provide quick access to it, without ever experiencing cache misses. Once the header is parsed, the threads begin the parallel decompression of data splits, where each thread retrieves the necessary information from shared memory and decompresses its portion using the shared symbol table. By keeping the symbol table in shared memory, only one copy is needed for all the threads, rather than creating separate copies for each one.

After the block header is loaded into shared memory, the primary use of global memory comes from reading compressed codes and writing decompressed data. To optimize this process, shared memory is used to buffer this data. However, shared memory is limited in size and can't store all the data at once, so an *input buffer* and *output buffer* are created for each thread in the thread block. These buffers are responsible for staging data between global memory and shared memory. The input buffer reads chunks of compressed data from global memory, and once the data is decompressed, the next chunk is loaded. Similarly, the output buffer temporarily stores decompressed data, which is then written back to global memory once the buffer is full.

Using these input and output buffers in shared memory improves the memory access latency of the decompression algorithm. During decompression, there are frequent memory accesses to both the compressed input data and the decompressed output data. However, once the data is decompressed, it is not accessed again. By keeping the hot data in shared memory, the algorithm minimizes cache misses during reads and avoids frequent write operations to global memory. Instead, the input buffer ensures that all accesses to compressed data are efficient, while the output buffer stitches together small write operations and only writes to global memory when necessary, improving overall throughput.

4.3.2. Memory Alignment

Memory alignment is a hardware-related design choice that affects how data is stored and accessed in memory. Modern hardware stores and accesses data in groups of bytes, such as 4 or 8 bytes at a time, rather than accessing individual bytes one by one. This is because memory is organized in fixed-size blocks, or *alignment boundaries*, that allow for more efficient data transfers when accessing entire blocks at once. If data is not aligned to these boundaries, the hardware has to perform extra steps, such as accessing multiple blocks and combining the relevant data, which results in wasted memory and computation throughput.

Aligning the memory accesses of GSST is split into two parts, the input alignment and the output alignment. Both use a similar approach to be able to utilize aligned transfers. The full process is illustrated in Figure 4.5.

Both input and output alignments rely on calculating the first aligned address in an unaligned buffer. The byte offset to this address can be calculated using the starting address of the buffer and the size of the alignment, as shown in Equation (4.7).

$$S_{offset} = (S_{align} - (A_{input} \mod S_{align})) \mod S_{align} \tag{4.7}$$

- $A_{input} \mod S_{align}$ gives the current alignment of the pointer.
- $S_{align} - (A_{input} \mod S_{align})$ calculates how many bytes you need to move forward to reach a S_{align}-byte boundary.
- The final modulo operation ensures that if the pointer is already aligned, the offset will be 0.

By adding this offset to the base address of the buffer, the first aligned address in the buffer can be calculated:

$$A_{aligned} = A_{input} + S_{offset} \tag{4.8}$$

Reading Input Alignment

In a GSST compressed buffer, the data is not naturally aligned because of the variable block and split sizes used during compression. Typically, the first block in the buffer starts at an aligned address, as memory buffers are usually allocated at alignment boundaries. However, as shown in the Figure 4.1d,

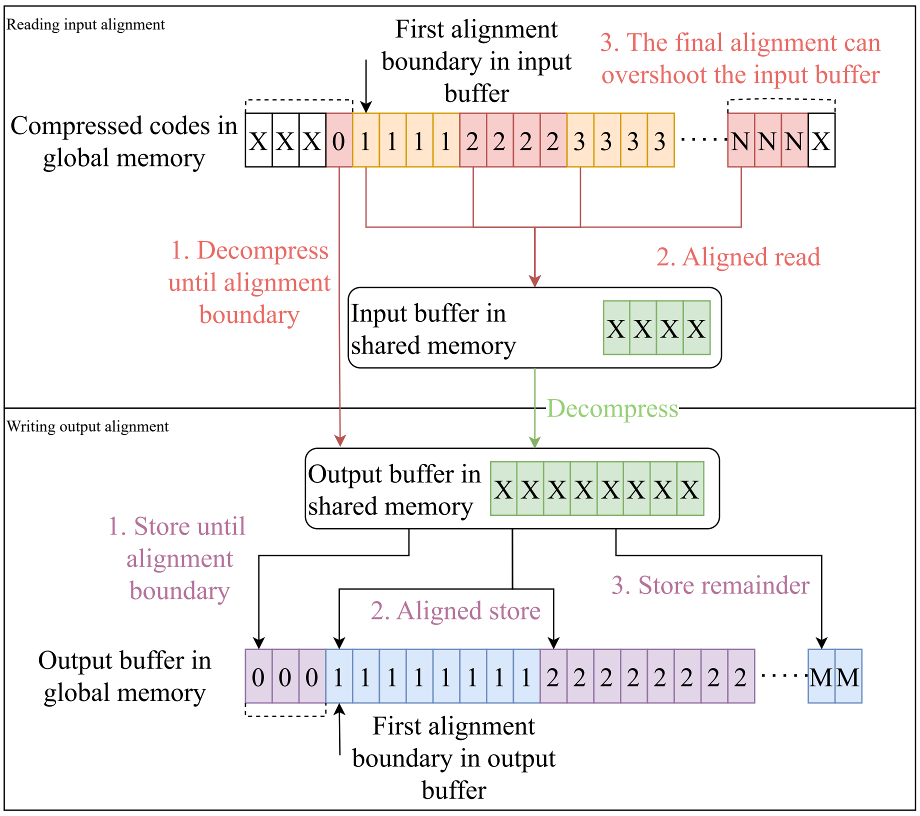

Reading input alignment
First alignment boundary in input buffer
3. The final alignment can overshoot the input buffer
Compressed codes in global memory
X X X 0 1 1 1 1 2 2 2 2 3 3 3 3 ····· N N N X
1. Decompress until alignment boundary
2. Aligned read
Input buffer in shared memory
X X X X
Decompress
Writing output alignment
Output buffer in shared memory
X X X X X X X X
1. Store until alignment boundary
2. Aligned store
3. Store remainder
Output buffer in global memory
0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 ····· M M
First alignment boundary in output buffer

Figure 4.5: Demonstration of how aligned data transfers are realized in an unaligned buffer in global GPU memory. In this example, the input pointer is aligned on a 4 byte boundary and the output pointer on 8 bytes. The input data is aligned by reading and decompressing individual bytes until the first aligned address in the input buffer is reached. Likewise, the output buffer is aligned by writing individual bytes until the first aligned address in the output buffer. After this, the majority of the decompression uses aligned data transfers. Input data is read from global memory to the buffer in shared memory by (4 byte) aligned transfers. This data is decompressed to the output buffer in shared memory. When the output buffer is full, it uses an (8 byte) aligned transfer to store the decompressed data to global memory. Finally, the alignment overshoot in the input buffer is handled, and the remainder in the output buffer is stored.

the length of the block headers varies depending on the number of splits within the block. This means that the starting address for the codes section within a block is often unaligned. Similarly, the locations of splits within the block are also unaligned. The blocks that follow are affected by this as well, meaning they likely do not begin at aligned addresses either, due to the variable block sizes of the preceding blocks. Essentially, the only reliably aligned part of the compressed data is the start of the first block.

During decompression, the input buffer is aligned to allow for efficient processing. This process, shown in the top part of Figure 4.5, begins by calculating the first aligned boundary after the start of the input buffer, using Equation (4.7). From this aligned point onward, data is transferred from global memory to shared memory can be done using full aligned transfers. However, since data before the alignment boundary still needs to be decompressed, the decompression algorithm handles it using unaligned memory access, writing it into the output buffer in shared memory.

Once the data up to the alignment boundary is decompressed, the remaining data can be read using aligned transfers into shared memory. At the end of the buffer, the final alignment might extend past the actual data because the buffer does not always end at an aligned boundary. The decompression step accounts for this by tracking how many codes have been processed to ensure it stops at the correct point, avoiding decompressing the overshoot data.

Writing Output Alignment

The output data from each thread is first stored in shared memory before being transferred to global memory. The specific location in global memory is determined by the number of blocks and number of splits used during compression. Because of these variables, the starting address for the output data in global memory doesn't align with memory boundaries.

The decompression process begins by aligning the input buffer, as described in the previous section. After that, the output buffer is aligned. This process is shown in the lower part of Figure 4.5. Aligning the output buffer, begins by calculating the first aligned boundary after the start of the output position, using Equation (4.7). From that point onward, data can be transferred using aligned transfers. However, before reaching that aligned boundary, the data needs to be placed in the unaligned section.

To handle this, the decompression process begins by reading data into shared memory, decompressing it, and storing it there until enough data has been produced to fill the unaligned section of the output buffer. This unaligned data is then transferred to global memory. If more data than necessary has been decompressed into shared memory, the excess is moved to the beginning of the buffer in shared memory.

Once the unaligned section is filled, the decompression process continues. The system fills the rest of the output buffer in shared memory and, when full, moves it to global memory using a single aligned transfer. Finally, when the decompression is complete, any remaining data in the shared memory output buffer is transferred and added to the global memory.

4.3.3. Asynchronous Memory Access

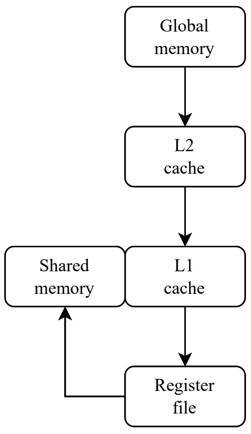
Introduced in CUDA 9, the new *cuda::memcpy_async()* API provides a new mechanism for performing data transfers between global memory and shared memory on the GPU [68], [69]. This API is distinct from *cudaMemcpyAsync()*, which handles data transfers between the CPU's main memory and the GPU's global memory. In contrast, *cuda::memcpy_async()* works entirely on the GPU, facilitating the movement of data from global memory directly into shared memory (L1).

On NVIDIA GPUs with a compute capability of 8.0 or higher, such as Ampere architectures and beyond, *cuda::memcpy_async()* transfers benefit from hardware acceleration in two ways:

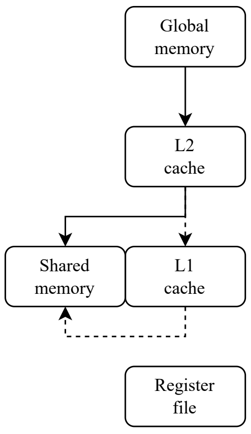
1. The data is copied directly from global memory, passing through the L2 cache, into shared memory. This data path is shown in Figure 4.6b. Notably, this process bypasses the SM's registers.

2. The transfer is asynchronous, meaning it can overlap with computation, allowing the GPU to execute other tasks while waiting for the data transfer to complete.

This API comes with certain constraints. Specifically, all threads within a block must collaborate simultaneously to perform the asynchronous transfer, and they must work on a contiguous chunk of memory. As a result, *cuda::memcpy_async()* cannot handle asynchronous transfers initiated by individual threads working on non-contiguous memory regions. This requirement requires a coalesced memory layout to fully leverage the API's capabilities, where memory accesses are aligned and contiguous across threads.



(a) Normal data path when reading data from global to shared memory in the GPU



(b) Data path when reading data from global to shared memory using *cuda::memcpy_async()*

Figure 4.6: Data paths of moving data from global memory to shared memory in a GPU.

To maximize performance when using the *cuda::memcpy_async()* API, NVIDIA's documentation advises that both the global and shared memory should be aligned to 128 bytes [66]. This alignment ensures that data is transferred more efficiently. Furthermore, the API supports copying data in blocks of 4, 8, or 16 bytes at a time. For optimal performance, the size of the data being transferred should be a multiple of these values. Additionally, both the source and destination pointers passed to *cuda::memcpy_async()* must be aligned to 4, 8, or 16-byte boundaries. When using these alignments, *cuda::memcpy_async()* can fully utilize the GPU's asynchronous memory operations, achieving maximum throughput. Aligning the unaligned transfers between global and shared memory is discussed in Section 4.3.2.

5 GSST Performance Parameters

This chapter outlines the experiments that are employed to evaluate and tune the parameters for the GSST decompression algorithms on the testing setup. First, the parameters itself are described and how they are used in the algorithm. Then experiments are conducted to observe how the parameters influence the performance of the algorithm, and to find parameters that achieve the highest decompression throughput on the testing setup.

5.1. Setup

All benchmarks are executed on the same server with the specifications specified in Table 5.1. The tuning of the algorithm is done for this specific hardware configuration. The specifications of the GPU are shown in Table 5.2, and will be used to derive the parameters for the algorithm. This approach not only optimizes performance for the current GPU but also provides a foundation for deriving configurations for other GPU models.

The following software versions are used:

- nvCOMP 4.0.1
- NVIDIA driver 545.23.08
- CUDA 12.3
- NVIDIA Nsight Systems version 2024.4.1.61-244134315967v0
- NVIDIA Nsight Compute Command Line Profiler Version 2023.3.1.0 (build 33474944)

5.2. Testing Data

The choice of data used for benchmarking significantly influences its results. Section 1.4 specifies that the focus is on text data, meaning that all testing and tuning will be performed using large volumes of text. Once the optimal parameters are identified for this text data, the same settings will be used to benchmark on other types of data too, for comparison.

CPU	2x Intel Xeon Platinum 8380 2.30GHz, 80 HW cores total, 2 threads per HW core
Memory	32x 128GB 3200MHz, 4TB total
GPUs	8x NVIDIA A100 80GB
Network	8x 200GbE
Storage	8 x 15 TB Samsung PM9A3

Table 5.1: Hardware specifications of the compute nodes used to perform the experiments

Chip	GA100
Cores	6912
Cores per SM	64
Compute capabilities	8.0
SM Count	108
Memory size	80 GB
Memory type	HBM2e
Memory bandwidth	2.04 TB/s
Base Clock	1275 MHz
Boost Clock	1410 MHz
L1 Cache	192 KB per SM
Shared Memory	up to 164 KB per SM
L2 Cache	40 MB
Cache Line Size	128 Bytes
Warps per SM	64
Threads per warp	32
Threads per SM	2048
Thread blocks per SM	32
Threads per block	1024

Table 5.2: NVIDIA A100 SXM4 80 GB specifications [60]

In Chapter 3, it was derived that file size does not affect ingestion throughput. However, to accurately calculate the model's ingest throughput, it is important to have an accurate measurement of the decompression throughput (T_{dout}) of the decompression algorithms used. The size of the input test data can influence the measurement of decompression throughput. The goal is to demonstrate that the decompression algorithm can maintain consistent throughput under a sustained load.

For this purpose, a 10 GB buffer of TPC-H text data was created using the dbgen tool [38]. This size was selected because it is the upper limit for certain nvCOMP compression algorithms in the testing environment. Any data exceeding this limit would result in out-of-memory errors. These errors are further highlighted in Section 6.4. This dataset is used during all testing, experiments, and benchmarks.

All performance measurements presented in this thesis were conducted three times, with the median result used for analysis.

5.3. Parameters

Parameter	Variable	Configurable	Explanation
Number of blocks	N_{blocks}	During compression at runtime	Divide the input in this number of blocks
Number of splits	N_{splits}	During compression at runtime	Divide a block into this number of splits
Alignment size	S_{align}	At compile time	Align buffers on this boundary and move data in transfers of this size
Input buffer size	$S_{\text{input buffer}}$	At compile time	Buffer this many input bytes in shared memory
Output buffer size	$S_{\text{output buffer}}$	At compile time	Buffer this many output bytes in shared memory
Number of thread blocks	$N_{\text{thread blocks}}$	During decompression at runtime	Use this number of decompression thread blocks
Number of threads per block	$N_{\text{threads per block}}$	During decompression at runtime	Use this number of decompression threads in a thread block

Table 5.3: GSST performance parameters

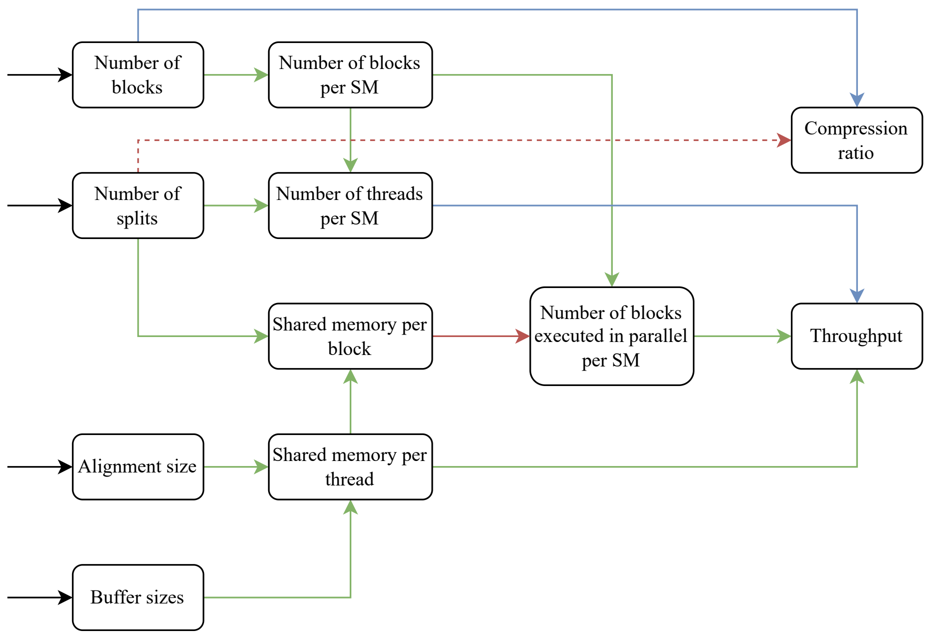


Figure 5.1: The impact of the four input parameters on the performance metrics of the decompression algorithm: compression ratio and throughput. A green arrow indicates that increasing one value leads to an increase in another, while a red arrow indicates that raising one value will cause a decrease in the other. Blue arrows represent cases where the effect of a change in value is uncertain.

Figure 5.1 shows how the parameters of the algorithm impact the two performance metrics, the compression ratio and decompression throughput. By isolating the path from the four parameters to the two metrics, test configurations derived where the impact of the parameters are isolated.

5.3.1. Number of Blocks

The GSST compressors compress data into independent blocks of data, as explained in Section 4.2.1. The number of blocks parameter tells the compression algorithm in how many blocks of data should be compressed, and indirectly determines the block size, the number of bytes in a block.

The number of blocks determines the level of parallelism on block level.

During decompression, a thread block can decompress one or more data blocks. A data block is always decompressed by a single thread block. In cases where the number of thread blocks is fewer than the total number of data blocks, multiple data blocks are automatically assigned to each thread block. The distribution of data blocks is designed to be as even as possible, meaning that each thread block processes a roughly equal number of data blocks, with the maximum difference being one.

On the other hand, if there are more thread blocks than data blocks, the extra thread blocks that don't have any data assigned to them will terminate automatically. This dynamic management of thread blocks ensures efficient resource utilization and minimizes any idle threads during the decompression process.

$$N_{\textit{thread blocks}} \leq N_{blocks} \tag{5.1}$$

5.3.2. Number of Splits

During decompression, each compressed block is assigned to a single Streaming Multiprocessor (SM) on the GPU. To achieve parallelism within an SM, the compressed block is subdivided into smaller segments called splits, as discussed in Section 4.2.2. The number of splits dictates the maximum level of parallelism within a block, as splits can be processed in parallel by the cores in the SM. Overall parallelism, therefore, depends on two factors: the number of blocks being decompressed and the number of splits each block is divided into, as outlined in Equation (5.3).

Increasing the number of splits typically boosts throughput, as more parts of the block are handled in parallel. However, increasing the number of splits also increases the size of the block's header, because more split locations need to be stored.

Moreover, the number of SMs and cores on a GPU is limited, and the number of active blocks that can run simultaneously is constrained by the hardware, as shown in Table 5.2. Therefore, dividing a block into too many splits can lead to diminishing performance returns. When too many splits are created, the overhead involved in managing them can outweigh the benefits of parallelism, and the per-core workload becomes too small to fully utilize the GPU's processing power.

The number of splits directly affects how much data each core processes. As more splits are introduced, the amount of data handled per split decreases. While this may initially seem advantageous for load balancing, it can lead to inefficiencies if the workload per core becomes too small, causing underutilization of available GPU resources. The challenge lies in finding the right balance: too few splits will underutilize the GPU's parallel processing capabilities, while too many splits will create overhead that degrades performance.

For optimal performance, the number of threads started in each thread block should always match the number of splits in the block. If this is not the case, threads would need to handle more than one split, which negatively affects throughput. It leads to increasing the thread divergence and causing an uneven workload distribution among threads, complicating the implementation and reducing efficiency. Thus, in all tests, experiments, and benchmarks, the number of threads per block is set equal to the number of splits, as expressed in Equation (5.2).

$$N_{\textit{threads per block}} = N_{splits} \tag{5.2}$$

$$N_{threads} = N_{blocks} \cdot N_{splits} \tag{5.3}$$

$$N_{\textit{warps per block}} = \left\lceil \frac{N_{splits}}{32} \right\rceil \tag{5.4}$$

5.3.3. Alignment Size

Data transferred between global and shared memory are done in transactions equal to the alignment size. As discussed in Section 4.3.2, the decompression algorithm begins by aligning the input and output buffers in global memory. It does this by decompressing until the first aligned address in the input buffer is reached, and storing data until the first aligned address in the output buffer is reached. The alignment size determines the maximum distance to this first aligned address. When the alignment size is increased, the maximum distance to the first aligned address also increases, which can lead to more time being spent aligning the buffers. In the worst case scenario, if the alignment size is set equal to the size of the split, no aligned transfers can occur, and the entire chunk will be decompressed using unaligned accesses.

Since the size of data transfers between global and shared memory is determined by the alignment size, the buffer sizes are also tied to this parameter. Each buffer consists of one or more aligned boundaries, so the alignment size dictates the minimum buffer size. A smaller alignment size is preferred because it allows for more precise control over buffer sizes.

The alignment size is a configurable parameter that is set during the compilation of the algorithm, to allows the buffers in shared memory to be statically allocated. This eliminates the dynamic allocation overhead during kernel launch. It also allows the compiler to optimize data transfers using larger load and store instructions. Which instruction are possible depends on the (compute) capabilities of the device the algorithm will run on. The goal of the alignment size is to maximize the amount of data transferred in a single instruction. The GPU supports large load and store instructions that are needed to fully utilize its high memory bandwidth. However, without the right alignment, the compiler can't optimize code to use these large instructions. The alignment size guarantees the compiler that the buffers are aligned on that boundary, such that it can apply the optimizations to increase the size of the instructions used for the transfers.

To determine the correct alignment size for the A100 GPU, the code is recompiled with various alignment sizes, and the generated Parallel Thread Execution (PTX) code is analyzed The aim is to keep the alignment size as small as possible to minimize overhead, while also optimizing the number of bytes transferred per instruction. As shown in Table 5.4, modifying the alignment size affects the instructions used for data transfers between shared and global memory. The maximum number of bytes that can be transferred in a single instruction is 16. Increasing the alignment size beyond this point only results in

Alignment size (bytes)	Reading the input buffer	Writing the output buffer	Bytes per transfer
2	ld.u64, st.shared.u64	ld.shared.u64, st.u64	2
4	ld.u64, st.shared.u64	ld.shared.u64, st.u64	4
8	ld.u64, st.shared.u64	ld.shared.u64, st.u64	8
16	ld.v2.u64, st.shared.v2.u64	ld.shared.v2.u64, st.v2.u64	16
32	2 × ld.v2.u64, 2 × st.shared.v2.u64	2 × ld.shared.v2.u64, 2 × st.v2.u64	16
64	4 × ld.v2.u64, 4 × st.shared.v2.u64	4 × ld.shared.v2.u64, 4 × st.v2.u64	16
128	8 × ld.v4.u32, 8 × st.shared.v4.u32	8 × ld.shared.v4.u32, 8 × st.v4.u32	16

Table 5.4: The effects of the alignment size on the PTX when the GSST splits kernel is compiled for compute capability 8.0. These are the instructions used to load and store a full alignment between shared and global memory.

more transfer instructions being generated. Based on this analysis, an alignment size of 16 bytes will be used for the remaining experiments and benchmarks.

5.3.4. Input and Output Buffer Size

On block level, the algorithm uses buffers in shared memory to stage input and output data. Compressed data is read from global memory to the input buffer in shared memory. This data is decompressed to the output buffer in shared memory. The output buffer is then moved to global memory.

The alignment size (S_{align}) parameter from Section 5.3.3 determines the size of the transfers between global and shared memory. The buffers are a multiple of the alignment size in size, meaning they will be filled or emptied in multiple aligned transfers.

Increasing the size of these buffers allows threads to process the input data for longer periods without interruption, which reduces the overhead associated with managing the start and end of the buffer. However, as these buffer sizes grow, the shared memory used by each block also increases significantly, putting more strain on the GPU's L1 cache.

$$S_{\mathit{input\ buffer}} = S_{align} \cdot N_{\mathit{input\ alignments}} \tag{5.5}$$

$$S_{\mathit{output\ buffer}} = S_{align} \cdot N_{\mathit{output\ alignments}} \tag{5.6}$$

5.4. Test Configurations

The previous section, Section 5.3, discussed the different parameters of the algorithm that can be adjusted to improve its performance. The challenge with tuning these parameters is that changing one affects the ideal settings for the others, turning it into a multivariable optimization problem. This section describes the tests carried out to find parameters that are close to optimal, using the specifications of the GPU hardware as a reference. The method for finding the best parameters is inspired by the approach outlined in [70].

5.4.1. Number of Data Blocks and Thread Blocks

This experiment examines whether any performance benefits can be achieved by creating more data blocks than thread blocks. In such cases, a single thread block would need to process multiple data blocks. A 10 GB text file is used as the input, which is divided into blocks of varying sizes as shown in Table 5.5. The block sizes range from many small blocks to a few large ones. For each of these block sizes, the decompression process is tested with a varying number of thread blocks, ranging from 128 to 2048. This setup helps to explore three different scenarios:

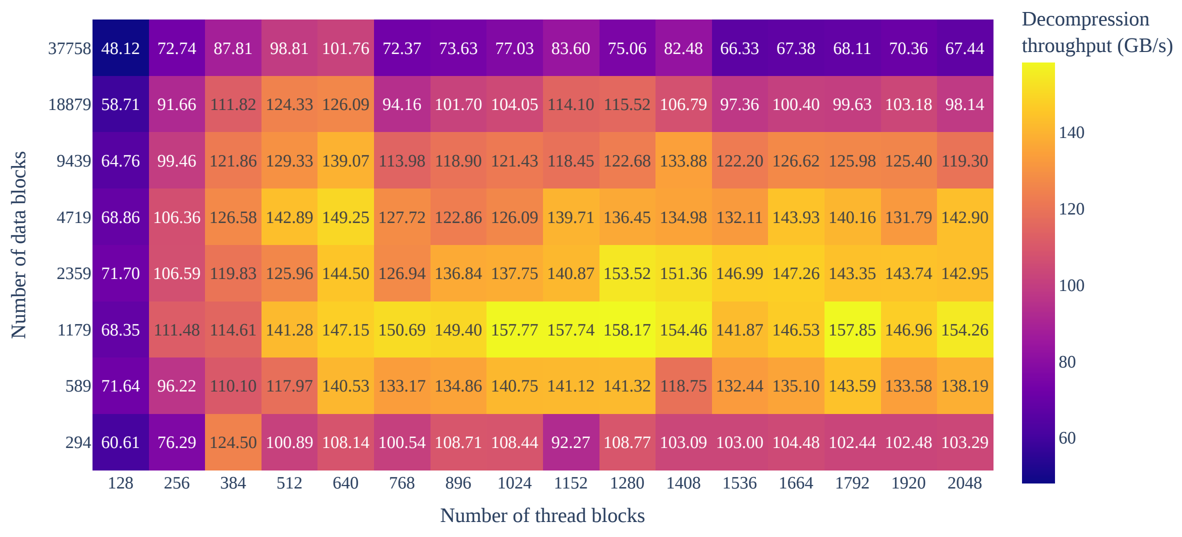
1. Having more thread blocks than data blocks
2. Having fewer thread blocks than data blocks
3. Having a nearly equal number of thread blocks and data blocks

Three key conclusions can be drawn from the results shown in Figure 5.2:

1. Having more thread blocks than data blocks is acceptable, but performance slightly decreases if the difference becomes too large.
2. Creating more data blocks than thread blocks leads to a significant drop in throughput.

S_{block}	N_{block}
$2^{18} = 262144$	37758
$2^{19} = 524288$	18879
$2^{20} = 1048576$	9439
$2^{21} = 2097152$	4719
$2^{22} = 4194304$	2359
$2^{23} = 8388608$	1179
$2^{24} = 16777216$	589
$2^{25} = 33554432$	294

Table 5.5: Test configurations for number of data blocks, used in the number of data and thread blocks expirement.



	128	256	384	512	640	768	896	1024	1152	1280	1408	1536	1664	1792	1920	2048
37758	48.12	72.74	87.81	98.81	101.76	72.37	73.63	77.03	83.60	75.06	82.48	66.33	67.38	68.11	70.36	67.44
18879	58.71	91.66	111.82	124.33	126.09	94.16	101.70	104.05	114.10	115.52	106.79	97.36	100.40	99.63	103.18	98.14
9439	64.76	99.46	121.86	129.33	139.07	113.98	118.90	121.43	118.45	122.68	133.88	122.20	126.62	125.98	125.40	119.30
4719	68.86	106.36	126.58	142.89	149.25	127.72	122.86	126.09	139.71	136.45	134.98	132.11	143.93	140.16	131.79	142.90
2359	71.70	106.59	119.83	125.96	144.50	126.94	136.84	137.75	140.87	153.52	151.36	146.99	147.26	143.35	143.74	142.95
1179	68.35	111.48	114.61	141.28	147.15	150.69	149.40	157.77	157.74	158.17	154.46	141.87	146.53	157.85	146.96	154.26
589	71.64	96.22	110.10	117.97	140.53	133.17	134.86	140.75	141.12	141.32	118.75	132.44	135.10	143.59	133.58	138.19
294	60.61	76.29	124.50	100.89	108.14	100.54	108.71	108.44	92.27	108.77	103.09	103.00	104.48	102.44	102.48	103.29

Figure 5.2: Heatmap showing the decompression throughput for various combinations of compressed data blocks and thread blocks used for decompression. The results indicate that the algorithm performs optimally when the number of decompression thread blocks matches the number of compressed data blocks.

3. The highest throughput occurs when the number of thread blocks is approximately equal to the number of data blocks.

Therefore, the best performance is achieved when each thread block processes a single data block. Based on this observation, the number of thread blocks will be set equal to the number of data blocks in subsequent experiments (Equation (5.7)).

$$N_{thread\ block} = N_{block} \tag{5.7}$$

5.4.2. Shared Memory Size and Block Scheduling

This experiment shows the effects of 'The number of blocks per SM' and 'Shared memory per block' on the throughput of the algorithm. The algorithm uses shared memory in L1 cache of each SM to stage compressed and decompressed data. Each data block is decompressed by a single thread block, and each thread block uses a configurable amount of shared memory.

An SM in the GPU can be assigned multiple thread blocks at the same time. Doing this allows the GPU to switch the execution from one thread block to another when one gets stalled. The switching generally improves performance, as it hides latencies of one block with another. However, all blocks assigned to an SM have their data stored in the L1 cache of the block to allow the GPU to quickly switch the execution. So, before a block can be assigned to an SM, the SM needs to have enough space left in its

Shared memory per block	Total shared memory per SM	Expected behavior
10 KB	160 KB	16 blocks fit at the same time (fastest)
20 KB	320 KB	8 blocks fit at the same time
30 KB	640 KB	6 blocks fit at the same time
40 KB	1280 KB	4 blocks fit at the same time (slowest)

Table 5.6: Test configurations used when testing the impact of the shared memory size on throughput of the algorithm

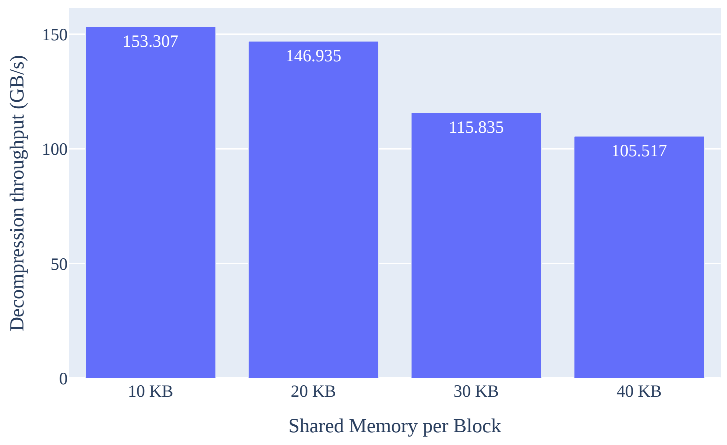


Figure 5.3: Graph showing how increasing the shared memory size per block impacts the algorithm's throughput. Since the additional shared memory wasn't actively utilized by the algorithm, it highlights that any performance gains from increasing shared memory must outweigh the overhead introduced by this increase.

L1 cache (shared memory) to fit the new block.

If one thread block takes up all the shared memory of the SM, the GPU might be able to only schedule that single block on the SM, preventing it from taking advantage of the latency hiding of multiple blocks. The purpose of this experiment is to verify whether this is the case and to quantify how much the size of shared memory per block influences the algorithm's throughput.

In order to achieve this, a compressed data buffer is generated with 64 blocks of compressed data for each SM in the GPU: $N_{blocks} = N_{thread\ blocks} = 16 \cdot N_{SM} = 1728$ Each block is configured with 32 splits, 1 warp, to prevent the warps from influencing this test.

During the experiment, the decompression algorithm is called multiple times on the same compressed data, while artificially increasing the amount of shared memory used by each block. This means that the shared memory of each block is increased, without the algorithm using the extra allocated shared memory. The algorithm will always use the same part of the shared memory. This setup completely isolates the effect of using more shared memory on the throughput of the decompression algorithm.

Each SM has 192 KB of shared memory. The total shared memory per block used during the experiments is shown in Table 5.6. It is expected using less shared memory per block will allow the GPU to schedule more blocks per SM at the same time, allowing them to apply more latency hiding, increasing the throughput.

The results of the experiment, shown in Section 5.4.2, demonstrate that increasing the amount of (unused) static shared memory per block does indeed decrease the algorithm's throughput. This is likely due to the SM's limited shared memory, which restricts the number of blocks that can be scheduled simultaneously. However, the decrease in performance was less significant than expected. Increasing the shared memory by four times resulted in only a 32% reduction in throughput.

Although the exact cause is not conclusive, it is suspected that the scheduler's inability to assign multiple blocks to an SM at once plays a significant role. An alternative explanation might be that the increased time required to allocate the larger shared memory affects performance, though this is considered unlikely since the shared memory is statically allocated.

These findings suggest that increasing the shared memory per block must provide a significant enough performance boost to compensate for the loss in throughput caused by scheduling fewer blocks per SM.

Based on these insights, the following guidelines can be drawn for optimizing the parameters of the

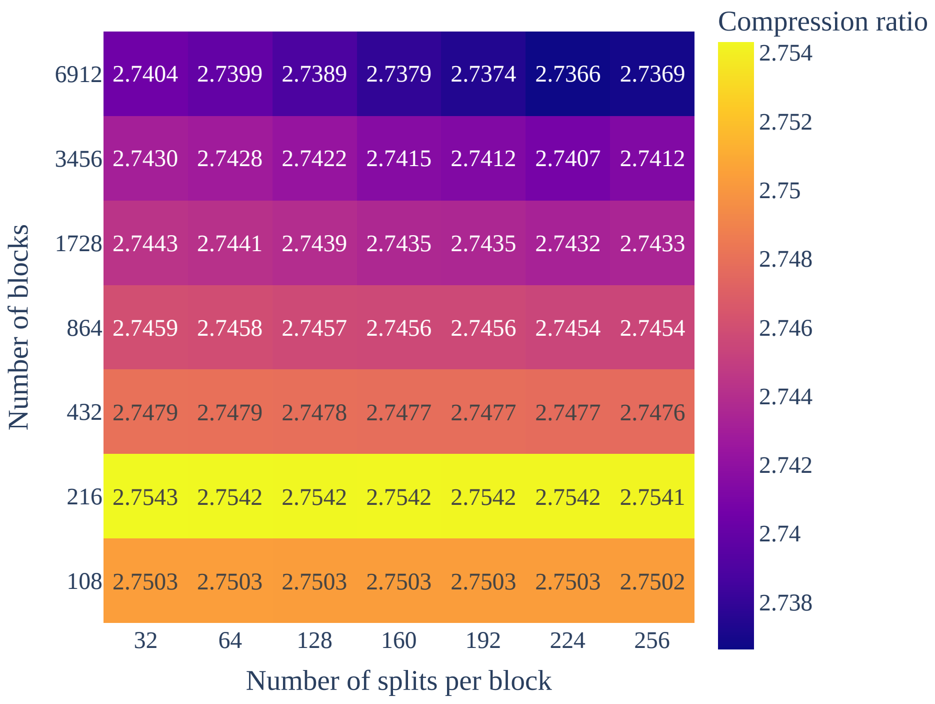


Figure 5.4: Heatmap showing how the compression ratio changes as an effect from changing the number of blocks and number of splits.

algorithm in Figure 5.1:

1. Use fewer blocks per SM with higher shared memory per block.
2. Use more blocks per SM with lower shared memory per block.

5.4.3. Compression Ratio

This experiment investigates how the number of blocks and splits in the input data affects the compression ratio. Although this thesis focuses primarily on decompression, adjustments to the compression algorithm have been made to better support the decompression stage. These modifications inevitably influence the resulting compression ratio.

Each block of data includes its own header, which contains a symbol table and information about the split locations. If a symbol table is used on a large portion of data, it can become inefficient, as it reaches its maximum capacity without capturing all the recurring patterns in the data. For example, when compressing a tar file that contains a variety of file types, such as text files, XML files, and CSV files, using the same symbol table for all file types can negatively affect the compression ratio. This is because the data patterns in an XML file are quite different from those in a CSV file, meaning that the entries in the symbol table created for one file type might not be useful for the others. To address this, creating a separate symbol table for each file type would significantly improve the compression ratio. The FSST binary, for instance, uses a separate symbol table for every 2 MB of data to manage this issue effectively [59].

The challenge lies in finding the ideal block size or number of blocks for a given input buffer to maximize the compression ratio. The optimal configuration is found by experimenting with different block and split numbers to see how they influence the compression ratio of the dataset.

From the results in Figure 5.4, the following conclusions are drawn:

1. The number of blocks and splits has a minimal effect on the compression ratio.

N_{block}	S_{block} (MB)
108	93
$2 \times 108 = 216$	47
$4 \times 108 = 432$	24
$8 \times 108 = 864$	12
$16 \times 108 = 1728$	6
$32 \times 108 = 3456$	3
$64 \times 108 = 6912$	2

Table 5.7: Table with the values used for the number of blocks parameter during the number of workers per SM experiment. The number of blocks is selected as a multiple of the GPU's 108 SMs. As the number of blocks increases, the block size decreases accordingly.

2. The highest compression ratio is achieved with 216 blocks and 32 splits, corresponding to 45 MB of data per block and 1.4 MB per split.
3. Increasing the number of splits slightly reduces the compression ratio.
4. Likewise, using too many blocks also leads to a slight decrease in compression ratio.

Given these findings, the impact of block and split overhead on the compression ratio is negligible. Therefore, the parameters for the number of blocks and splits will be chosen based on their impact on throughput rather than compression ratio.

5.4.4. Number of Workers per SM

This experiment explores how the 'number of splits' and 'number of blocks per SM' affect the throughput of the algorithm. To isolate these effects, the 'shared memory per block' is kept constant, allowing for a focus on how different configurations of blocks and splits affect performance. The findings in Section 5.4.2 tell us that using more shared memory decreases the throughput if it doesn't make the algorithm go faster. This is why the shared memory usage will be kept to a minimum. During all benchmarks an alignment size of 16 is used, together with an input buffer multiplier of 1 and output buffer multiplier of 4, resulting in using 80 bytes of shared memory per split.

It is expected that at least a single block per SM in the GPU is needed to make all SMs partake in the decompression. As shown in Table 5.2, a single SM is able to schedule 64 warps or 32 thread blocks at the same time. Having multiple warps available allows the GPU to apply latency hiding by switching to another warp when one is stalled. Increasing the number of warps in the SM can be achieved in 2 ways:

1. Increase the number of blocks, which increases the number of blocks per SM.
   Pros: Blocks are independent, making them effective for hiding latency.
   Cons: Creating more blocks uses more resources and overhead.
2. Increase the number of splits, which increases the number of warps per block.
   Pros: Adding splits involves minimal overhead.
   Con: Splits work on the same memory. When one thread is waiting for memory, all threads in the block are waiting for memory, making it less effective for latency hiding.

The default test set of 10 GB is used. In order to find the right trade-off between number of blocks and number of splits, all possible combinations of the number of blocks in Table 5.7 with the following number of splits: $N_{split} \in \{32, 64, 128, 160, 192, 224, 256\}$

From the results in Figure 5.5, the following conclusions can be drawn:

1. The highest throughput was achieved with 864 blocks and 192 splits per block. This configuration corresponds to 8 blocks per SM and 6 warps per block, resulting in 24 warps per SM.
2. Adding too many blocks reduces performance.
3. Increasing the number of splits has a more significant impact when there are fewer blocks.

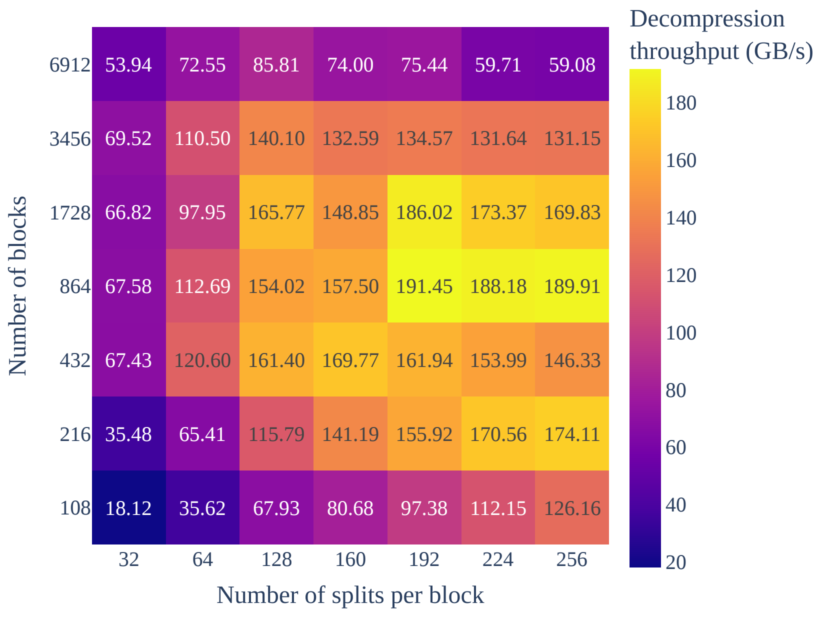


Figure 5.5: Heatmap of the decompression throughput achieved on the A100 GPU. All optimal parameters from previous experiments are applied, leaving only the effects of the number of blocks and splits on the throughput.

These results show that it is very important to carefully pick the number of blocks and splits during compression. The best configuration found here is specifically tuned for the A100 on a data size of 10 GB. In order to always get high throughput, the compression should pick a combination of number of blocks and splits that fits the device that will be used for decompression. For datacenter applications, this might be possible, as there is a small selection of devices used with similar performance characteristics. However, on consumer devices, the number of SMs, memory bandwidth vary widely, making it very difficult to find a configuration that achieves high throughput on all devices. Further research is needed to test which parameters achieve the highest throughput on other hardware configurations, and how this relates to the hardware of the device. It would be interesting to see if the ’8-16 of blocks per SM’ guideline holds.

6 Results

This chapter presents the performance of GSST, discusses the results, and evaluates if it reached its overall goal to increase data ingestion throughput on GPUs.

6.1. GSST Performance Development

In developing the GSST algorithm, several optimizations were introduced to increase the throughput. These optimizations fall into two categories: format optimizations (blocks, splits, and coalesce formats) and memory management optimizations (shared memory, memory alignment, and asynchronous memory access). Together, these techniques helped increase the algorithm's throughput, as illustrated in Figure 6.1.

The first GPU implementation, *GSST Blocks Kernel*, implements block parallelism, which results in very low throughput. This implementation is extended by adding splits in the *GSST Splits Kernel Initial*, leading to a ninefold increase in decompression throughput. Adding the use of shared memory in *GSST Splits Kernel Shared Memory* again tripled the throughput. The most significant improvement comes from the memory alignment in *GSST Splits Kernel Memory Alignment*, which boosts throughput nearly fivefold. Unfortunately, after implementing the coalesced format as the *GSST Coalesce Kernel*, the throughput unexpectedly dropped by about one-third. However, the coalesced format allows for asynchronous memory access, implemented as the *GSST Coalesce Kernel Async*, which improves its throughput by 15%. This still does not surpass the performance achieved with the splits format. As a result, further parameter tuning is done on the *GSST Splits Kernel Memory Alignment*, which leads to an additional 17% improvement in throughput, shown as *GSST Splits Kernel Parameter Optimized*.

GPU Utilization Analysis

All implementations were analyzed using NVIDIA Nsight Compute to identify the bottlenecks. The overall compute and memory throughput results are shown in Table 6.1. The first version,*GSST Blocks Kernel*, shows very low GPU utilization for both compute and memory throughput. This is expected since this version can only use a small number of threads in each thread block. In contrast, the *GSST Splits Kernel Initial* is able to utilize all threads in a block, leading to increased compute and memory throughput. However, both of these early versions show a high percentage of *Stall wait* and *Stall Long Scoreboard* as warp states, indicating that the warps are mostly idle, waiting for data from global memory. This observation leads to the decision to introduce the use of shared memory.

With shared memory implemented in the *GSST Splits Kernel Shared Memory*, the *Stall Long Scoreboard* warp states are replaced by *Stall Long Scoreboard* and *Stall MIO Throttle* states, meaning that warps are now waiting for access to shared memory instead of global memory. However, this version also reveals a new issue: when decompressing a 1.5 GB file, the SMs write a total of 52 GB to the L2 cache. This means that for each byte written to the output buffer, 35 bytes are written to the L2 cache. That is over 32 times higher than expected. The reason for this is inefficient memory alignment. Unaligned data transfers of 8 bytes are written as 8 times 1 byte transfers. The GPU uses 32-byte cache lines, resulting in each 1 byte transfer being 32 bytes instead.

Implementation	Memory Throughput (%)	Compute Throughput (%)
GSST Blocks Kernel	53.67	5.90
GSST Splits Kernel Initial	63.48	8.97
GSST Splits Kernel Shared Memory	71.12	10.46
GSST Splits Kernel Memory Alignment	92.05	59.57
GSST Coalesce Kernel	71.02	51.18
GSST Coalesce Kernel Async	53.51	52.51
GSST Splits Kernel Parameter Optimized	94.78	63.01

Table 6.1: Utilization percentages of the GSST implementations during the decompression of a 10 GB test dataset on the A100 GPU, as reported by NVIDIA Nsight Compute. The memory throughput indicates the highest utilization observed for all memory components, which could be the L1 cache, L2 cache, or global memory, depending on which was most heavily used. The compute throughput reflects the average percentage of GPU instructions executed relative to its theoretical maximum capacity.

To address this, memory alignment is implemented in the *GSST Splits Kernel Memory Alignment*, nearly maximizing the GPU's shared memory throughput and significantly increasing compute throughput. The decompression throughput is now limited by the GPU's access to shared memory. The SMs are waiting for data to transfer from the L1 cache to registers, as indicated by the even further increased *Stall Short Scoreboard* and *Stall MIO Throttle* warp states. The number of accesses to shared memory can be reduced by using asynchronous memory copies, as they only pass through the L1 cache once, compared to normal loads that pass through the L1 twice.

The use of asynchronous memory copies requires a thread block to access memory in a coalesced pattern. This leads to the development of the *GSST Coalesce Kernel*, which implements coalesced memory access. Unfortunately, this version forces all threads to process the same data simultaneously, resulting in a significant increase in the warp state *Stall Barrier* as threads wait for others to decompress their part of the shared input buffer. Although increasing the input buffer size in shared memory helps reduce this, *Stall Barrier* remains the most common warp state, and causes the decompression throughput decreases. However, enabling asynchronous memory access in the *GSST Coalesce Kernel Async* increases decompression throughput and reduces shared memory usage by 20%. Despite these improvements, this version still does not surpass the decompression throughput of the splits format.

Ultimately, the tuning of algorithm parameters is done on the splits format as the *GSST Splits Kernel Parameter Optimized*, where reducing the size of the shared memory buffers decreases the *Stall Long Scoreboard* and achieves the highest decompression throughput, memory throughput, and compute throughput of all implementations.

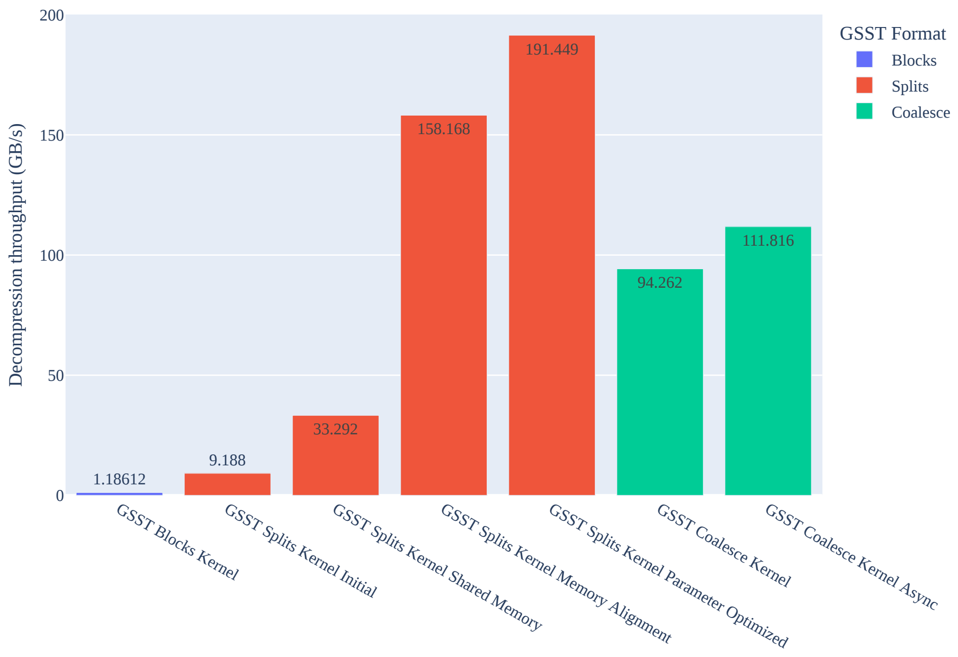


Figure 6.1: Maximum decompression throughput achieved by each GSST implementation.
GSST Blocks Kernel introduces block parallelism, as explained in Section 4.2.1.
GSST Splits Kernel Initial is the first implementation utilizing the split format, described in Section 4.2.2.
GSST Splits Kernel Shared Memory enhances this by incorporating shared memory for data staging, discussed in Section 4.3.1.
GSST Splits Kernel Memory Alignment adds aligned memory transfers, as covered in Section 4.3.2.
GSST Splits Kernel Parameter Optimized is the same as the memory-aligned splits implementation, but with parameters fine-tuned in the experiments from Chapter 5. *GSST Coalesce Kernel* represents the first use of the reordered codes structure, explained in Section 4.2.3.
GSST Coalesce Kernel Async introduces asynchronous memory transfers, as described in Section 4.3.3.

6.2. Text Decompression Throughput

The throughput and compression ratio of GSST is compared against the state-of-the-art nvCOMP compression algorithms. From Sections 2.2.2 and 5.4.4 the best configurations are selected and benchmarked again on the test setup. The compression ratio and the throughput of the decompression is measured.

From the results shown in Figure 6.2, the following conclusions can be drawn:

1. Excellent Decompression Throughput: GSST significantly outperforms traditional CPU-based formats in nvCOMP, such as LZ4, Snappy, and Zstd, when it comes to decompression speed. **GSST achieves a decompression throughput of 190 GB/s, which is only surpassed by ANS and Bitcomp in nvCOMP**. While these two algorithms have a 49% and 46% higher decompression throughput, respectively, GSST offers a 49% and 151% higher compression ratio compared to ANS and Bitcomp.

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 1858% faster than Zstd's.

3. GSST strikes an excellent balance between compression ratio and decompression throughput, improving upon the traditional trade-offs and effectively moving the Pareto frontier.

Overall, GSST is well-positioned as an optimal solution, offering a strong trade-off between compression efficiency and decompression speed.

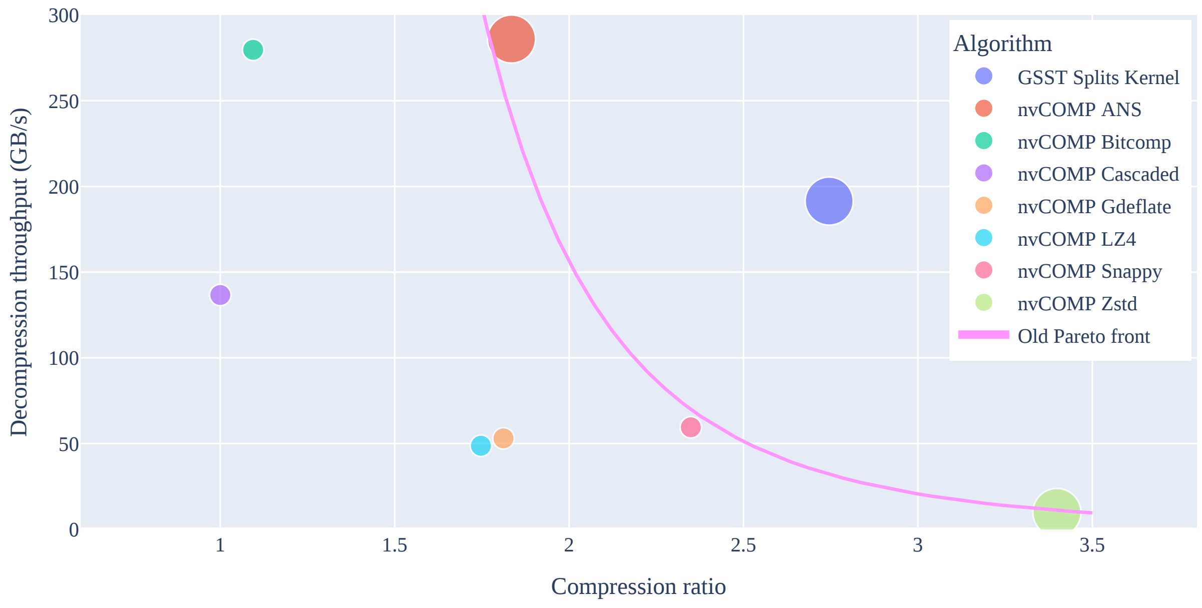


Figure 6.2: Comparison of decompression throughput and compression ratio of GSST against nvCOMP algorithms on an NVIDIA A100, using the TPC-H text dataset. The decompression throughput measures the time it takes for compressed data in GPU memory to be decompressed to GPU memory. GSST shifts the Pareto front, offering an unparalleled balance of both decompression speed and compression efficiency.

6.3. Ingestion Throughput

The impact of GSST on data ingestion is quantified using the model described in Figure 3.2 and equation (3.4). To evaluate performance, the best combination of decompression throughput and compression ratio from Figure 6.2 is used of all algorithms, and are applied to this model. The interconnect throughput of is varied between 0 and 100 GB/s. As shown in Figure 6.3, **GSST achieves the highest ingestion throughput for all compression algorithms for interconnects with a throughput between 800 MB/s and 87 GB/s**. The most significant difference occurs at an interconnect speed of 22 GB/s, where GSST achieves an ingestion throughput of 45 GB/s, compared to 35 GB/s achieved nvCOMP ANS. This represents a 27% increase in ingestion throughput. Compared to the industry standard Snappy, GSST increases ingestion throughput up to 130%.

Below an interconnect throughput of 800 MB/s, Zstd outperforms GSST due to its higher compression ratio, while above 87 GB/s, ANS surpasses GSST because of its faster decompression throughput. However, within the 800 MB/s to 87 GB/s range, GSST provides the best ingestion throughput as a result of its balanced compression ratio and decompression speed.

Given that the fastest current networking equipment can transfer data at up to 50 GB/s, this demonstrates that GSST is an ideal choice for today's data ingestion tasks, with room to accommodate even faster future network equipment [71].

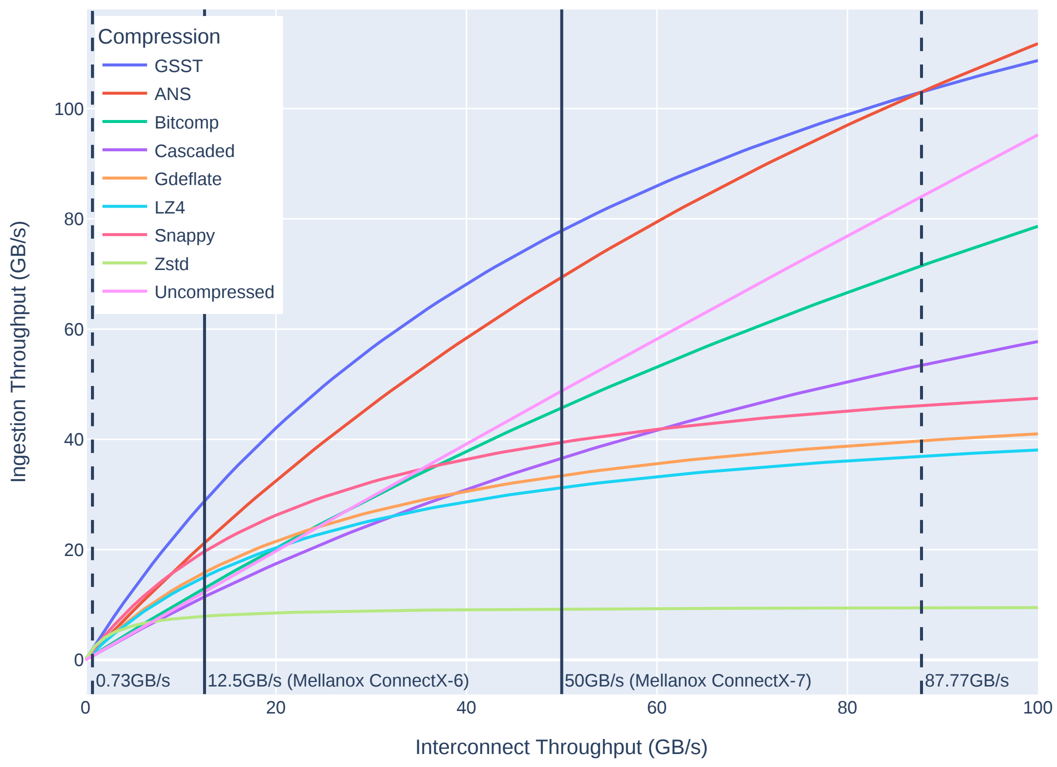


Figure 6.3: Comparison of the ingestion throughput of GSST with the decompression algorithms from nvCOMP. The data combines the highest decompression throughput and compression ratios from Figure 6.2 with the ingestion model outlined in Figure 3.1b. Using Equation (3.4) for ingestion throughput without overlap, the graph shows throughput over a range of interconnect speeds. It highlights that GSST achieves the highest ingestion throughput when the interconnect throughput ranges from 800 MB/s to 87 GB/s.

6.4. GPU Memory Usage

Memory usage is an important factor when evaluating compression algorithms, especially on GPUs, where efficient use of memory is critical for performance. Excessive memory consumption can slow down processing, limit throughput, and prevent large datasets from being handled effectively. 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 a 10 GB text file. Each algorithm was tested with its highest throughput configuration, as found in Section 5.4.4 and shown in Figure 6.2.

For each algorithm, the measurement was executed in the following steps:

1. The compression program is launched using NVIDIA Nsight Systems
2. First the compression algorithm is called to compress the buffer
3. Then the compression manager is deconstructed, and its buffers are deallocated to make sure it does not influence the memory usage measurements of the decompression
4. Finally the decompression is executed

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.

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

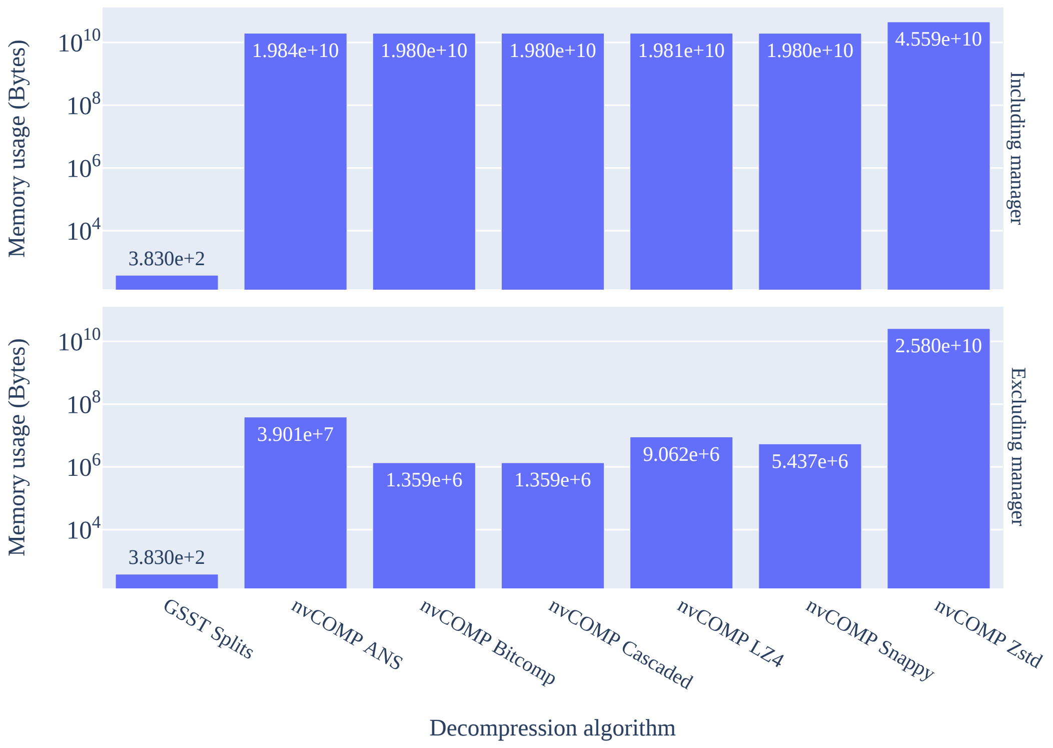


Figure 6.4: Peak GPU memory usage of GSST compared to the algorithms in the nvCOMP library when decompressing the 10 GB TPC-H text dataset on the A100 80 GB, as reported by NVIDIA Nsight Systems. This only includes the memory usage from initializing the compression manager and calling the decompression 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 launch 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.

algorithm, the manager allocated a buffer twice the size of the input data, 20 GB for a 10 GB file, 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's possible that using the low-level API could reduce these large memory allocations. The results are shown in a memory usage chart Figure 6.4.

GSST demonstrates a significant advantage over nvCOMP in terms of memory efficiency. By storing all processing variables in shared memory, the algorithm requires almost no global memory. In fact, **GSST uses less than 3,500 times the memory of nvCOMP's most efficient decompressor (Bitcomp) and over 67 million times less compared to nvCOMP's most memory-intensive compressor (Zstd)**.

As stated before, this is mainly the result of nvCOMP using ridiculous amounts of global memory. As nvCOMP is closed source, it is unclear why it allocates such high amounts of memory. While using the low-level API might help reduce memory consumption, it is unlikely to bring memory usage down to the same levels as GSST.

7 Conclusions and Future Work

7.1. Conclusions

The main research question in this thesis is: **How can compression increase data ingestion throughput in GPU processing?**

The following sub-questions are answered:

1. **What are the bottlenecks in data ingestion on a GPU?**
   The bottlenecks in GPU data ingestion can be modeled in two parts: the limited throughput of data transfers and the limited throughput of decompression algorithms. Traditionally, compression is applied to overcome slow data transfer bottlenecks. However, with new high throughput storage systems, the bottleneck in data ingestion is shifting towards the decompression algorithms in the device memory, rather than data transfer throughput on the interconnect.
2. **Which CPU compression formats have the potential for high performance on GPUs?**
   Heavyweight compression techniques suffer from sequential data dependencies, making them difficult to parallelize. After evaluating several CPU-based compression formats, FSST was identified as a promising candidate for GPU adaptation. FSST achieves high decompression throughput and can be parallelized by overcoming only a single sequential data dependency.
3. **What strategies can be used to effectively distribute data across the parallel processing units in the GPU?**
   The thesis demonstrated that distributing data in smaller blocks and introducing splits within these blocks can significantly increase the parallelism. By adding additional metadata to the compressed data, GSST enables multiple threads to work simultaneously on different sections of the compressed data.
4. **How can memory access patterns be optimized to reduce memory latency during decompression?**
   Optimizing memory access patterns proved to be crucial for maximizing decompression throughput. The data format is changed to achieve coalesced memory access by grouping the data based on the order they are accessed. Three GPU specific memory optimizations are applied: the use of shared memory for frequently used data, the alignment of data transfers, and utilizing asynchronous memory accesses.
5. **What are optimal parameters for the GPU decompression algorithms?**
   Experiments are conducted to find the optimal parameters for GSST compression, derived from the hardware specifications of the GPU used. Based on the number of SMs, and the cache sizes in the GPU, a configuration for GSST is derived that achieves the highest throughput during decompression on the test setup.

In conclusion, GSST increases data ingestion throughput up to 27% compared to the second-best algorithm nvCOMP ANS, and up to 130% compared to industry standard Snappy. By providing a good balance between decompression throughput with compression ratio and by fine-tuning parallelism and memory access, GSST achieved a decompression throughput of 190 GB/s with a compression ratio of 2.74. This makes GSST a highly effective solution for modern and future high-speed GPU data processing applications. The source code of GSST will be made available on GitHub[3].

7.2. Future Work

7.2.1. GSST Implementation Improvements

Coalesced Memory Access Format Performance

Contrary to my expectations, the coalesced format actually performs worse than the splits format. As discussed in Section 6.1 the coalesced format shows an advantage in reducing the load on the shared memory. However, the barriers introduced by this format prevent it from fully utilizing the newly available memory throughput.

If it is possible to better balance the load per thread within a block, or remove the barrier on the input data altogether, the coalesced format has the potential to outperform the splits format in throughput.

FSST12

This thesis is based on the FSST implementation that uses 8-bit codes for indexes in the symbol table, because of the small size (2 KB) of this symbol table. However, FSST also provides an alternative implementation that uses 12-bit codes as indexes, which expands the number of entries from 255 to 4096, resulting in a symbol table size of 32 KB. In some cases, this larger version can achieve better compression ratios. It would be interesting to investigate whether these improvements could also be applied to GSST. However, the increased size of the symbol table could pose challenges when trying to fit it into shared memory.

Asynchronous Memory Access

Currently, CUDA's asynchronous memory access, as described in Section 4.3.3, is limited to transferring data from global memory to shared memory, with the participation of all threads in a thread block. There are two key improvements that could enhance the performance of GSST decompression:

1. **Support for Asynchronous Transfers from Shared to Global Memory**: Decompression algorithms typically write more data to global memory than they read from it. Therefore, enabling asynchronous transfers when moving data from shared memory back to global memory would provide a greater performance boost than the current support for transfers from global to shared memory.

2. **Support for Asynchronous Transfers at the Thread Level**: Allowing asynchronous data transfers at the thread level would enable GSST to take advantage of this functionality when in the splits format. This potentially can boost the throughput by 15%, as it did for the coalesced format.

Multiple Buffering

Multiple buffering, also known as double or triple buffering depending on the number of buffers used, is a technique used to overlap the processing of data with its transfer. This technique involves the use of multiple buffers to hold data that is being processed, allowing one buffer to be filled with new data while another is being processed. This overlapping of data transfer and processing helps to keep the pipeline filled, reducing idle times for the processor and improving overall throughput.

The CUDA extended API includes the *cuda::pipeline* data structure, which can be used to implement double buffering in combination with asynchronous memory access support.

7.2.2. GSST Tuning Improvements

Entropy, Block Size, and Compression Ratio

In this thesis, the entropy of the data wasn't specifically used to optimize the algorithm's parameters, but there's likely a strong connection between entropy, block size, and compression ratio. Entropy refers to the amount of randomness in the data. If the input data has high entropy, meaning it's more random, I believe using a smaller block size could improve the compression ratio to some extent. However, if the data is completely random and incompressible, this could have the opposite effect and make the situation worse.

On the other hand, if the data has low entropy, meaning it's more predictable, a larger block size could be beneficial. In this case, the symbol table can cover more of the data efficiently.

The compressor could calculate the data's entropy (or be provided with this information) and adjust the block size accordingly. This would help improve the compression ratio.

Additionally, another approach could be used to identify hot spots in the data based on the information density. These hot spots can be grouped into clusters, where each cluster contains the same amount of information rather than the same number of bytes. This could enhance both the compression ratio, by reducing data variety in each block, and decompression speed, by balancing the number of operations needed to decompress each block. The current format already supports blocks of varying compressed and uncompressed sizes, so no changes to the file format are needed to implement this.

Other (GPU) Hardware Configurations
In Chapter 5, the default settings for GSST are optimized specifically for the A100 80 GB GPU. These parameters were derived based on the hardware specifications of that GPU. The same approach can be used to find optimal parameters for other GPUs, though it hasn't been tested whether the configuration would be the best for them as well.

Ideally, the compression process should account for the GPU device used for decompression. By knowing the target device, the compression algorithm can adjust the number of blocks and splits to maximize decompression throughput on that particular hardware.

7.2.3. GSST Compression

This thesis focuses on the decompression algorithm of FSST on GPUs. The compression process is currently implemented on the CPU, and is only implemented for the purpose of testing the decompression. However, for GSST to be a complete solution, a GPU-based compression algorithm is needed. Here are some thoughts on how this could be implemented:

Similar to the decompression, compression can be done in parallel on a block-by-block basis. Each SM on the GPU would be responsible for compressing one block of input data independently. However, a key challenge is how to combine the compressed blocks. Because blocks X and X+1 are compressed at the same time and the compressed size of block X is not known in advance, the starting position of the output for block X+1 cannot be determined until block X finishes compressing. This creates a sequential dependency between blocks.

A possible solution is to compress each block into its own temporary output buffer. Once block X has been compressed, the data from block X+1 can be moved to the correct position in the output. However, this method increases memory usage and requires a sequential post-processing step after all blocks are compressed, which is not ideal.

Another challenge is parallelizing the compression process within each block. Each block is compressed into splits, and compression could potentially be parallelized across these splits. However, the same issue arises: the compressed size of split Y is unknown, so it's hard to determine where to place the output for split Y+1 in parallel.

There are other opportunities for parallelism within a block. FSST compression involves two main steps, as described in Section 2.4. First, a symbol table needs to be created from a sample of the input data. This symbol table will be shared by all splits within the block during the compression. To parallelize this step, multiple threads could scan the input data at the same time, each looking for different patterns and building the table cooperatively. This can be efficiently done using shared memory on the GPU.

The second step is encoding the input data by replacing it with symbols from the symbol table. Since the symbol table is in shared memory, each thread can work independently on its split of data, using the table to find the longest matching symbols and replacing the input data accordingly. The challenge again is determining the output location of the compressed data.

Finally, moving data between global memory to shared memory can benefit from the same memory alignment techniques described in Sections 4.3.1 and 4.3.2. This will ensure efficient data transfer and processing during compression.

Bibliography

[1] R. Vonk, J. Hoozemans, and Z. Al-Ars, “GSST: Parallel string decompression at 150 GB/s on GPU,” submitted to CIDR 2025.

[2] J. Hoozemans, R. Vonk, J. Peltenburg, F. Aramburu, and Z. Al-Ars, “Benchmarking GPU direct storage for high-performance filesystems: Impact & future trends,” submitted to Sigmod 2025.

[3] R. Vonk. “Voltrondata/gsst: GPU static symbol table (GSST): GPU implementation of FSST compression,” GitHub. (Sep. 2024), [Online]. Available: `https://github.com/voltrondata/gsst` (visited on 09/13/2024).

[4] D. Reinsel, J. Gantz, and J. Rydning, “The digitization of the world from edge to core,” [Online]. Available: `https://www.seagate.com/files/www-content/our-story/trends/files/dataage-idc-report-final.pdf`.

[5] Voltron Data. “Theseus data processing engine,” Theseus Data Processing Engine. (Aug. 15, 2024), [Online]. Available: `https://voltrondata.com/theseus.html` (visited on 08/15/2024).

[6] M. Abadi, P. Barham, J. Chen, *et al.*, “TensorFlow: A system for large-scale machine learning,”

[7] M. Bedford Taylor, “The evolution of bitcoin hardware,” *Computer*, vol. 50, no. 9, pp. 58–66, 2017, Conference Name: Computer, ISSN: 1558-0814. DOI: `10.1109/MC.2017.3571056`. [Online]. Available: `https://ieeexplore-ieee-org.tudelft.idm.oclc.org/document/8048662` (visited on 08/15/2024).

[8] G. Florimbi, H. Fabelo, E. Torti, *et al.*, “Towards real-time computing of intraoperative hyperspectral imaging for brain cancer detection using multi-GPU platforms,” *IEEE Access*, vol. 8, pp. 8485–8501, 2020, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: `10.1109/ACCESS.2020.2963939`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8949497` (visited on 08/20/2024).

[9] D. Sierra-Sosa, B. Garcia-Zapirain, C. Castillo, *et al.*, “Scalable healthcare assessment for diabetic patients using deep learning on multiple GPUs,” *IEEE Transactions on Industrial Informatics*, vol. 15, no. 10, pp. 5682–5689, Oct. 2019, Conference Name: IEEE Transactions on Industrial Informatics, ISSN: 1941-0050. DOI: `10.1109/TII.2019.2919168`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8723173` (visited on 08/20/2024).

[10] B. Sukhwani and M. C. Herbordt, “GPU acceleration of a production molecular docking code,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2, New York, NY, USA: Association for Computing Machinery, Mar. 8, 2009, pp. 19–27, ISBN: 978-1-60558-517-8. DOI: `10.1145/1513895.1513898`. [Online]. Available: `https://dl.acm.org/doi/10.1145/1513895.1513898` (visited on 08/20/2024).

[11] J. Cowls, A. Tsamados, M. Taddeo, and L. Floridi, “The AI gambit: Leveraging artificial intelligence to combat climate change—opportunities, challenges, and recommendations,” *AI & SOCIETY*, vol. 38, no. 1, pp. 283–307, Feb. 1, 2023, ISSN: 1435-5655. DOI: `10.1007/s00146-021-01294-x`. [Online]. Available: `https://doi.org/10.1007/s00146-021-01294-x` (visited on 08/20/2024).

[12] J. D. C. Maia, G. A. Urquiza Carvalho, C. P. J. Mangueira, S. R. Santana, L. A. F. Cabral, and G. B. Rocha, “GPU linear algebra libraries and GPGPU programming for accelerating MOPAC semiempirical quantum chemistry calculations,” *Journal of Chemical Theory and Computation*, vol. 8, no. 9, pp. 3072–3081, Sep. 11, 2012, Publisher: American Chemical Society, ISSN: 1549-9618. DOI: `10.1021/ct3004645`. [Online]. Available: `https://doi.org/10.1021/ct3004645` (visited on 08/20/2024).

[13] V. Makoviychuk, L. Wawrzyniak, Y. Guo, *et al.*, *Isaac gym: High performance GPU-based physics simulation for robot learning*, Aug. 25, 2021. DOI: `10.48550/arXiv.2108.10470`. arXiv: `2108.10470[cs]`. [Online]. Available: `http://arxiv.org/abs/2108.10470` (visited on 08/20/2024).

[14] N. Ahmed, H. Mushtaq, K. Bertels, and Z. Al-Ars, "GPU accelerated API for alignment of genomics sequencing data," in *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Nov. 2017, pp. 510–515. DOI: `10.1109/BIBM.2017.8217699`. [Online]. Available: `https://ieeexplore.ieee.org/document/8217699` (visited on 08/20/2024).

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008, Conference Name: Proceedings of the IEEE, ISSN: 1558-2256. DOI: `10.1109/JPROC.2008.917757`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/4490127` (visited on 08/20/2024).

[16] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar RAM-CPU cache compression," in *22nd International Conference on Data Engineering (ICDE'06)*, Atlanta, GA, USA: IEEE, 2006, pp. 59–59, ISBN: 978-0-7695-2570-9. DOI: `10.1109/ICDE.2006.150`. [Online]. Available: `http://ieeexplore.ieee.org/document/1617427/` (visited on 09/03/2024).

[17] G. E. Moore, "Cramming more components onto integrated circuits," *PROCEEDINGS OF THE IEEE*, vol. 86, no. 1, 1998.

[18] S. Mittal and J. S. Vetter, "A survey of architectural approaches for data compression in cache and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1524–1536, May 2016, Conference Name: IEEE Transactions on Parallel and Distributed Systems, ISSN: 1558-2183. DOI: `10.1109/TPDS.2015.2435788`. [Online]. Available: `https://ieeexplore.ieee.org/document/7110612` (visited on 09/03/2024).

[19] A. F. A. Furtunato, K. Georgiou, K. Eder, and S. Xavier-De-Souza, "When parallel speedups hit the memory wall," *IEEE Access*, vol. 8, pp. 79 225–79 238, 2020, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: `10.1109/ACCESS.2020.2990418`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/9078685` (visited on 09/03/2024).

[20] A. Maurya, J. Ye, M. M. Rafique, F. Cappello, and B. Nicolae, *Breaking the memory wall: A study of i/o patterns and GPU memory utilization for hybrid CPU-GPU offloaded optimizers*, Jun. 15, 2024. DOI: `10.1145/3659995.3660038`. arXiv: `2406.10728[cs]`. [Online]. Available: `http://arxiv.org/abs/2406.10728` (visited on 08/23/2024).

[21] T. Mladenova, Y. Kalmukov, M. Marinov, and I. Valova, "Impact of data compression on the performance of column-oriented data stores," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 12, no. 7, 2021, Number: 7 Publisher: The Science and Information (SAI) Organization Limited, ISSN: 2156-5570. DOI: `10.14569/IJACSA.2021.0120747`. [Online]. Available: `https://thesai.org/Publications/ViewPaper?Volume=12&Issue=7&Code=IJACSA&SerialNo=47` (visited on 09/03/2024).

[22] D. J. Abadi, "Query execution in column-oriented database systems,"

[23] J. Cao, R. Sen, M. Interlandi, J. Arulraj, and H. Kim, *Revisiting query performance in GPU database systems*, Feb. 1, 2023. arXiv: `2302.00734[cs]`. [Online]. Available: `http://arxiv.org/abs/2302.00734` (visited on 03/04/2024).

[24] *BlazingDB/blazingsql*, original-date: 2018-09-24T18:25:45Z, Sep. 9, 2024. [Online]. Available: `https://github.com/BlazingDB/blazingsql` (visited on 09/17/2024).

[25] NVIDIA Corporation. "GPUDirect storage (GDS)," NVIDIA Docs. (Jul. 31, 2024), [Online]. Available: `https://docs.nvidia.com/gpudirect-storage/index.html` (visited on 08/15/2024).

[26] NVIDIA Corporation, "NVIDIA NVLink high-speed interconnect: Application performance," Nov. 2014. [Online]. Available: `https://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf` (visited on 08/19/2024).

[27] D. Das Sharma, R. Blankenship, and D. Berger, "An introduction to the compute express link (CXL) interconnect," *ACM Comput. Surv.*, vol. 56, no. 11, 290:1–290:37, Jul. 8, 2024, ISSN: 0360-0300. DOI: `10.1145/3669900`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3669900` (visited on 09/04/2024).

[28] J. Stuecheli, W. J. Starke, J. D. Irish, *et al.*, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI," *IBM Journal of Research and Development*, vol. 62, no. 4, 8:1–8:8, Jul. 2018, Conference Name: IBM Journal of Research and Development, ISSN: 0018-8646. DOI: `10.1147/JRD.2018.2856978`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8413085` (visited on 09/04/2024).

[29] V. Rosenfeld, S. Breß, and V. Markl, "Query processing on heterogeneous CPU/GPU systems," *ACM Comput. Surv.*, vol. 55, no. 1, 11:1–11:38, Jan. 17, 2022, ISSN: 0360-0300. DOI: `10.1145/3485126`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3485126` (visited on 09/10/2024).

[30] NVIDIA Corporation. "RAPIDS cuDF accelerates pandas nearly 150x with zero code changes," NVIDIA Technical Blog. (Mar. 18, 2024), [Online]. Available: `https://developer.nvidia.com/blog/rapids-cudf-accelerates-pandas-nearly-150x-with-zero-code-changes/` (visited on 09/04/2024).

[31] D. Vohra, "Apache parquet," in *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, D. Vohra, Ed., Berkeley, CA: Apress, 2016, pp. 325–335, ISBN: 978-1-4842-2199-0. DOI: `10.1007/978-1-4842-2199-0_8`. [Online]. Available: `https://doi.org/10.1007/978-1-4842-2199-0_8` (visited on 09/11/2024).

[32] K. I. Iourcha, K. S. Nayak, and Z. Hong, "System and method for fixed-rate block-based image compression with inferred pixel values," U.S. Patent 5956431A, Sep. 21, 1999.

[33] A. Weißenberger and B. Schmidt, *Accelerating JPEG decompression on GPUs*, Nov. 17, 2021. arXiv: `2111.09219[cs]`. [Online]. Available: `http://arxiv.org/abs/2111.09219` (visited on 09/10/2024).

[34] L. Hasan, M. Kentie, and Z. Al-Ars, "DOPA: GPU-based protein alignment using database and memory access optimizations," *BMC Research Notes*, vol. 4, no. 1, p. 261, Jul. 28, 2011, ISSN: 1756-0500. DOI: `10.1186/1756-0500-4-261`. [Online]. Available: `https://doi.org/10.1186/1756-0500-4-261` (visited on 06/10/2024).

[35] J. Park, Z. Qureshi, V. Mailthody, *et al.*, *CODAG: Characterizing and optimizing decompression algorithms for GPUs*, Jul. 7, 2023. DOI: `10.48550/arXiv.2307.03760`. arXiv: `2307.03760[cs]`. [Online]. Available: `http://arxiv.org/abs/2307.03760` (visited on 12/20/2023).

[36] Madonna University and A. C. Eberendu, "Unstructured data: An overview of the data of big data," *International Journal of Computer Trends and Technology*, vol. 38, no. 1, pp. 46–50, Aug. 25, 2016, ISSN: 22312803. DOI: `10.14445/22312803/IJCTT-V38P109`. [Online]. Available: `http://www.ijcttjournal.org/archives/ijctt-v38p109` (visited on 08/26/2024).

[37] H. Muscolino, A. Machado, J. Rydning, and D. Vesset, "Untapped value: What every executive needs to know about unstructured data," Aug. 2023.

[38] D. Phillips, *Electrum/tpch-dbgen*, original-date: 2012-01-18T19:28:20Z, Aug. 22, 2024. [Online]. Available: `https://github.com/electrum/tpch-dbgen` (visited on 09/04/2024).

[39] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952, Conference Name: Proceedings of the IRE, ISSN: 2162-6634. DOI: `10.1109/JRPROC.1952.273898`. [Online]. Available: `https://ieeexplore.ieee.org/document/4051119` (visited on 09/05/2024).

[40] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977, Conference Name: IEEE Transactions on Information Theory, ISSN: 1557-9654. DOI: `10.1109/TIT.1977.1055714`. [Online]. Available: `https://ieeexplore.ieee.org/document/1055714` (visited on 11/24/2023).

[41] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987, ISSN: 0001-0782, 1557-7317. DOI: `10.1145/214762.214771`. [Online]. Available: `https://dl.acm.org/doi/10.1145/214762.214771` (visited on 02/14/2024).

[42] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Transactions on Communications*, vol. 38, no. 11, pp. 1917–1921, Nov. 1990, Conference Name: IEEE Transactions on Communications, ISSN: 1558-0857. DOI: `10.1109/26.61469`. [Online]. Available: `https://ieeexplore.ieee.org/document/61469` (visited on 09/10/2024).

[43] M. Burrows, D. J. W. D. I. G. I. T. A. L, R. W. Taylor, D. Wheeler, and D. Wheeler, "A block-sorting lossless data compression algorithm," 1994. [Online]. Available: https://www.semanticscholar.org/paper/A-Block-sorting-Lossless-Data-Compression-Algorithm-Burrows-D.J.WheelerDIGITA/af56e6d4901dcd0f589bf969e604663d40f1be5d (visited on 09/10/2024).

[44] M. Mahoney, "Adaptive weighing of context models for lossless data compression," Dec. 21, 2005. [Online]. Available: https://repository.fit.edu/cgi/viewcontent.cgi?article=1164&context=ces_faculty (visited on 09/11/2024).

[45] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for huffman coding," in *2015 Picture Coding Symposium (PCS)*, May 2015, pp. 65–69. DOI: 10.1109/PCS.2015.7170048. [Online]. Available: https://ieeexplore.ieee.org/document/7170048 (visited on 12/04/2023).

[46] M. Mahoney, *Large text compression benchmark*, Jun. 4, 2024. [Online]. Available: https://mattmahoney.net/dc/text.html (visited on 02/09/2024).

[47] P. Skibinski, *Inikep/lzbench*, original-date: 2015-10-31T14:41:09Z, Sep. 12, 2024. [Online]. Available: https://github.com/inikep/lzbench (visited on 09/13/2024).

[48] NVIDIA Corporation. "nvCOMP," NVIDIA nvCOMP. (2024), [Online]. Available: https://docs.nvidia.com/cuda/nvcomp/index.html (visited on 07/24/2024).

[49] *Facebookresearch/dietgpu*, original-date: 2022-01-18T19:54:23Z, Sep. 11, 2024. [Online]. Available: https://github.com/facebookresearch/dietgpu (visited on 09/13/2024).

[50] F. Knorr, P. Thoman, and T. Fahringer, "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*, ser. SC '21, New York, NY, USA: Association for Computing Machinery, Nov. 13, 2021, pp. 1–14, ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3476224. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3476224 (visited on 12/08/2023).

[51] B. Zhang, J. Tian, S. Di, *et al.*, "GPULZ: Optimizing LZSS lossless compression for multi-byte data on modern GPUs," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23, New York, NY, USA: Association for Computing Machinery, Jun. 21, 2023, pp. 348–359. DOI: 10.1145/3577193.3593706. [Online]. Available: https://dl.acm.org/doi/10.1145/3577193.3593706 (visited on 03/15/2024).

[52] F. Lin, K. Arunruangsirilert, H. Sun, and J. Katto, "Recoil: Parallel rANS decoding with decoder-adaptive scalability," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23, New York, NY, USA: Association for Computing Machinery, Sep. 13, 2023, pp. 31–40. DOI: 10.1145/3605573.3605588. [Online]. Available: https://dl.acm.org/doi/10.1145/3605573.3605588 (visited on 12/08/2023).

[53] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a GPU," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4, New York, NY, USA: Association for Computing Machinery, Mar. 5, 2011, pp. 1–7, ISBN: 978-1-4503-0569-3. DOI: 10.1145/1964179.1964189. [Online]. Available: https://dl.acm.org/doi/10.1145/1964179.1964189 (visited on 04/15/2024).

[54] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *2016 45th International Conference on Parallel Processing (ICPP)*, ISSN: 2332-5690, Aug. 2016, pp. 242–247. DOI: 10.1109/ICPP.2016.35. [Online]. Available: https://ieeexplore.ieee.org/document/7573824 (visited on 01/08/2024).

[55] A. Weißenberger and B. Schmidt, "Massively parallel ANS decoding on GPUs," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19, New York, NY, USA: Association for Computing Machinery, Aug. 5, 2019, pp. 1–10, ISBN: 978-1-4503-6295-5. DOI: 10.1145/3337821.3337888. [Online]. Available: https://dl.acm.org/doi/10.1145/3337821.3337888 (visited on 12/05/2023).

[56] A. Weißenberger and B. Schmidt, "Massively parallel inverse block-sorting transforms for bzip2 decompression on GPUs," in *Proceedings of the 53rd International Conference on Parallel Processing*, Gotland Sweden: ACM, Aug. 12, 2024, pp. 856–865. DOI: 10.1145/3673038.3673067. [Online]. Available: https://dl.acm.org/doi/10.1145/3673038.3673067 (visited on 08/12/2024).

[57] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden, “Tile-based lightweight integer compression in GPU,” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD ’22, New York, NY, USA: Association for Computing Machinery, Jun. 11, 2022, pp. 1390–1403, ISBN: 978-1-4503-9249-5. DOI: `10.1145/3514221.3526132`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3514221.3526132` (visited on 01/17/2024).

[58] J. Duda, *Asymmetric numeral systems: Entropy coding combining speed of huffman coding with compression rate of arithmetic coding*, Jan. 6, 2014. DOI: `10.48550/arXiv.1311.2540`. arXiv: `1311.2540[cs,math]`. [Online]. Available: `http://arxiv.org/abs/1311.2540` (visited on 12/01/2023).

[59] P. Boncz, T. Neumann, and V. Leis, “FSST: Fast random access string compression,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2649–2661, Jul. 1, 2020, ISSN: 2150-8097. DOI: `10.14778/3407790.3407851`. [Online]. Available: `https://dl.acm.org/doi/10.14778/3407790.3407851` (visited on 12/07/2023).

[60] NVIDIA Corporation, “NVIDIA a100 tensor core GPU architecture,” NVIDIA Corporation, 2020, p. 82. [Online]. Available: `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`.

[61] NVIDIA Corporation. “GPUDirect storage: A direct path between storage and GPU memory,” NVIDIA Technical Blog. (Aug. 6, 2019), [Online]. Available: `https://developer.nvidia.com/blog/gpudirect-storage/` (visited on 01/24/2024).

[62] Y. Collet and M. Kucherawy, “Zstandard compression and the ’application/zstd’ media type,” Internet Engineering Task Force, Request for Comments RFC 8878, Feb. 2021, Num Pages: 45. DOI: `10.17487/RFC8878`. [Online]. Available: `https://datatracker.ietf.org/doc/rfc8878` (visited on 09/02/2024).

[63] Y. Collet, *Lz4/lz4*, original-date: 2014-03-25T15:52:21Z, Sep. 4, 2024. [Online]. Available: `https://github.com/lz4/lz4` (visited on 09/04/2024).

[64] J. Alakuijala and Z. Szabadka, “Brotli compressed data format,” Internet Engineering Task Force, Request for Comments RFC 7932, Jul. 2016, Num Pages: 128. DOI: `10.17487/RFC7932`. [Online]. Available: `https://datatracker.ietf.org/doc/rfc7932` (visited on 09/02/2024).

[65] L. Noordsij, S. v. d. Vlugt, M. A. Bamakhrama, Z. Al-Ars, and P. Lindstrom, “Parallelization of variable rate decompression through metadata,” in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, ISSN: 2377-5750, Mar. 2020, pp. 245–252. DOI: `10.1109/PDP50117.2020.00045`. [Online]. Available: `https://ieeexplore.ieee.org/document/9092414` (visited on 04/29/2024).

[66] NVIDIA Corporation. “CUDA c++ programming guide.” (2024), [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visited on 02/09/2024).

[67] A. Afroozeh and P. Boncz, “The FastLanes compression layout: Decoding > 100 billion integers per second with scalar code,” *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2132–2144, May 1, 2023, ISSN: 2150-8097. DOI: `10.14778/3598581.3598587`. [Online]. Available: `https://dl.acm.org/doi/10.14778/3598581.3598587` (visited on 11/22/2023).

[68] NVIDIA Corporation. “Controlling data movement to boost performance on the NVIDIA ampere architecture,” NVIDIA Technical Blog. (Sep. 23, 2020), [Online]. Available: `https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/` (visited on 08/27/2024).

[69] G. Thomas-Collignon and V. Mehta, “Optimizing CUDA applications for NVIDIA a100 GPU,” May 21, 2020. [Online]. Available: `https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf`.

[70] H. H. B. Sørensen, “Auto-tuning of level 1 and level 2 BLAS for GPUs,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1183–1198, 2013, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ ISSN: 1532-0634. DOI: `10.1002/cpe.2916`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.2916` (visited on 08/29/2024).

[71] NVIDIA Corporation, "NVIDIA CONNECTX-7 400g ethernet: Smart accelleration for cloud, datacenter and edge," Apr. 2021. [Online]. Available: `https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf` (visited on 09/12/2024).

A
Ingestion Throughput Derivation

$$t_{\textit{ingestion Figure 3.2b}} = t_{\textit{io total}} + t_{\textit{d block N}} \tag{A.1}$$

$$t_{\textit{ingestion Figure 3.2c}} = t_{\textit{d total}} + t_{\textit{io block 1}} \tag{A.2}$$

$$t_{ingestion} = \max(t_{\textit{ingestion Figure 3.2b}}, t_{\textit{ingestion Figure 3.2c}}) \tag{A.3}$$

$$\begin{aligned}
T_{\textit{ingestion Figure 3.2b}} &= \frac{S_u}{t_{\textit{ingestion Figure 3.2b}}} \\
&= \frac{S_u}{t_{\textit{io total}} + t_{\textit{d block N}}} \\
&= \frac{S_u}{\frac{S_c}{T_{io}} + \frac{S_u/N_{blocks}}{T_{dout}}} \\
&= \frac{S_u}{\frac{S_u/CR}{T_{io}} + \frac{S_u}{N_{blocks} \cdot T_{dout}}} \\
&= \frac{S_u}{\frac{S_u}{CR \cdot T_{io}} + \frac{S_u}{N_{blocks} \cdot T_{dout}}} \\
&= \frac{1}{\frac{1}{CR \cdot T_{io}} + \frac{1}{N_{blocks} \cdot T_{dout}}} \\
&= \frac{1}{\frac{N_{blocks} \cdot T_{dout}}{CR \cdot T_{io} \cdot N_{blocks} \cdot T_{dout}} + \frac{CR \cdot T_{io}}{CR \cdot T_{io} \cdot N_{blocks} \cdot T_{dout}}} \\
&= \frac{1}{\frac{N_{blocks} \cdot T_{dout} + CR \cdot T_{io}}{CR \cdot T_{io} \cdot N_{blocks} \cdot T_{dout}}} \\
&= \frac{CR \cdot T_{io} \cdot N_{blocks} \cdot T_{dout}}{N_{blocks} \cdot T_{dout} + CR \cdot T_{io}}
\end{aligned} \tag{A.4}$$

$$
\begin{aligned}
T_{\textit{ingestion Figure 3.2c}} &= \frac{S_u}{t_{\textit{ingestion Figure 3.2c}}} \\
&= \frac{S_u}{t_{\textit{d total}} + t_{\textit{io block 1}}} \\
&= \frac{S_u}{\dfrac{S_u}{T_{dout}} + \dfrac{S_c/N_{blocks}}{T_{io}}} \\
&= \frac{S_u}{\dfrac{S_u}{T_{dout}} + \dfrac{S_c}{N_{blocks} \cdot T_{io}}} \\
&= \frac{S_u}{\dfrac{S_u}{T_{dout}} + \dfrac{S_u/CR}{N_{blocks} \cdot T_{io}}} \\
&= \frac{S_u}{\dfrac{S_u}{T_{dout}} + \dfrac{S_u}{CR \cdot N_{blocks} \cdot T_{io}}} \\
&= \frac{1}{\dfrac{1}{T_{dout}} + \dfrac{1}{CR \cdot N_{blocks} \cdot T_{io}}} \\
&= \frac{1}{\dfrac{CR \cdot N_{blocks} \cdot T_{io}}{T_{dout} \cdot CR \cdot N_{blocks} \cdot T_{io}} + \dfrac{T_{dout}}{T_{dout} \cdot CR \cdot N_{blocks} \cdot T_{io}}} \\
&= \frac{1}{\dfrac{CR \cdot N_{blocks} \cdot T_{io} + T_{dout}}{T_{dout} \cdot CR \cdot N_{blocks} \cdot T_{io}}} \\
&= \frac{T_{dout} \cdot CR \cdot N_{blocks} \cdot T_{io}}{CR \cdot N_{blocks} \cdot T_{io} + T_{dout}}
\end{aligned}
\tag{A.5}
$$

B

CPU Block Parallelism Decompression Throughput

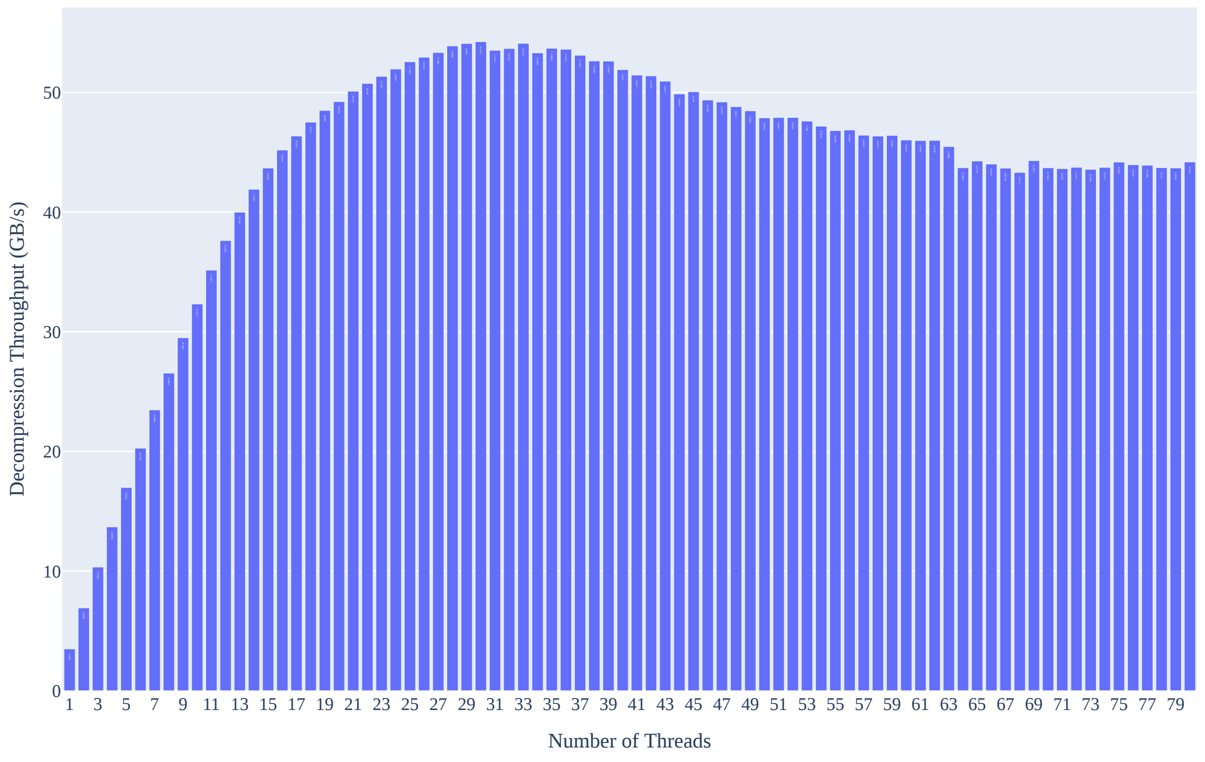


Figure B.1: Graph with the decompression throughput of the CPU implementation using GSST block-level parallelism. The benchmark involves decompressing a 1.5 GB text file, generated by dbgen, which has been compressed into 1024 blocks. The benchmark measures decompression speed of decompressing data from an in-memory input buffer with compressed data to an in-memory output buffer. The decompression throughput steadily increases by 3 GB/s as the number of threads rises up to 14. However, beyond this point, the performance per thread gradually declines until 32 threads, after which adding more threads actually leads to a decrease in throughput. The benchmark was conducted on the setup described in Table 5.1.