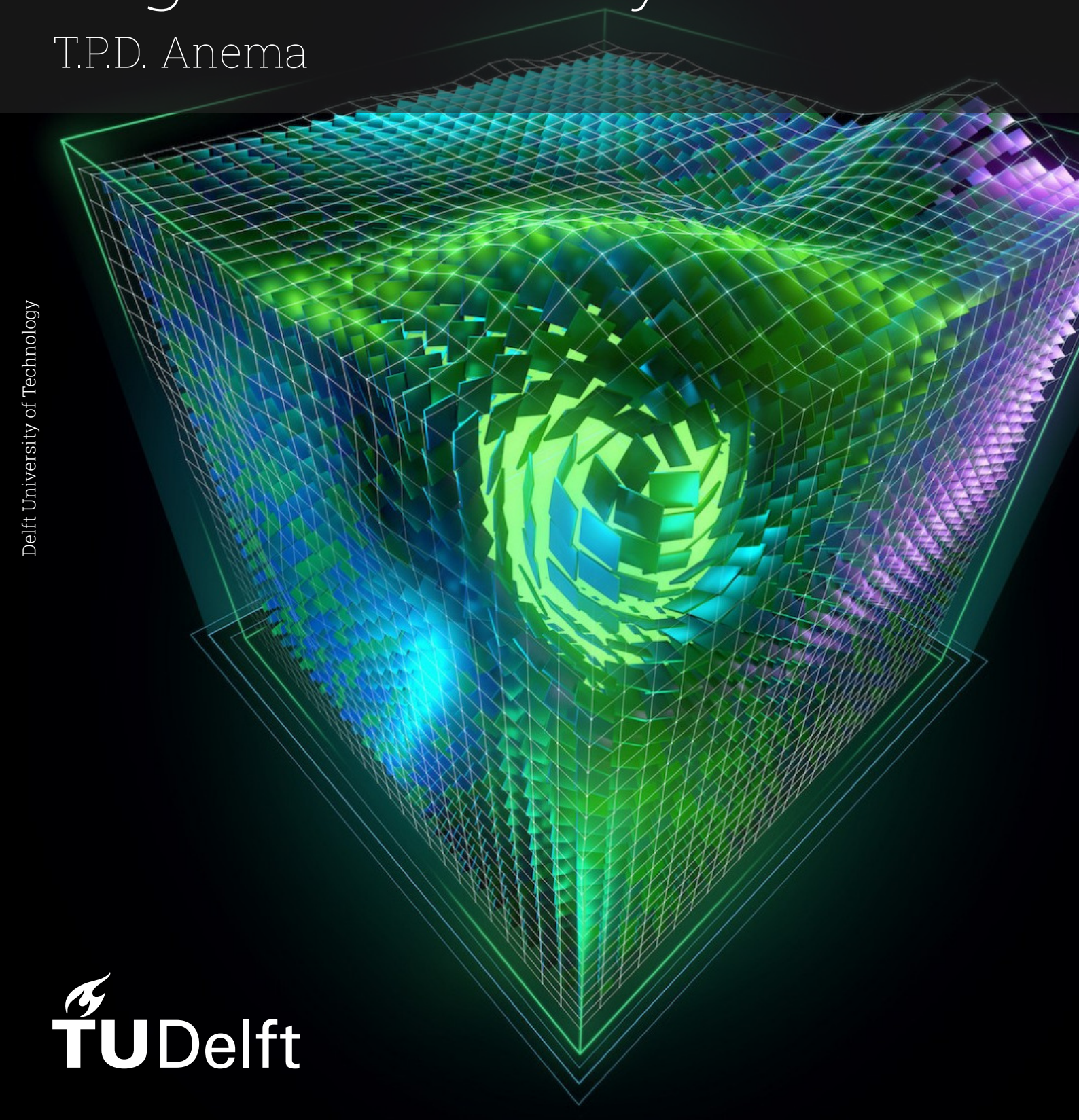
GPU-Accelerated String Compression for Big Data Analytics

T.P.D. Anema



Delft University of Technology

TU Delft

GPU-Accelerated String Compression for Big Data Analytics

by

T.P.D. Anema

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday July 4, 2025 at 10:00 AM.

Student number: 4953940
Project duration: November 25, 2024 – July 4, 2025
Thesis committee:

Prof. dr. H. P. Hofstee,	TU Delft, advisor
Dr. ir. Z. Al-Ars,	TU Delft, supervisor
Dr. ir. M. Möller,	TU Delft
Dr. ir. J. Hoozeman,	Voltron Data

Cover: ”CUDA cube” by NVIDIA

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

This thesis presents a GPU-accelerated string compression algorithm based on FSST (*Fast Static Symbol Table*). The proposed compressor leverages several advanced CUDA techniques to optimize performance, including a voting mechanism that maximizes memory bandwidth and an efficient gathering pipeline utilizing stream compaction. Additionally, the algorithm uses GPU compute capacity to support a memory-efficient encoding table through a space-time tradeoff.

The compression task is parallelized by tiling input data and adapting the data layout. We introduce multiple compression pipelines, each with distinct tradeoffs. To maximize encoding kernel throughput, the design introduces sliding windows and output packing to optimize register use and maximize effective memory bandwidth. Pipeline-level throughput is further enhanced by introducing pipelined transposition stages and stream compaction to remove intermediate padding efficiently.

We evaluate these pipelines across several benchmark datasets and compare the best-performing version against state-of-the-art GPU compression algorithms, including nvCOMP, GPULZ, and compressors generated using the LC framework. The proposed compressor achieves a throughput of 74GB/s on an RTX4090 while maintaining compression ratios comparable to FSST. In terms of compression ratio, it consistently outperforms ANS, Bitcomp, Cascaded, and GPULZ across all datasets. Its overall throughput exceeds that of GPULZ and all nvCOMP compressors except ANS, Bitcomp, Cascaded, and those produced by the LC framework. Our compressor lies on the Pareto frontier for all evaluated datasets, advancing the state-of-the-art toward ideal compression. It achieves near-identical compression ratios to FSST (except for machine-readable datasets), while achieving a speedup of 42.06x. Compared to multithreaded CPU compression, it achieves a 6.45x speedup.

To assess end-to-end performance, we integrate the compressor with the GSST decompressor. The resulting (de)compression pipeline achieves a combined throughput of 55GB/s, outperforming uncompressed data transfer on links with a bandwidth up to 37.5 GB/s. It also outperforms all state-of-the-art (de)compressors when the link bandwidth ranges between 3GB/s and 20GB/s.

While further research is needed to enhance robustness and integrate the compressor into analytical engines, this work demonstrates a viable and Pareto-optimal alternative to existing string compression methods.

The source code of all our compression pipelines is publicly available on GitHub [6]. This work also serves as the foundation for a scientific paper that has been accepted for presentation at ADMS 2025.

To my father, who never saw this adventure

Contents

Acknowledgements

This thesis marks the end of my academic journey at TU Delft. Over the years, I have expanded my knowledge in the field of Computer Science and Engineering and met great people along the way. My time with Formula Student Team Delft introduced me to the world of electronics and electric motor control, and it was there that I discovered how much I enjoy combining software with hardware in hands-on projects. It's a passion I hope to carry forward into my professional career.

The last eight months were dedicated to working on the thesis you are about to read. It was an adventure in itself, as I got to work with some experienced researchers on a very interesting topic that I only briefly touched upon in one of their courses during my master's. I would like to express my gratitude to everyone who helped me in my process, but I would like to explicitly thank some of them.

First, I would like to thank Peter Hofstee for his insights and weekly meetings. Your feedback was always valuable and often challenged me to view the work from new perspectives. I firmly believe his encouragement to publish my work has greatly improved the overall quality of the work.

I would also like to thank my supervisor, Zaid Al-Ars, for his role in the thesis and his feedback. Your eye for detail has helped me improve the quality of the report more than once.

A special thanks as well to Joost Hoozemans. We discussed many technical topics related to CUDA and big data analytics, which helped me to grasp some ideas and better understand the context of this work.

Finally, I would like to thank Lisa for her constant support. You helped me stay grounded when things became overwhelming, helped me think clearly, and always listened to my ideas, even when they drifted well outside your aerospace expertise. You are the best rubber duck I could have wished for.

T.P.D. Anema
Delft, June 2025

List of Figures

List of Tables

1 Introduction

1.1. Context

Over the past decades, there has been a dramatic increase in the amount of data produced and consumed globally. In the mid-1990s, individual data usage was relatively limited, mostly involving text-based emails and basic web browsing, with very low bandwidth requirements [44]. By 2025, however, the average daily data usage per person has risen significantly, mainly driven by streaming services, cloud-based applications, and the widespread use of smartphones [20, 92]. When looking at the future, data generation is only expected to grow more, driven by technology such as 5G networks, AI, and IoT. By 2030, global data production is predicted to exceed several hundred zettabytes annually, reflecting the increasingly digital nature of everyday life and work [56, 93].

Often called big data [52], this extensive and diverse data collection serves a valuable purpose: we use it in predictive modeling, machine learning, and other advanced analytics. Or, to name some more concrete examples, we can use big data to detect fraud [42, 112], accelerate the development of new drugs by analyzing clinical trial data [31, 8], and power recommendations on platforms like Netflix and Spotify [54, 47].

Query engines are essential to process and execute database queries efficiently [80]. With the significant growth in database sizes and complexity, traditional query engines have faced limitations in scalability and performance [52, 48, 7]. Consequently, this has led to the development of specialized systems known as big data query engines. These modern query engines are specifically optimized for handling massive, distributed datasets across clusters of computers, allowing for faster processing and enhanced scalability [110, 127].

However, even if the systems themselves are optimized, the movement of the data itself can become the bottleneck [55, 63]. Compression techniques can reduce the amount of data to be transferred at the cost of compression computational time.

An interesting active research in accelerated big data systems is the acceleration of compression itself [82, 97, 59, 4, 87, 27, 13, 96, 81, 131, 114, 69]. This can lead to higher overall throughput and allow accelerators to use more of their available on-chip memory. This thesis investigates the acceleration potential of data compression, keeping the context of big data query engines in mind.

1.1.1. Compression

In essence, compression is the process of reducing the overall data size while capturing the original data. In the context of compression, there are usually two key metrics: the compression ratio and the compression throughput. The ratio is simply a factor of the original size and resulting size; if we can encode every two bytes in the original data with a single byte in the compressed data, we would have a compression ratio of two. The compression throughput is a performance metric that represents the speed of compression.

What constitutes the original data can be one of two things. If it is possible to bit-identically retrieve the original data, i.e., a 100 percent match of every bit of data, we are dealing with *lossless data compression*. Here, data is often compressed by cleverly using repetition or patterns in the original data.

The alternative is *lossy data compression*, where the compressed data should only be sufficiently similar in the eye of the beholder, usually a person. This means losing a certain amount of information is acceptable if the resulting data resembles the original data. This naturally fits with multimedia, where information can be dropped without significantly impacting the resulting media. This type of compression usually results in a better compression ratio at the cost of losing some information.

1.1.2. Query engines at scale

Classic database systems are incredibly flexible. They usually operate at the row level and in a transactional fashion. This means they are robust to failures and provide consistency, essential properties for almost any (enterprise) application [80]. However, these properties are no longer relevant when we shift our focus to data analysis; a typical workload will query an entire column, not a specific row. This mismatch limits their efficiency and scalability for large-scale analytical tasks [14, 1].

We can see this shift in data format when looking at the current generation of big data query engines like Apache Spark SQL [7], Google BigQuery [62], and Amazon Athena. Their approach to analytical workloads at scale is similar: a (scalable) distributed system focusing on high-performance columnar queries. Typically, these engines leverage substantial memory capacities to reduce I/O overhead by loading relevant data segments into memory, with optimizations aimed at minimizing network data transfers (or *shuffles*), a common bottleneck in distributed computing [7, 62].

More recently, *Graphics Processing Units* (GPUs) have been integrated into analytical engines due to their inherent strengths in parallel processing and high computational throughput. These accelerators are well-suited to analytical workloads due to their natural strength in parallel processing and massive computational power for compute-intensive queries [16, 32, 31, 43, 95, 125]. A significant challenge is efficient data ingestion, as memory throughput limits the theoretical maximum throughput of a GPU [34, 16]. Furthermore, to fully utilize a GPU's computational power, the algorithm has to be (re)designed from the ground up. Examples of this new generation of query engines are cuDF [63], Theseus [23], and HeavyDB [40].

1.1.3. GPU acceleration

Originally, GPUs were specialized electronic circuits designed for digital image processing and basic raster graphics in the 1970s, still as additional (micro)processors on the main board. In the 1990s, we saw the rise of dedicated GPUs with the release of the Voodoo Graphics cards and the start of the NVIDIA GeForce line with an engine for real-time 3D graphics, offloading these tasks from the CPU entirely.

Only in the early 2000s did a shift occur to High-Performance computing, with NVIDIA and ATI (AMD) supporting general-purpose computing tasks beyond graphics. This allowed them to be used for scientific computing due to their ability to perform massively parallel computations. This trend continued in the 2010s with the release of the Fermi and Pascal architectures, marking a turning point for data science and AI.

In modern times, GPUs are an integral part of AI and Big Data computation, with the introduction of the Ampere, Hopper, and the latest Blackwell architectures. They are the foundation for platforms like BlazingSQL and Theseus [23] and the primary processor for machine learning models. Recent innovations like GPU Direct Storage [109], NVLink [77], and the Grace Hopper Superchip further reduce the load on the strained link between GPU and CPU.

However, the usage of GPUs for general-purpose computing (GPGPUs) is not limited to the domain of AI and big data. Several problems in the areas of finance [29], medical computing [39, 5, 99, 102], image processing [129, 65, 115, 90], and many others in different areas have been accelerated using GPGPUs.

1.1.4. Compression in query engines

One aspect is shared among all query engines: keeping as much data as possible in system memory will enhance throughput significantly. This can be solved for CPU-based engines by adding more memory to the system, as the current generation of enterprise-grade server processors can handle 2TB to 4.5TB per socket.

Unfortunately, this is not true for big data query engines like Theseus, which use a GPU as the primary processing device. The current state-of-the-art GPU (B200) only supports up to 192GB of on-device *High-bandwidth memory* (HBM), which means only a relatively small workload can fit in memory

and fully leverage the GPU.

Usually, there are two distinct approaches [125]. The first option is to transfer data to the GPU through the PCIe bus when data is required for a query [55, 122, 126]. Another option is to use the CPU and GPU both for heterogeneous query processing, only using the CPU for certain subqueries or infrequently used data [15, 19].

A significant downside of using GPUs for analytics acceleration is the considerable amount of data transferred over PCIe, which can become the main bottleneck [95, 18]. Although new technologies like NVLink [77] will likely provide higher interconnect bandwidth, they cannot compete with the massive memory bandwidth of modern GPUs. To emphasize this, the new B200 uses the latest HBM3e memory with a bandwidth of 1TB/s per stack, totaling 8.2TB/s overall memory bandwidth [74]. Meanwhile, the latest versions of PCIe and NVLink provide a theoretical 242GB/s and 1.8TB/s, respectively [77].

One technique to overcome this problem is compression. In the case of data ingestion, compression can provide an overall speedup when the decompression time is less than the time gained by transferring less data over an interconnect. For that reason, *de*compression is most important in the context of data analytics. For example, NVIDIA introduced the *Decompression Engine* with Blackwell, which is reported to achieve decompression speeds of 180 GB/s for Snappy on a B200 [71, 74]. In addition, other GPU decompressors have been proposed [114, 57, 100].

When considering big data query engines, there are also interesting gains to be found for compression. GPU memory is a scarce and expensive resource, creating a necessity to temporarily offload memory to host memory (or fast storage, for example, using GDS [109]). Another use case is distributing (shuffling) data between GPUs on a multi-device system or to other nodes in a cluster.

The exact performance requirements of compression depend on the use case. For example, offloading data to fast storage may prioritize a higher compression ratio to reduce storage and transmission volume, while inter-GPU communication (such as during shuffles) may benefit more from higher throughput to minimize latency.

Determining whether the throughput constraint applies to compression, decompression, or both is also important. This distinction is crucial in how performance metrics translate into overall system efficiency.

To illustrate this, consider a scenario where data is temporarily moved to fast storage. In this case, the data must be compressed, transmitted, and later retrieved and decompressed on the GPU. Here, the performance gain of compression must outweigh the combined cost of compressing and decompressing the data. Given the compression ratio CR, compression throughput TH_C, decompression throughput TH_D, and link bandwidth TH_b, the following inequality must hold for a file of size x: $\frac{2x}{TH_b} \geq \frac{x}{CR} \cdot \frac{2}{TH_b} + \frac{x}{TH_C} + \frac{x}{TH_D}$.

Another use case is transferring data to storage, where a different application processes it later. In this case, only compression and transmission are relevant, and the inequality simplifies to $\frac{x}{TH_b} \geq \frac{x}{CR} \cdot \frac{1}{TH_b} + \frac{x}{TH_C}$. We will use this equation in Section 2.1 to determine the effective throughput for a compression algorithm under these conditions.

A third scenario involves real-time data transmission, such as during a shuffle operation. In this case, the data is compressed and transmitted, and another GPU immediately decompresses the data. The minimum required performance of the compression and decompression pipeline is determined by the inequality $\frac{x}{TH_b} \geq \frac{x}{CR} \cdot \frac{1}{TH_b} + \frac{x}{TH_C} + \frac{x}{TH_D}$. In other words, the time required to transmit uncompressed data must exceed the time needed to compress, transmit, and decompress the data. These equations provide a simplified framework for evaluating the suitability of a compression pipeline in different deployment scenarios.

1.2. Research questions and scope of the thesis

In the section above, we discussed the basics of compression and how *lossless* compression can help overcome the communication link bottleneck in big data query engines. We have mainly focused on the PCIe link between the host and GPU, but a similar thing is happening in a more traditional sense with regular network links and shuffles.

This brings us to the primary research goal of this thesis. We aim to research the acceleration potential of existing string compression schemes and determine which scheme could be extended to a GPU-accelerated version. We then design and implement this accelerated compressor. Finally, we would like to investigate if this compressor is a feasible candidate to offload GPU memory, as currently,

there are no good candidates that also provide a high compression ratio. String compression is the most general form of compression, and textual data makes up a significant part of data in analytical systems, which is why we focus on string compression.

We can formulate the following research questions:

- How can we GPU-accelerate an existing string compression scheme for use cases such as the described query engine memory offloading?
- Can we implement this scheme fully on-chip to use it for memory offloading? Would a heterogeneous solution be sufficient or preferred?
- Can we integrate our compressor with an existing GPU-accelerated decompressor for optimal performance?
- How can we tune our parameters to achieve optimal key performance metrics such as compression ratio, throughput, and memory usage?
- What speedup can we get from this GPU-accelerated compression scheme?

To answer these questions, we will first investigate the current state-of-the-art in the realm of compression schemes and compare current CPU- and GPU-based solutions. After identifying a suitable acceleration candidate, we will deeply profile and analyze the algorithm and identify potential acceleration kernels. We will then attempt to port the remaining parts of the algorithm to the GPU to get a scheme that can run entirely on the GPU, and integrate it with an existing decompressor. Finally, we will profile and benchmark the accelerated scheme to achieve optimal key performance metrics.

1.3. Thesis organization

This thesis is organized as follows. We start with Chapter 2, where we introduce some general background about compression, related work, and GPU development. In this chapter, we also identify the algorithm we aim to accelerate.

We then continue with Chapter 3, where we analyze the selected algorithm and identify which components are most suitable for acceleration, as well as how they can be optimized. We identify some challenges and propose mitigations. Furthermore, we show our accelerated pipeline design and several iterations and improvements. We will extend our compression pipeline by integrating it with an existing decompressor in Chapter 4.

In Chapter 5, we analyze the performance of our proposed pipeline and the performance of individual components, in terms of compression ratio and throughput. We compare it to the state-of-the-art and draw some conclusions from these results. We also compare secondary performance indicators like memory usage and energy consumption.

Finally, we conclude the thesis in Chapter 6. We provide an overview of this thesis and summarize our contributions. We will also answer our research questions and discuss some possible future work.

This work serves as the foundation for a paper accepted to ADMS 2025. For completeness, the submitted version is included in Appendix A, as the camera-ready version was not yet available at the time of writing.

2

Background

2.1. Compression

Compression reduces the data size while retaining the original information with a certain margin of information loss. We refer to *lossless* compression when information loss is not allowed. Lossless compression is a reversible process, which means that for every compression algorithm, another algorithm exists that can reconstruct the original data based on the compressed data. This is called decompression.

A compression scheme compresses a data buffer of length L_u (in bytes) to a (compressed) buffer of length L_c in T_c seconds, and a decompression scheme decompresses the compressed buffer back to its original size in T_d seconds. This fundamental workflow of any compression scheme can be seen in Figure 2.1.

Based on this fundamental flow of data, we can already establish some key performance metrics that hold for any compression scheme. First, the relation between the length of the original data and that of the compressed data is called the compression ratio. This metric shows how *effective* a compression algorithm is in reducing the length of data. The compression ratio CR is defined in Equation 2.1.

$$CR = \frac{L_u}{L_c} \tag{2.1}$$

Another important metric is the algorithm's throughput, which tells us something about how *performant* a compression algorithm is. The throughput is the speed at which data is (de)compressed; hence, it is defined by the amount of data being processed divided by the time it took to process the data. In theory, two throughputs can be defined, one considering the length of the uncompressed data L_u and the other considering the length of the compressed data L_c. In practice, we use the former definition that uses uncompressed length to keep the compression ratio CR out of the equation. The compression throughput and decompression throughput equations can be seen in Equation 2.2 and Equation 2.3, respectively.

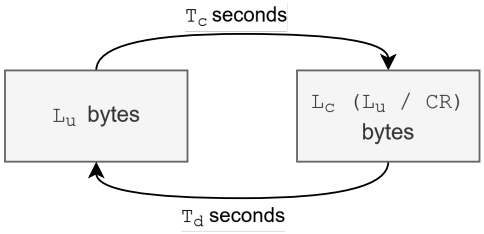


Figure 2.1: The compression cycle consists of a block of data with a size of L_u bytes that is compressed in T_c seconds to a size of L_c bytes, which corresponds to $\frac{L_u}{CR}$ where CR is the compression ratio. It can then be decompressed in T_d seconds to the original data.

$$TH_c = \frac{L_u}{T_c} \tag{2.2}$$

$$TH_d = \frac{L_u}{T_d} \tag{2.3}$$

When comparing different compression schemes, it is essential to consider both performance metrics. Both metrics influence the overall transfer time and the effective throughput. Therefore, without considering both metrics, one scheme cannot be objectively deemed more suited for transferring data over a link than another. Equation 2.4 shows how the total transfer time T_{transfer} of L_U bytes over a link with a bandwidth of TH_B for a compression algorithm can be calculated. Note that T_t is used as the time it takes to transfer the compressed data. This equation shows how both compression ratio and throughput influence the total transfer time.

A more useful metric would be the *effective throughput*, i.e., the throughput of the process that compresses data before transmitting it. This is more useful because throughput does not depend on the length of the data stream, which allows us to compare compression algorithms for a range of link bandwidths directly. Equation 2.5 shows how the effective throughput TH_{transfer} follows from the transfer time.

These extended metrics are relatively simple and assume a sequential process without asynchronous chunked writes. In practice, it might be the case that the data is compressed in blocks (or chunks) and transmitted when ready. This means the second term representing the transfer time is incorrect and should be lower, as some data with length L_A is already transferred when the compression is finished. This means the length of the data that still has to be transmitted will be $L_C - L_A$ instead of L_C.

While these equations are good enough to objectively compare different compression schemes in simple scenarios, some more elaborate models for both compression and decompression have been made in the past [113].

$$\begin{aligned} T_{\text{transfer}} &= T_C + T_t \\ &= \frac{L_u}{TH_c} + \frac{L_c}{TH_B} \\ &= \frac{L_u}{TH_c} + \frac{\frac{L_u}{CR}}{TH_B} \\ &= \frac{L_u}{TH_c} + \frac{L_u}{CR \cdot TH_B} \end{aligned} \tag{2.4}$$

$$\begin{aligned} TH_{\text{transfer}} &= \frac{L_u}{T_{\text{transfer}}} \\ &= \frac{L_u}{\frac{L_u}{TH_c} + \frac{L_u}{CR \cdot TH_B}} \\ &= \frac{L_u}{L_u \cdot \frac{TH_c + CR \cdot TH_B}{TH_c \cdot CR \cdot TH_B}} \\ &= \frac{1}{\frac{TH_c + CR \cdot TH_B}{TH_c \cdot CR \cdot TH_B}} \\ &= \frac{TH_c \cdot CR \cdot TH_B}{TH_c + CR \cdot TH_B} \end{aligned} \tag{2.5}$$

Another metric that is often overlooked but is highly relevant, especially for GPU-based implementations, is memory usage. Ideally, compression is done in place, i.e., compression is directly done in the memory buffer of the input data. While some compression schemes exist, such as run-length encoding (RLE) and delta encoding, most schemes are incapable of in-place compression due to the challenges inherent to in-place memory operations and use a separate output buffer. Some other algorithms, like GDeflate and ZSTD, use even more memory [71, 114].

Memory consumption is not a primary concern in most algorithms' design phases because compression can be chunked. If a single data buffer cannot fit into memory, the compression can be completed

in phases, and the results can be concatenated. This is an acceptable solution for both CPU and GPU-based implementations when the goal is only to (de)compress data for storage or transport, assuming there is temporary storage that can store data before compression without becoming a bottleneck itself.

However, in the memory offloading case we described in Section 1.1.4, this temporary storage does not exist. Since one of this thesis's research goals is to discover if we can accelerate memory offloading, we must consider memory consumption, as this directly influences the usefulness of any solution. For example, it would be unacceptable for a compression scheme to require 50GB of memory to compress 5GB of data.

Although not specific to compression, energy consumption is a critical factor for evaluating the overall efficiency of any algorithm. This consideration is especially relevant in data center environments, where energy usage accounts for a substantial part of operational costs [89]. The environmental impact of data centers is also significant [26, 98]. Therefore, it is crucial to assess the energy footprint of the proposed algorithm to ensure that it does not impose excessive energy demands relative to its performance benefits.

2.2. Types of lossless compression

We have established some key performance metrics for compression algorithms, and that two general compression schemes exist: *lossless* and *lossy* schemes. However, to take meaningful steps to accelerate compression on a GPU, we need to make a more apparent distinction between different types of algorithms.

One way to group the plethora of compression algorithms is to look at their underlying technique. However, this is not trivial since algorithms often use several methods, and the lines between techniques are sometimes blurry. Some attempts have been made to formalize the classification of compression algorithms in the past, but they are not always conclusive or give multiple ways to classify an algorithm [86, 67, 94, 50]. In general, we can roughly distinguish between the different techniques listed in Tab. 2.1.

Category	Description
Entropy-based compression	Encodes data based on probabilities, often using statistical models
Pattern recognition	Identifies and replaces repeating patterns with shorter codes
Transform-based compression	Rearranges data for better compressibility
Prediction-based compression	Predicts and encodes differences between data points
Run-length encoding	Compresses sequences of repeated symbols/values
Bit-level compression	Optimizes data representation at the binary level

Table 2.1: Types of compression techniques (not exhaustive)

Another, more coarse, way to group (lossless) compression algorithms is to consider their complexity. This is a subjective measure and arguably less correct than using the above categories. Still, it is easier to better understand the *fundamental* techniques without being overwhelmed by all variations. Subsequently, for the remainder of this section, we will first elaborate on what we consider to be the fundamental compression techniques and then discuss some (well-known) derivations.

2.2.1. Fundamental techniques

The main goal of lossless compression algorithms is to eliminate redundancy in the information. Even if the techniques are different, the underlying goal is identical.

The distinction between a fundamental technique and one that is not can be vague. Not all techniques listed here are used in every compression scheme, and some do not use any of the techniques below. However, many compression schemes are variations of these techniques or at least incorporate some. An alternative way to differentiate between techniques is to consider them *lightweight* techniques, achieving compression with a simple algorithm. Other techniques, or schemes, might achieve considerably higher compression ratios, but at the cost of complexity or throughput.

- **Run-Length Encoding (RLE)**: Run-length encoding is possibly the most well-known example of removing redundant information from a data stream. It achieves this by replacing repeating

tokens, or runs, with the token and the number of occurrences. This is particularly effective when the original data contains many repeated tokens.

Example:

- Input data: `XXXYZZZZYYY`
- Compressed data: `X3Y1Z4Y3`

- **Dictionary encoding**: Dictionary encoding is a class of algorithms. They operate by substituting predesignated tokens with shorter tokens. They use a data structure called the dictionary to map tokens to symbols. A dictionary coder will scan the input data for the tokens in the dictionary, and all matches will be replaced by their respective token. The key to an efficient dictionary coder is creating a mapping that will lead to a high compression ratio, so the longest and most occurring tokens should be replaced with the shortest symbols. Dictionary encoders fall in the *Pattern recognition* category in the compression technique categories above.

  Example: Assume the dictionary contains (`RED=1`, `BEAR=2`, `GOLF=3`, `ESCAPE=$`). Note that tokens with no entry will need to be escaped somehow to distinguish between a symbol and a regular byte during decompression.

  - Input data: `REDBEARBEARGOLFCOURSEGOLFGOLF`
  - Compressed data: `1223$COURSE$33`

- **Bit packing**: Usually, compression schemes work at the byte/word level, so they focus on removing redundant bytes. In some instances, there is also the opportunity to work at the lowest bit level to eliminate redundant information, and this is where bit packing can be beneficial. This technique focuses on instances where not all available bits are used to encode information. Those redundant bits can encode helpful information in those instances, leading to data compression.

  Example 1: One example is where not all bits are used, and bytes can be packed together

  - Input data: `00001011 00001100 00001110 00000010`
  - Compressed data: (len: 4) `10111100 11100010`

  Example 2: Another example is where all bits are used, but only the least significant bits vary. The LSB can be packed together in this case, and the base is only stored once. Note that *example 1* is essentially a special case of this example.

  - Input data: `11001011 11001100 11001110 11000010`
  - Compressed data: (base: `1100`) `10111100 11100010`

- **Delta encoding**: Delta encoding stores values as relative differences (deltas) between the previous value. Delta encoding does not necessarily decrease the data size on its own, but it introduces more repetition in the data and reduces the absolute range of the data. This allows for better compression results when used with other compression techniques. This would be a *prediction-based* encoding technique.

  Example:

  - Input data: `12000 12001 12002 12003 12004 12304 12009 12008 12007 12006 12005`
  - Compressed data: `12000 1 1 1 1 300 -295 -1 -1 -1 -1`

- **Cascading**: Cascading is not a compression technique but can combine several other techniques and enhance their overall output. Combining a *transform-based* technique with another scheme can be especially useful. Cascaded compression itself could be considered a *transform-based* compression technique, but in reality, multiple different techniques could be used together, making it hard to classify. In practice, RLE, Delta encoding, and bit-packing are often used where RLE and Delta encoding are interleaved, and bit-packing is performed on all resulting data [70].

  Example: An example cascaded scheme is a dictionary encoding, followed by RLE, and finalized with bit packing. Note that this is a superficial sample, but it highlights how cascading schemes can achieve higher compression ratios than individual schemes.

  - Dictionary: RED=1, TOP=2, GOLF=3, SIX=4
  - Input data (66 bytes): REDREDTOPTOPTOPTOPGOLFGOLFSIXSIXGOLFGOLFGOLFGOLF
  - After DICT (15 bytes): 1122223333443333
  - After RLE (10 bytes): 1 2 2 4 3 3 4 2 3 4
    - Binary: 00000001 00000010 00000010 00000100 00000011 00000011 00000100 00000010 00000011 00000100
    - Isolating LSB: 001 010 010 100 011 011 100 010 011 100
    - After bin packing: 00001010 01010001 10111000 10011100
  - Compressed data (4 bytes): 10 81 184 156

- **Huffman encoding**: Huffman encoding [45] is the most common entropy technique focused on assigning shorter symbols to more frequent tokens and longer codes to less frequent symbols. A nice property is that this type of encoding is independent of the kind of data and only relies on the statistical distribution of symbols. Huffman encoding builds a binary tree (the Huffman Tree) based on token frequencies, where frequently occurring symbols are closer to the root.

  Example:

  - Input data: BANANA BANDANA
  - Codes assigned based on the tree in Figure 2.2:
    - A: 0
    - N: 10
    - B: 110
    - D: 1110
    - Space: 1111
  - Output data: 110 0 10 0 10 0 1111 110 0 10 1110 0 10 0

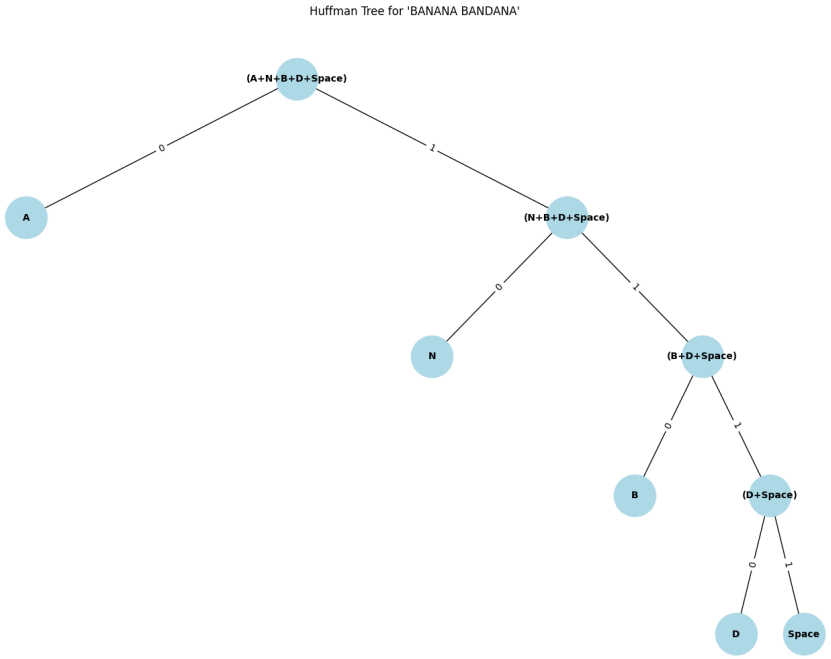


Figure 2.2: Example Huffman tree for sample text. Concatenating edge labels can find the final code for a leaf node.

While the abovementioned techniques are fundamental, and many other compression schemes are derived from these, this list is not exhaustive. We focused on some lossless methods used for general

applications, but many techniques are specific to a single domain or for lossy purposes. Also note that while these encoding schemes will generally reduce the data size, depending on the input data, they can have the opposite effect and increase the output data instead. This is a property of all encoding schemes and will highly depend on the type and entropy of the input data. Correctly understanding the strengths of a specific encoding scheme concerning its input data is, therefore, vital to achieving satisfactory results!

2.2.2. Derived schemes

Besides the fundamental techniques, there are many different compression algorithms. Some are entirely new techniques, but very specific; others are applications of the fundamental techniques.

- **Lempel-Ziv (LZ) family [130]**: The LZ family of algorithms uses a sliding window approach to find and encode repeated sequences. Some variants, like LZ78 and LZW, build dynamic dictionary tables. This family of compressors is widely used in formats like ZIP, PNG, GIF, and tools like gzip and 7-Zip.

  Example:

  - Input data: `ABABABAB`
  - Output data: `A B (2,2) (4,4)`
  - Decompression 1: `A B AB (4,4)`
  - Decompression 2: `A B AB ABAB`

- **Arithmetic coding [121]**: Arithmetic coding is a form of entropy encoding, different from Huffman Coding. It encodes a full sequence of characters to a single floating-point number between zero and one by narrowing a numerical range based on the probabilities of the symbols in the sequence. Arithmetic coding is very efficient for sequences with skewed probabilities, as it can achieve close to the theoretical compression limit.

- **Prediction by partial matching (PPM) [66]**: PPM uses a context model to predict the next symbol based on previous ones. It tries longer contexts first (e.g., last four symbols), then falls back to shorter ones. Probabilities are based on symbol frequency in these contexts, and encoding is often done via arithmetic coding. PPM is used in compressors like PPMd (used in 7-Zip) and PAQ.

- **Fast Static Symbol Table (FSST) [13]**: FSST is a byte-level compression algorithm designed for fast compression of short strings. It constructs a static table of common substrings and encodes them as unused byte values. FSST works well with low-entropy data like log entries, UUIDs, and JSON fields, and is used in columnar databases like DuckDB.

  Example: FSST works precisely like the dictionary encoding example in the previous section. The difference is that FSST uses a static symbol table, enabling random data access.

- **SEQUITUR [68]**: SEQUITUR is a grammar-based algorithm. It constructs a context-free grammar by identifying repeated digrams (adjacent symbol pairs) and replacing them with non-terminal rules. It builds a hierarchical representation of the input. It is used in theoretical compression research, DNA sequence analysis, and structural pattern discovery.

  Example:

  - Input data: `abcab`
  - Grammar rule 1: `A->ab`
  - Grammar rule 2: `S->AcA`
  - Compressed rule: `S`

- **Asymmetric Numeral Systems (ANS) [25]**: ANS is a newer entropy coding method offering compression ratios close to arithmetic coding, but with faster encoding/decoding. It uses a single integer ”state” instead of intervals. There are two primary flavors: rANS (range variant) and tANS (table-based). ANS is used in modern formats like Zstandard (zstd), Facebook’s ZPAQ, and video/audio codecs.

- **Burrows-Wheeler Transform (BWT) [17]**: BWT is a block-based reversible transform. It rearranges data so that similar characters are grouped, improving the effectiveness of further compression steps like Run-Length Encoding (RLE) or Huffman coding, used in bzip2, Zstandard, DNA sequence compression, and search engines for indexing (FM-index).

  Example: `BANANA$` will be transformed to `BNNAA$A` to group similar characters together

- **Byte pair encoding (BPE) [28]**: BPE is a simple greedy compression algorithm. It finds the most frequent adjacent byte pair and replaces it with a new token (often a new symbol or number). It repeats this until no gains are possible. BTW was originally used for text compression, but is now common in tokenization for neural language models like GPT, BERT, etc.

  Example:

  - Input data: `aaabdaaabac`
  - Step 1 (aa -> Z): `ZabdZabac (aa->Z)`
  - Step 2 (ab -> X): `ZXdZXac (aa->Z),(ab->X)`
  - Step 3 (ZX -> Y): `YdYac (aa->Z),(ab->X),(ZX->Y)`

- **842 [41]**: 842 is a variant of the original LZ compression with a different dictionary length. It offers higher compression throughput at the cost of an approximate reduction in compression ratio of ten percent. IBM implemented this algorithm in their POWER processors with hardware acceleration, and there also exist decompressor implementations for FPGAs and GPUs [88, 104].

The above list is only a fraction of all compression algorithms available. These are some well-known algorithms, but many more exist: Context mixing (CM) [61], Context tree weighting (CTW) [119], Dynamic Markov compression (DMC) [22], Golomb coding [33], Re-Pair compression [53], Deflate [24], and many more.

2.2.3. Data dependencies

When looking ahead at the parallelization of (one of) these schemes, in an ideal world, every thread would work on its block and output its result to a fixed location in a memory buffer. Unfortunately, this view underestimates the reality and the complex problems that arise.

One key fact we have omitted thus far is that most of these schemes have internal data dependencies. This means that some information regarding the surrounding data chunks is required to compress (or decompress) a data block. In the worst case, the entire prior data block must be processed before the next block can start.

An example of this data dependency can be seen in LZ compression schemes, where backreferences are used. For (naive) compression, a data block can only be compressed when all its references have been resolved. When decompressing, a reference can only be resolved when the previous chunks have been decompressed. Some solutions have been proposed, such as using a voting mechanism to process nested backreferences or prohibiting nested backreferences during the compression phase [100]. Another potential, more universal, solution is using a tile-based mechanism [4, 3, 96, 114]. With tiling, the data is divided into tiles with no inter-tile dependencies. While this solution could be applied to most compression algorithms, it could potentially negatively influence the compression ratio.

Another common dependency is when the compressed data relies on other compressed data (or decompressed to decompressed when decompressing). The problem becomes clear when asking the following question: When writing a compressed block to its destination buffer, where should it be written to in the buffer? Writing a block N means knowing where $N-1$ ends, which requires information about its starting position and length. Unfortunately, this is a recursive issue because block N requires $N-1$, which unrolls all the way to the first block. Some solutions exist, such as doing a second pass, which concatenates all buffers or performing stream compaction [27, 100, 13], but they can potentially lower throughput and do not work for all schemes. In the case of decompression, metadata from the compression phase can be used to solve this problem [114].

This is not to say that parallelizing compression schemes is an impossible feat. On the contrary, many GPU-based implementations exist, which we will cover in Section 2.3.2. However, we want to highlight that these data dependencies are inherent to all compression schemes, and overcoming them

so that the throughput or compression ratio is not significantly reduced is possibly the most complex challenge to accelerating compression schemes. Carefully considering the involved data dependencies and how they can be resolved should be an essential factor in determining which scheme has the potential to be parallelized.

2.3. Accelerating compression

Until now, we have primarily focused on *how* different algorithms work, but not on their performance in terms of our key performance indicators, compression throughput, and ratio. Our final goal is to design and implement an accelerated compression scheme, but we first need to see the performance of existing state-of-the-art schemes. Furthermore, we would like to investigate possible candidates for acceleration.

2.3.1. The CPU compression landscape

As we've established in Section 2.2, there are many compression algorithms, and it is hard to classify them all. This also makes it challenging to compare all the existing types of compression algorithms, but several attempts have been made.

One such attempt is the *Large Text Compression Benchmark* [60]. The benchmark "ranks lossless data compression programs by the compressed size (including the size of the decompression program) of the first 109 bytes of the XML text dump of the English version of Wikipedia on Mar. 3, 2006" [60], making it a good overall benchmark to compare different types of lossless compression algorithms. This benchmark differentiates between various compression algorithms, so we can get an overall trend of all compression algorithms by aggregating them based on their assigned category. It shows that compressors based on context mixing currently achieve the highest compression ratios, while LZ-based compressors generally score well on the compression throughput.

Another benchmark is *LZBench* [101]. This benchmark focuses specifically on LZ-based compression algorithms and can be used to benchmark several algorithms at once using our datasets. We have also added some other algorithms to this benchmark so that we can add some interesting alternatives. We use three datasets for this test and will also use them for all other tests in this thesis. We use the TPC-H dataset (specifically the *lineitem comments*) [111], the GDelt dataset (specifically the *locations*) [30], and the DBText corpus (specifically the *hex, yago, email, wiki, uuid, urls2, urls* columns) [13]. We use the same datasets throughout the entire thesis, and we will discuss their contents and properties in more detail in Section 5.1.2. The benchmark results can be found in Figure 2.3.

The individual performance of the compressors depends on the actual dataset used. For example, we can see that *zstd* performs well on the TPC-H data (specifically the customer data), but in the other datasets, it is more in line with the different algorithms. For *lzav*, it is the exact opposite case. Some algorithms perform well in all datasets, such as FSST, LZ4, and kanzi. However, kanzi was unable to compress part of the DBtext dataset. We can see the tradeoff between throughput and ratio, especially in the TPC-H dataset.

One aspect not considered in LZBench is the ability, or at least the potential, to compress data multithreaded. FSST excels in this, as every block can be compressed independently since static tables are used. This is not the case for most other algorithms. The only exception is LZ4, which received a significant update in 2024 that introduced multithreading [21], and as a result, improved compression throughput by a factor of six, depending on the underlying hardware.

2.3.2. GPU compression algorithms

While compression is a process that is historically better suited for CPU implementations due to its sequential nature [85], many GPU-based compressors also exist. Some are accelerated versions of existing algorithms, while others are novel GPU algorithms. Usually, the focus is on decompression throughput, but some are also optimized for compression throughput. A (non-exhaustive) list of different GPU compressors can be found in Table 2.2. For general-purpose compressors, we use the throughput reported for string data and mention numerical data separately if interesting.

In addition to reported values, we benchmarked the nvCOMP compressors [71] on the three test datasets in Figure 2.4, in addition to GPULZ [128] and pipelines with one, two, and multiple stages generated with the LC framework [10]. For GPULZ, we use three configurations: fast, average, and max-compression, which match the configurations based on the original authors' parameter sweep of

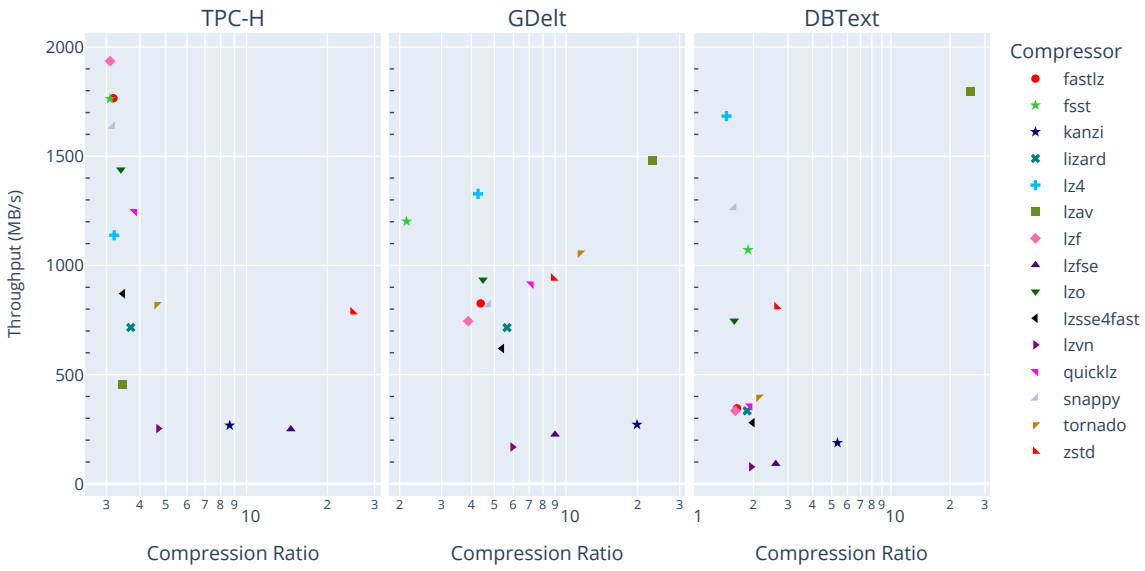


Figure 2.3: The results of running LZBench on the TPC-H, GDelt, and DBText datasets using a Ryzen 9 9950X.

`(C=4096, W=32, S=4)`, `(C=4096, W=128, S=2)`, and `(C=4096, W=255, S=1)`, respectively.

Based on this list, we can already make some observations. For one, compressors targeting numerical data are on average significantly more performant than their general-purpose counterparts. On textual data, the performance of compressors drops significantly. For example, LZ4 has a 94 percent drop in performance when compressing string data compared to numerical data. We can also see that there is a significant drop in throughput above a particular compression ratio, with differences as substantial as a 97 percent drop in throughput when comparing the compressor with the highest throughput (ANS) to the compressor with the highest compression ratio (ZSTD).

2.3.3. Acceleration candidates

We have now seen several CPU-based compressors and both novel and ported GPU-accelerated compressors. Suppose we aim to fill the gap between high throughput and high compression. In that case, we can either try to modify an implementation with high throughput to increase its effective compression or we can try to accelerate a compressor with a high compression ratio.

When looking at the performance of existing compressors in Table 2.2, there are only two compressors with performance high enough to make them interesting to modify in terms of compression ratio: ANS and Cascaded [71]. Unfortunately, these are proprietary compressors with a closed-source implementation, making them infeasible to modify.

The next best option is GPULZ [128], which is based on LZSS [103] and is essentially an improvement on CULZZ [82, 84, 83]. This compressor achieves an acceptable compression ratio, but its throughput is too low to be used in the shuffling use case. This is already a highly optimized version, supported by the fact that this outperforms all other compressors in the same family: CULZSS [82], Culzss-bit [81], GLZSS [131], and G-Match [59]. The remaining options have such a low throughput that they would have to be improved by a factor of 5 to 50, which is unlikely.

Our focus then shifts to accelerating CPU compressors. LZ-based compressors have been widely studied, with the results being compressors like GPULZ [128], LZ4 [71], and G-Match [59]. Therefore, we want to avoid all LZ-based compressors, like Snappy, ZSTD, and Deflate. Instead, we would like to use a scheme with the least data dependencies during compression as possible, as described in Section 2.2.3.

This makes the *Fast Static Symbol Table* (FSST) compressor [13] a compelling option for several

GPU Algorithm	Data type	Reported compression throughput	Ratio
ANS [71]	General purpose	Up to 271.1GB/s (H100) [72]	1.7
Cascaded [71]	General purpose	Up to 171.3GB/s for GP, 459.7GB/s for num (H100) [72]	1.0
Snappy [71]	General purpose	Up to 28.7GB/s for GP, 81.9GB/s for num (H100) [72]	2.3
GPULZ [128]	General purpose	Up to 22.9GB/s (A100)	3.1
.. ANS decoding on GPUs [116]	General purpose	Up to 22GB/s (V100)	N/A
LZ4 [71]	General purpose	Up to 14.8GB/s for GP, 244.5GB/s for num (H100) [72]	2.1
.. transforms for bzip2.. [117]	General purpose	Up to 11.6GB/s (8x H100)	5.6
Deflate [71]	General purpose	Up to 10.5GB/s for GP, 85.5GB/s for num (H100) [72]	2.6
ZSTD [71]	General purpose	Up to 9.0GB/s for GP, 88.8GB/s for num (H100) [72]	5.1
Gdeflate [71]	General purpose	Up to 7.9GB/s for GP, 96.1GB/s for num (H100) [72]	2.6
G-Match [59]	General purpose	Up to 2.2GB/s (GTX 980)	1.7
GLZSS [131]	General purpose	Up to 1.8GB/s (GTX 980) [59]	1.7
CULZSS [82, 84]	General purpose	Up to 1.1GB/s (GTX 980) [59]	1.3
Culzss-bit [81]	General purpose	Up to 0.8GB/s (GTX 980) [59]	1.6
Gompresso [100]	General purpose	Up to 0.3GB/s (K40)	2.1
Recoil [57]	General purpose	Not reported (RTX 2080Ti)	N/A
Bitcomp [71]	Numerical	Up to 708.6GB/s (H100) [72]	1.4
SPspeed [9]	Numerical	Up to 510GB/s (RTX 4090)	1.4
SPratio [9]	Numerical	Up to 350GB/s (RTX 4090)	1.6
DietGPU [49]	Numerical	Up to 350GB/s (A100)	1.3
Ndzip [51]	Numerical	Up to 259GB/s (A100)	1.4
GFC [79]	Numerical	Up to 32.3GB/s (K40) [123]	1.2
MPC [123]	Numerical	Up to 10.8GB/s (K40)	1.5
Fast LZW compression .. [27]	TIFF images	N/A	4.5

Table 2.2: This table lists several GPU compression algorithms, their targeted data type, and their reported throughput. We use the throughput that the original paper reported or a later report of a third party if it is higher (which will then be referenced). The list is sorted by the reported throughput and grouped by the data type.

reasons. For one, to the best of our knowledge, no accelerated compressor has made use of this compression scheme. Only a high-performance decompressor was recently released: GSST [114, 113]. This is an advantage, since there is a GPU-accelerated decompressor we can potentially integrate with. Furthermore, FSST uses static tables, which means there only exists a data dependency between compressed blocks, which can be solved efficiently on the GPU using a gather operation. Unlike LZ-based compression, there are no backreferences, only references to the static encoding table. This property makes it interesting from a GPU perspective.

Another reason that makes FSST interesting is its ability to perform random access decompression. FSST is part of the DuckDB system because columns can be compressed and only decompressed when required. If we maintain this property with our compressor, the GPU-accelerated version of FSST can be used on GPU-accelerated database systems for the same reason.

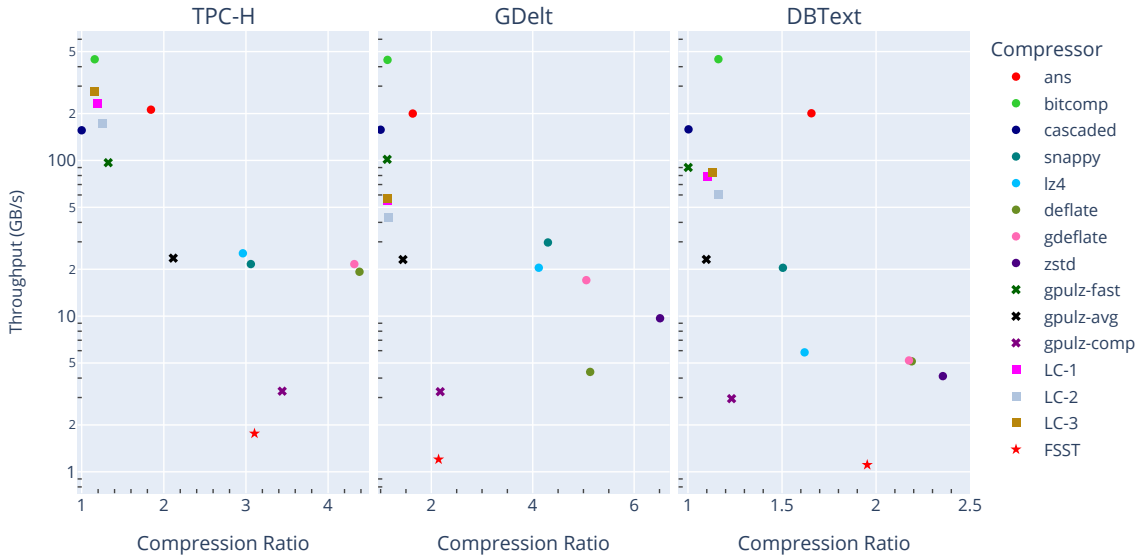


Figure 2.4: The overall performance on our datasets using the nvCOMP compressors, GPULZ, compressors generated with the LC framework with a different number of stages, and the original FSST algorithm. All benchmarks were completed on the same machine (RTX 4090 with Ryzen 9 9950X) and used the same 2GB files.

2.4. Fast Static Symbol Table

FSST is a dictionary coder that replaces frequently occurring strings (symbols) with smaller single-byte symbols that are one to eight bytes. We have seen that FSST [13] is an interesting candidate for several reasons, which we will elaborate on in Section 2.4.1.

The FSST compression process involves creating a symbol table for every block and then replacing matching entries in the block with their corresponding codes. Bytes not matched by any symbol in the table will be escaped with a special character. Figure 2.5 shows an example of the FSST compression process. During encoding, FSST transforms the input data stream into a smaller one using the symbol table, or encoding table, for every block. Decompression is the reverse operation, where blocks will be decoded using their symbol tables and then concatenated to form the full original data stream. We will briefly discuss the two main phases: symbol table generation in Section 2.4.2 and encoding in Section 2.4.3.

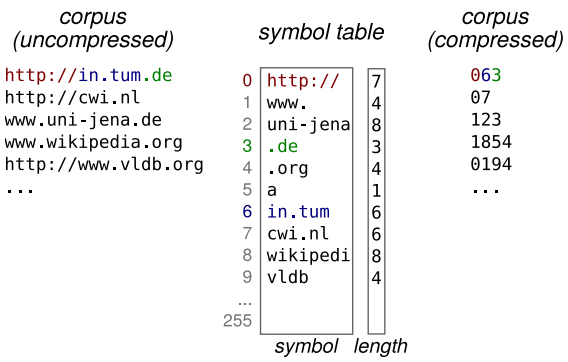


Figure 2.5: An example of FSST compression. The uncompressed data is encoded to a (smaller) format using a static dictionary. Source: [13]

2.4.1. FSST as an acceleration candidate

At 1.5 GB/s, the single-threaded implementation of FSST already achieves high compression throughput compared to other compressors. As decompression is relatively straightforward, it performs similarly well when comparing decompression throughputs. Additionally, FSST results in high compression ratios. Furthermore, when considering the type of data dependencies within FSST, the only dependency is the length of previous compressed data buffers, like the second example in Section 2.2.3. Unlike other dictionary-based algorithms [27, 130, 103, 82, 131], FSST uses a static table, which means every block can be (de)compressed independently, and data within a block does not depend on each other.

FSST is already a compelling candidate based on these results. What makes it even more compelling is the introduction of a GPU-accelerated decompessor that was recently introduced: GSST [114, 113]. While this implementation only provides a decompression scheme for FSST, it confirms that the static table allows for excellent parallelization and that data blocks can be further split up to introduce thread-level parallelization without sacrificing too much compression ratio. It reports a decompression throughput of 191 GB/s and a compression ratio of 2.74 on an A100 with the testing data, outperforming existing solutions when considering the combined transfer throughput similar to what we showed in Section 2.1. In addition to that, GSST has a considerably lower memory footprint than others.

While the results obtained for decompression do not necessarily guarantee similar results for compression, they do show that there is potential, given that we can overcome challenges specific to compression. Therefore, we will base our accelerated compression scheme on the FSST compression scheme and integrate with the GSST decompressor to achieve a high overall (de)compression throughput.

2.4.2. Table generation

Possibly the most critical step in achieving a good compression ratio is table generation. This is a particularly challenging step, since the chosen symbols will affect the effectiveness of others. Simply choosing symbols greedily will not yield the maximum compression ratio. This makes it difficult to estimate the gain of symbols accurately.

FSST mitigates this by using two key components in the table generation algorithm. These components are using on-the-fly compression to 'learn' the true worth of symbols, and performing multiple iterations of the generation algorithm. Every iteration starts with the symbol table of the previous iteration and attempts to improve the table for the next iteration by selecting new promising symbols. Promising symbols are all symbols in the current symbol table, all concatenations of occurring pairs of symbols, all symbols of a single byte, and all extensions of current symbols with the next occurring byte. The algorithm starts with an empty symbol table, expanding everything to escaped characters. All promising symbols are then created and ranked based on their apparent gain, which is just their length multiplied by the number of occurrences. The best symbols are then chosen to be part of the next symbol table. This process is repeated for several iterations.

The number of iterations partially determines the overall quality of the resulting symbol table. Initially, there are many smaller symbols, but with more iterations, the symbols will grow in length and quality. The authors of FSST determined that performing five iterations is generally good enough to converge to an efficient symbol table. Another factor is the size of the training sample. The authors found experimentally that a modest sample will already result in compression ratios similar to those of using all data. We can also use this fact to achieve a higher throughput for table generation.

2.4.3. Encoding

Encoding is essentially a find-and-replace operation on the data stream; it replaces every occurrence of a symbol in the encoding table with its respective code. Conceptually, this can be implemented by sequentially scanning a data stream, finding the longest matching symbol (the longest matching prefix of the current eight bytes), and replacing the token with its symbol. While this is the easiest solution, it does not allow for parallelism and limits performance. In the original FSST paper, the authors also provided an alternative SIMD implementation using AVX512. This implementation provides a foundation for a GPU implementation, as AVX512 limits the developer to specific instructions and disallows branching.

One important data structure the authors introduce to avoid branching (and is more efficient than a linear search) is a (nearly) perfect hash table. A regular hash map would not work, as the keys are of different lengths, which are not known in advance. The authors proposed a solution where only the

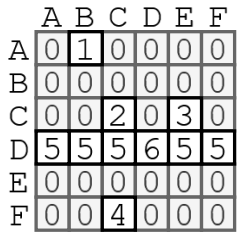


Figure 2.6: An example of the lookup matrix used in FSST for short symbols (length of one or two). For simplicity, this matrix has been changed to only list A through F, but in reality, this is a $256x256$ matrix for all possible characters. Six symbols have been encoded in this example: `AB (1)`, `CC (2)`, `CE (3)`, `FC (4)`, `D (5)`, and `DD (6)`. A lookup would use the two characters of a symbol to identify the row and column in the matrix. This is why a symbol consisting of a single byte fills an entire row, such that a lookup starting with the correct symbol can be paired with any second symbol.

first three bytes are used to create hash tables. This eliminates the issue of variable-length keys but introduces two issues.

First, by using the first three bytes to hash every entry in the hash table, shorter symbols with only one or two characters cannot be used. This was solved by adding a lookup structure specifically for the short entries. This lookup structure is a $256x256$ matrix filled with codes. The matrix is filled such that the code for the symbol AB is in row X and column Y, where X and Y correspond to the 8-bit number representing characters A and B, respectively. In addition to that, all unassigned cells in row X (where X corresponds to symbol A) are set to the code of symbol A. All remaining cells are set to an escape code. An example illustration can be found in Figure 2.6. For every encoding step, a code is retrieved from this data structure using the first two characters, and a code from the hash table is retrieved using the hash of the first three characters. The hash table result will be returned if there is a hash table hit. Otherwise, the result from the lookup matrix is used. This way, all symbol lengths will be supported appropriately.

The second issue arises from hash collisions. Since only the first three bytes are used in the hash function, any symbol that is a prefix of a longer symbol will be a hash collision, in addition to typical hash collisions. The authors showed that using the most impactful symbol and disregarding all hash collisions has a low impact on the actual compression ratio but is significantly faster than using perfect hash tables or a solution like probing. The retrieval code uses the hash of the first three bytes of the current word to retrieve the best symbol for that hash and then uses an additional comparison to check equality.

2.4.4. GSST modifications

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

One problem with this approach is that it negatively affects the compression ratio. Every symbol on the border of a split will be decomposed into at least two smaller symbols, reducing the overall compression ratio. The authors of GSST used the TPC-H dataset to show that this effect is minimal.

The other problem is that the location where each thread should output its decompressed data is unknown. There are two ways to solve this: make every tile output a constant amount of decompressed data (varying the compressed data size) or keep track of how much data every tile outputs in the compression stage. GSST relies on the compressor providing metadata detailing the structure of a block, which aligns with the second approach. The decompressor can then use this information in the file header to deduce where every thread should output its data. The file header following their *splits format* can be seen in Figure 2.7.

Overall, GSST achieves considerable throughput while maintaining the high compression ratio that the FSST table generation algorithm provides by limiting the amount of information it needs from a

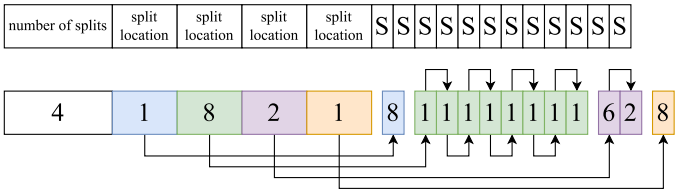


Figure 2.7: The split format GSST uses. Every block is divided into splits, which individual threads will process. Source: [114, 113]

compressor to reconstruct the original output structure. In addition to that, GSST uses significantly less memory than implementations in nvCOMP. This property is useful when decompressing large data files and might lead to an overall throughput gain when using a chunked approach for extremely large files that do not fit into memory.

2.5. GPU development

A Graphical Processing Unit (GPU) is a special processor originating in graphics processing, such as shaders. A GPU follows the *Single instruction, Multiple threads* (SIMT) paradigm, a combination of *Single instruction, Multiple data* (SIMD) and multithreading. This execution model is suitable for algorithms that can be massively parallelized and run on general-purpose GPUs or GPGPUs. This section will briefly describe the general architecture of NVIDIA GPUs and how we can utilize them as GPGPUS. We will also describe some practical techniques available through this interface.

2.5.1. GPU architecture

At the core of GPUs lie many small, various cores (CUDA, Tensor, RT), which enable massive parallelism. The different cores have different specialties, but CUDA cores are the standard processing cores. These cores are grouped in *Streaming Processors* (SMs), each with its own schedulers, register files, and caches. Figure 2.8 shows the SM architecture of NVIDIA's latest Blackwell architecture.

The SMs can execute multiple threads simultaneously, achieving high throughput through parallelism. Threads running on an SM are grouped into thread blocks and into warps, which run in lockstep. This means all threads execute the same instructions, potentially leading to inefficiencies if there is divergence between threads in the same warp.

A GPU has a hierarchical memory architecture, something that is similar to the CPU. The largest and slowest type of memory is global memory. This memory is accessible to all threads and is relatively plentiful, but it has the largest access latency and stricter requirements for optimal bandwidth utilization. The next layer is the L2 cache, which can reduce latency for frequently accessed memory. The next layer is the L1 cache, which functions as shared memory within a thread block. This memory is located on the SMs themselves. It can be used to communicate between threads on the same SM and to cache intermediate results before issuing expensive instructions to global memory. Memory that is not required by the compute load can be used as a regular L1 cache. Finally, there is the register file, which is used directly by the threads. Figure 2.9 shows an example of this memory hierarchy.

2.5.2. Quantifying GPU acceleration limits

The main strength of a GPU is in its massive parallel processing power. When a task can be executed in a parallel fashion, it conceptually makes sense that its overall performance is enhanced. However, there are limitations to what we can achieve with a GPU. There are two 'classes' of constraints: fundamental limitations of the algorithm to be accelerated, and hardware limitations of the GPU.

The first limitation is described by Amdahl's law, which states that the overall performance improvement of accelerating an algorithm depends on how much of the algorithm can be parallelized. Equation 2.6 shows this law, where P is the part of the program that can be accelerated and N is the number of processors. Following Amdahl's law, the most crucial limitation for GPU acceleration is that the overall acceleration potential is limited by the portion of the algorithm that we manage to accelerate.

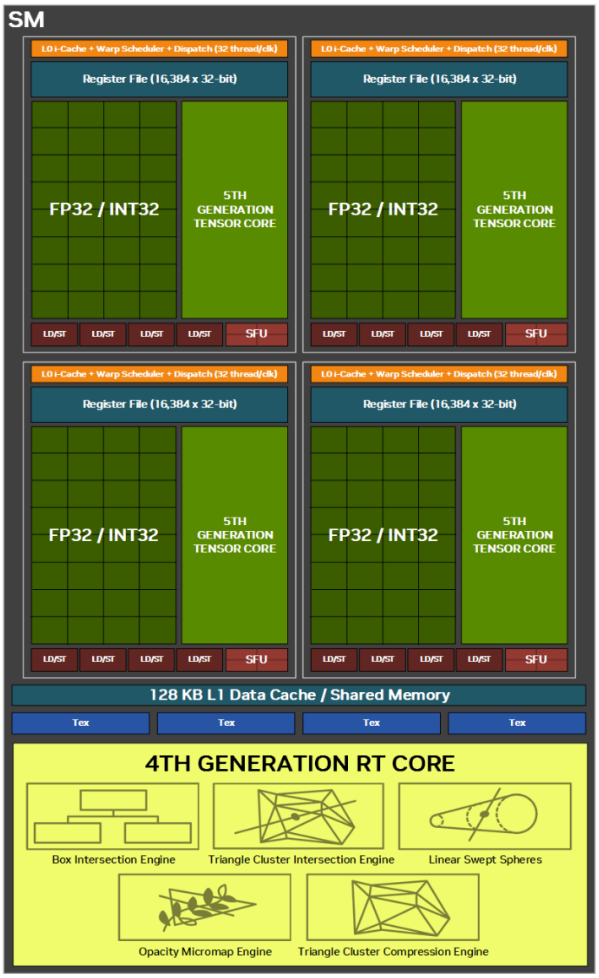


Figure 2.8: NVIDIA Blackwell Streaming Multiprocessor (SM). Source: [76]

Figure 2.9: NVIDIA Blackwell GB202 GPU block diagram. Source: [76]

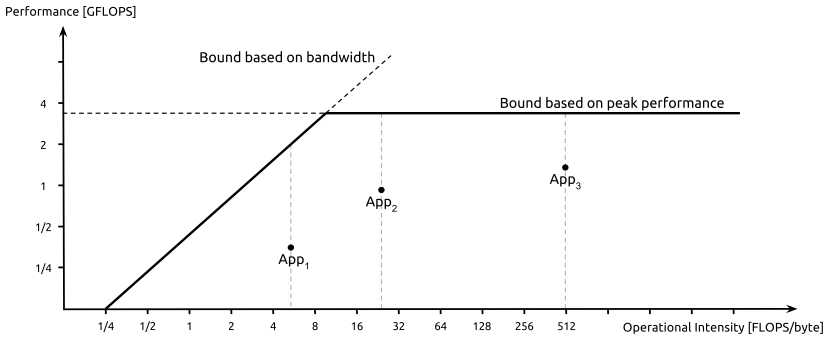


Figure 2.10: A simple roofline model example. It shows the two main limits determined by computational bandwidth and memory bandwidth with respect to the *Arithmetic Intensity* of programs. In this plot, three applications are shown, the first of which is limited by memory bandwidth, while the other two are limited by computational performance. Source: [118]

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.6}$$

While Amdahl's law helps realize our theoretical limits, it is too simplistic to estimate our performance limit accurately. It is more important to consider the two main limitations of GPUs: (peak) memory bandwidth and (peak) computational performance. We measure the memory bandwidth as the number of bytes that can be transferred per second, and the computation performance is traditionally measured as the number of floating-point operations per second (FLOPS). Additionally, we can characterize any kernel by using the *Arithmetic Intensity* (also called *Operational Intensity*), which is defined by the number of floating-point operations per memory operation. The *AI* can then be used to determine if a kernel is memory-bound or compute-bound, by comparing the compute bandwidth to the effective memory bandwidth ($AI \times BW$). Note that in the case of compression, FLOPS are less relevant than integer operations, as most operations will be integer operations.

The roofline model [120] can be used to compare these. A simple roofline model is shown in Figure 2.10. Computational limits and memory limits define the two main bounds. The *AI* determines if the theoretical maximum performance is limited by memory bandwidth or overall compute. This model can be used to determine how to further improve a kernel within the limits of the underlying hardware, by modifying the *AI*.

2.5.3. Compute Unified Device Architecture (CUDA)

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

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

One effect of this architecture is that all threads within a block are guaranteed to use the same L1 memory, which enables its use as shared memory. However, threads in the same grid but not in the same block are not guaranteed to use the same L1 memory.

2.5.4. Streams

CUDA uses the concept of *streams*, a sequence of operations that will be executed in order. As mentioned before, a GPU has the potential to execute multiple kernels concurrently, but it can also perform memory transfers concurrently with kernel execution. This concurrent execution of kernels and memory operations is achieved using various streams. Figure 2.11 shows an example.

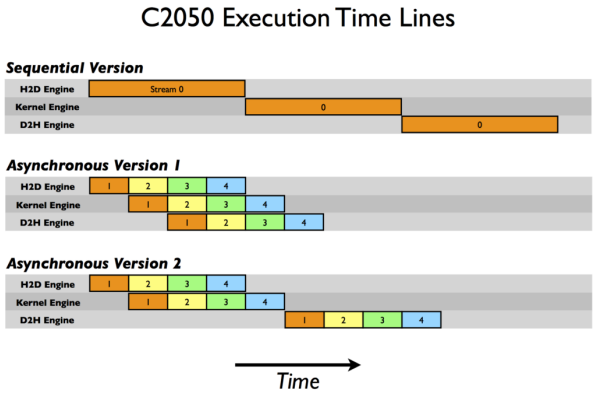
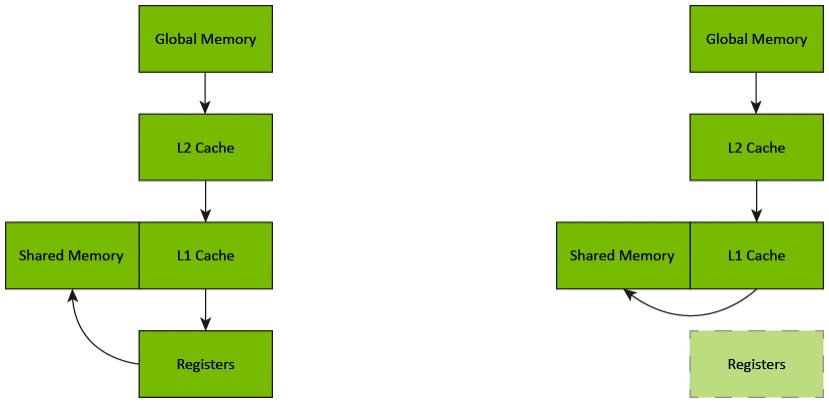


Figure 2.11: The execution timeline of an NVIDIA C2050 using multiple streams to overlap memory transfers. This GPU uses the Kepler architecture, but the example is still relevant. Source: [36]



(a) Before Ampere, data movement to shared memory is always a two-step process. First, data is copied to the kernel's registers, from where it is then copied to shared memory.

(b) With the new asynchronous data movement introduced in Ampere, data can be directly copied to shared memory, bypassing the register file

Figure 2.12: Comparing the regular data flow from global to shared memory to the asynchronous flow introduced in the *Ampere* architecture. Kernels must be changed to take full advantage of this new mechanism, but there is significant potential for performance improvement. This is because compute stages can be overlapped with fetch stages, and register pressure is reduced. Source: [105]

2.5.5. Asynchronous data movement

Most CUDA kernels follow a similar pattern of loading the data, performing some computation/operation on said data, and writing back a result. As we've already established, GPUs are usually bottlenecked by data movement, so ideally, the loading and computation phases are (partially) overlapping. We can use streams and techniques like CudaDMA [11] to allow for at least partial overlapping, but we can do better.

To understand this, we look at Figure 2.12a. With all techniques mentioned until now, the path of data through the memory architecture has been the same: a kernel first loads the data to registers and then loads it into shared memory. The result is a long journey through the entire memory architecture.

With the introduction of the *Ampere* architecture, new mechanisms to control data movement were introduced. This allows the kernel to influence data residency in the L2 cache and copy data into shared memory asynchronously. This directly copies data to shared memory, avoiding the longer path through kernel registers, as shown in Figure 2.12. The compute and fetch stages can be fully overlapped using the async copy engine combined with the pipeline synchronization.

2.5.6. Dynamic parallelism

In a classic CUDA pipeline, the CPU launches multiple kernels, each with a flat grid layout. This works well enough for most processes where inherent loops expose enough parallelism to use the GPU efficiently, but some parallel patterns cannot be expressed easily. An example is fluid simulation, where a coarse-grain grid would lose details, and a fine-grain grid would result in too many (unwanted)

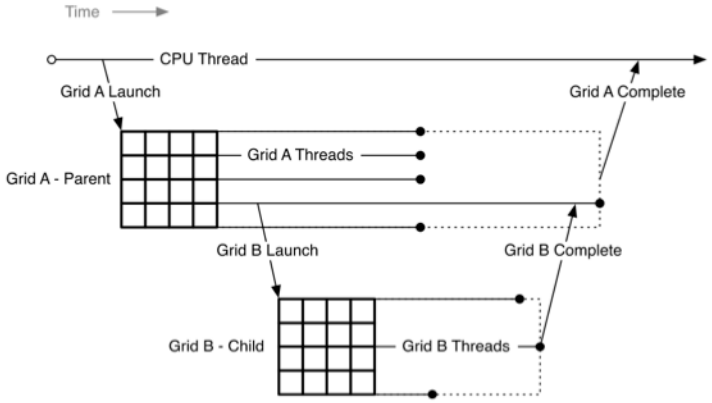


Figure 2.13: With Dynamic Parallelism, parent grids can launch multiple child grids. This process allows for recursive subdivision but can also be used for additional pipelining in some specific cases. Source: [2]

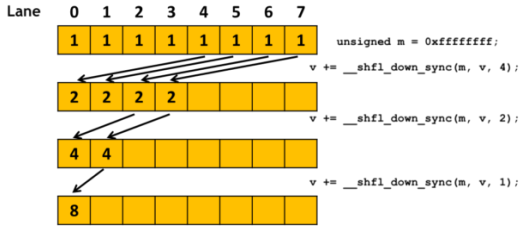


Figure 2.14: An example of a reduction using warp-level register shuffling functions. Source: [58]

computations.

With the introduction of *Dynamic parallelism*, it becomes possible to launch kernels from within kernels. An example is shown in Figure 2.13. This allows kernels with nested parallelism to be more efficiently implemented. In the case of the fluid simulation, the CPU can initially launch a coarse-grain grid, which in turn launches finer grids when required.

Dynamic parallelism is helpful for multiple algorithms, such as algorithms with hierarchical data structures, algorithms using recursion, and algorithms where work is naturally split into independent batches. One crucial aspect that makes dynamic parallelism work for those algorithms is that the CUDA runtime guarantees that parent and child grids have a fully consistent view of global memory. This means that child grids are guaranteed to see all writes by the parent before it is launched, and that the parent is guaranteed to see all writes by the child after the parent synchronizes with the child.

2.5.7. Warp-level primitives

Some algorithms use collective communication operations, such as parallel reductions and scans. These operations require threads to communicate, which is mainly done using shared memory. Cooperative groups [37] provide a higher-level abstraction, but this is not a lower level than shared memory. In those cases, having an additional communication level on the register file level within warps would be beneficial. Warp-level primitives allow communication between threads in the same warp.

CUDA has three categories of warp-level primitives: synchronized data exchange, active mask query, and thread synchronization. With synchronized data exchange, threads can exchange data directly through registers and use voting functions. This allows threads within a warp to perform a reduction fully in the register file, for example, as shown in Figure 2.14. Another example is accelerating stream compaction using ballots [46, 12].

The active mask query and thread synchronization allow for opportunistic warp-level programming, which can be used for algorithms that can use any available threads for their computations. Thread synchronization can enforce a barrier for all threads within a warp, even in diverging branches.

3

Accelerator design

3.1. FSST profiling

Intuitively, the encoding stage of the FSST algorithm is the component bottlenecking the entire process, as this stage needs to process the entire data stream, unlike table generation, which works on a small fixed-size sample of the data. In addition, assuming low entropy in the data, the block size could be extended to scale the table generation throughput. In this section, we will investigate the baseline throughput of the FSST algorithm and the acceleration potential of a multithreaded CPU implementation of FSST.

3.1.1. Baseline throughput

With FSST, every block is processed independently from other blocks, with the exception of output organization. This means we can focus on a single block for detailed profiling, and these results can then be extrapolated for larger data sets. We will also run several tests on full datasets to confirm this extrapolation is indeed valid.

The tests will be run on the TPC-H, GDelt, and DBText datasets. These datasets represent natural data in database systems. For this test, we run the FSST algorithm with minimal changes, only adding some timing code to gather simple statistics: average table generation time per block and average encode time per block. We also gather the total time per stage and then take the average over multiple runs and file sizes.

The results of this test are presented in Figure 3.1. The data confirm that the encoding phase is the most computationally intensive component of the pipeline and, therefore, represents a strong candidate for GPU acceleration. This observation is consistent with the findings of the original FSST paper, in which the authors used a SIMD-based implementation (specifically AVX-512) to accelerate the encoding stage.

3.1.2. Multithreaded CPU implementation

To scale the performance of the table generation algorithm, we can increase the block size without increasing the sample target. In theory, this could lead to a reduced compression ratio, as the table would be less effective at capturing patterns in the data. However, as the authors of FSST also noted, the actual results are minimally degraded when using a larger table size, up until a certain point. In later chapters, we will see that the effects on the compression ratio are indeed minimal.

Another, more obvious, way to increase the overall throughput of table generation and encoding is to use multiple CPU cores. A simple way to achieve this is to create a thread for every block and gather the results once all blocks have been processed. To avoid saturating the CPU or OS scheduler, it would be better to use a more complex thread pool, but that is outside of the scope of this simple test. A shared barrier can be used to signal completion, and then a single CPU thread can initiate memory copies to a single contiguous block of memory.

After modifying the original FSST with the above modifications, we achieve a throughput of 11.49 GB/s for the TPC-H datasets, which is a 6.5x speedup compared to the original implementation. We will use the multithreaded table generation for our future pipelines.

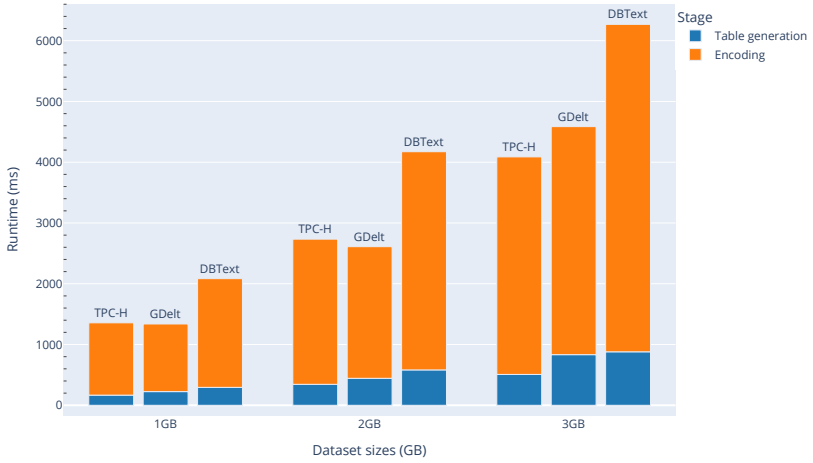


Figure 3.1: The runtime of the two main stages in the FSST algorithm. The test was performed on a system with a Ryzen 7 5800X.

3.2. Acceleration potential of FSST

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

Table generation is highly sequential and uses data structures unfit for a GPU, such as a priority queue. However, table generation only needs a small sample of the data to work with, so modifying this to run in parallel on the CPU will likely already yield high throughput.

In Section 3.1.1, we have found the baseline throughput of FSST, and in Section 3.1.2, we have implemented a multithreaded version of FSST. Based on these tests, we have confirmed this hypothesis and identified the main bottleneck of the FSST compression algorithms: the encoding stage.

The encoding stage operates on all data and, therefore, must be executed on the GPU itself. To achieve parallelism, we can divide the data into blocks and encode each block in a separate thread, a common technique often called tiling or chunking [4, 3, 96, 114], which is similar to the splits concept used in GSST.

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

3.2.1. Applying tiling

After the tables have been created, the encoding stage will start. Encoding is done on a block level, i.e., every FSST block can be encoded separately. This is the first level of parallelism and maps fairly naturally to a CUDA thread block.

To create parallelism within a (thread) block, we could utilize the tiling technique. In that case, we would split the data within a block to multiple tiles, which map to a single thread. This means a single thread works on a small contiguous block of memory, which is part of the original data block, and all threads in the thread block work in parallel to encode a single data block.

An important aspect for this option is determining the tile size. In general, we can distinguish between three options: use a constant tile size, use a variable tile size such that the output tile size is (close to) constant, or use some work-stealing approach where every thread will continue with another tile once it finishes with its own.

Regarding implementation complexity, the work-stealing approach is significantly more complex

and will likely have substantial overhead. Using a variable tile size will require a (partial) preprocessing step, possibly in combination with some heuristics, but has the potential to provide an overall speedup. Constant tile size is the most straightforward approach, but will likely have the highest amount of thread divergence. A valuable property of constant tile size is that the amount of data each thread generates during decompression is equal, which means there is little to no thread divergence. While this property will not benefit the compressor, it will help the full compression and decompression cycle, which is the overarching goal. This is also why GSST uses a constant uncompressed tile size.

Another method would be to create parallelism by performing sub-tasks in a parallel fashion. As the original FSST authors highlighted, the most intensive task within the encoding cycle is finding the best match for the current eight (or fewer) bytes in the encoding table. This process essentially involves finding the longest prefix that matches the current data. In the original FSST implementation, a hash and matrix lookup are used. Still, it would also be an option for all threads within a warp to do individual lookups and determine the best match using warp-native communication, such as ballots[1]. This method would eliminate all thread divergence and allow for simple (and efficient!) data movement from and to global memory.

The decision between the two global approaches will highly influence the final result and the intermediate data structures. Therefore, both options are kept open now, but we will specify which division we will use. In Section 3.4, we will discuss and briefly analyze all options and determine the best way forward.

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

3.2.2. Encoding table storage

One of the key issues in the encoding stage is that the encoding table does not fit in shared memory due to its size. The shortcodes lookup table alone requires approximately 130kB, and the hash table adds an additional 16kB, resulting in a combined memory footprint of around 146kB. This exceeds the shared memory (L1) capacity available on most GPUs, forcing the table to reside in global memory.

This is problematic because global memory is not well suited for the random access patterns typical of encoding table lookups. Accessing global memory under such conditions leads to increased latency and warp stalls, ultimately limiting throughput.

Our final pipeline, which will be discussed in detail throughout this chapter, confirms this hypothesis: a version of the encoding kernel that fits the lookup table into shared memory achieves nearly 11 times higher throughput compared to the global memory variant. We will discuss how we manage to reduce the overall footprint in Section 3.3.

3.2.3. Output organization

As mentioned in Section 2.2.3, most (de)compression schemes will have a data dependency. In our case, this data dependency is between blocks and their output locations. Since blocks will have a variable output length, not all data will compress to the same length. This means a block cannot output its data before all preceding blocks have been compressed. The problem is illustrated in Figure 3.2. There are three solutions to this problem: concatenating the output of blocks using memory copies, precomputing all output locations, or performing stream compaction.

The first solution of concatenating blocks is the most straightforward and will be able to fully utilize the high memory bandwidth of modern GPUs. However, this assumes that the output for a single block is in contiguous memory, which might not be the case. That is because the problem of output organization is present not only at the block level but also at the tile level, since threads will also output variable-length tiles. This means several hundred, or even thousands, small copies would be needed.

The second solution would be to precompute all output locations. This means we would run the

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

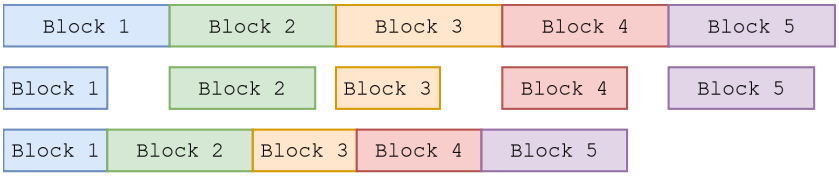


Figure 3.2: In this example, we can see the original blocks on the first line and then the compressed versions on the second line. The problem resides in creating the third line, where we must combine all the compressed blocks. Block 2 cannot be placed until the compressed size of block one is known, and block three cannot be placed until block two's size and starting location are known.

encoding kernel once without writing its results to memory and only keep track of how much data is generated per thread. The second step would be to aggregate these results using a prefix-sum into a struct indicating where every thread would write to. The final step would be to run the encoding kernel, but now, it is using its designated block in memory. This method requires no additional postprocessing but an additional encoding kernel run. In addition to that, some padding would likely be required to maintain memory alignment when writing to output locations.

The final option would be to perform stream compaction, also known as parallel stream filtering. Every thread would get its own block of memory to which it can write its output, adding padding to fill the unused space. All padding is then filtered out in a separate post-processing step. This algorithm is well-known and supported out-of-the-box by standard CUDA libraries such as Thrust[2]. The existing implementations are capable of high throughput. However, one property of FSST is that the worst-case compression is 0.5, or a doubling in size. For this reason, every block should be allocated twice its size, which means the filter has to process twice the amount of input data. This effectively halves the throughput of existing stream compaction algorithms since we calculate throughput in terms of input size.

All options have advantages and disadvantages, so there is not necessarily one option that is better than the others. Stream compaction is the best choice when the encoding kernel outputs a fixed-size block with padding evenly spread over the output block. If the output format is more dense, i.e., all padding will be at the end of a block, memory copies would be more performant. We will determine which strategy to apply in Section 3.4.3, as it heavily depends on the kernel implementation and its output.

3.2.4. Performance considerations

The challenges discussed so far must be addressed to make the algorithm work, but there are also some issues that are mostly related to the expected performance of our pipeline.

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

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

We will address these issues in Section 3.6 and 3.7, respectively.

3.3. Reducing table size

As mentioned in Section 2.4.3, the encoding process uses a hash table and a lookup table. The hash table and the shortcodes matrix. The hash table is used for symbols with a length between three and eight, while the shortcodes matrix is used to encode symbols that consist of one or two characters efficiently.

[2]https://developer.nvidia.com/thrust

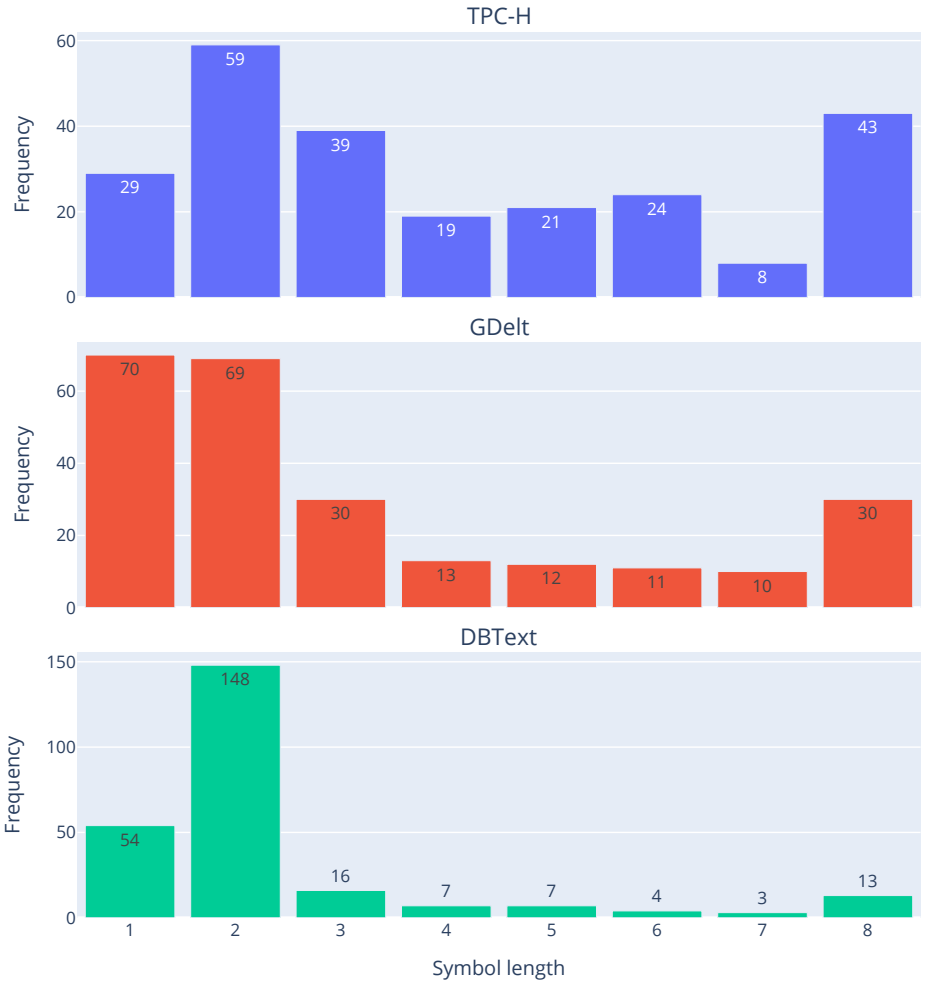


Figure 3.3: This histogram shows the spread of symbols regarding their length for three datasets: TPC-H, GDelt, and DBText. In general, we can see that the DBText corpus heavily uses short symbols, while the other datasets also use longer symbols more often.

Before considering any optimizations, we must investigate what kind of data is in these data structures. For this purpose, we use the same datasets we used earlier: TPC-H, GDelt, and DBText. In Figure 3.3, we can see the histograms for symbol length. We can see that DBText uses a lot of short symbols, likely leading to a lower compression ratio than achievable with TPC-H and GDelt. This is inherent to the underlying data, but it tells us the importance of the lookup table for smaller symbols.

Another important aspect, besides the absolute number of entries in the two data structures, is how the data is divided within them. For example, the hash table has 65 entries on average, which means the hash table is filled for 25 percent, and only 6 percent of memory is actively used. When considering the lookup table, only 21 percent of memory is used, with 13148[3] entries on average, of which $54 * 256$ are single-byte entries. This means the lookup table is a highly sparse matrix, and the same story holds for the hash table.

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

3.3.1. Reducing hash table size

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

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

Overall, this changes the memory requirement from $1024 * 16$ to $1024 + 12 * X$ at the cost of an additional lookup, where X is the size of the secondary data table. This parameter balances the compression ratio and, indirectly, performance. We will investigate the effect of this parameter in Chapter 5.

3.3.2. Lookup table

We will now focus on the lookup table. As we already established in Section 3.3, this data structure is extremely sparse. There are 65536 possible entries, but only 13148 entries are used on average. Note that all symbols that consist of a single character use 256 entries in the lookup table.

Ignoring single-byte entries and using a separate table can save 50 percent of the total memory footprint. Every entry consists of two bytes: the code and the symbol length. We no longer need to store the length because this can be deduced from context; all two-byte symbols are in the lookup table, and one-byte symbols are in the new table.

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

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

Table 3.1 shows the usage of the shortcodes data structure for the three datasets, in terms of the metrics described above. Note that we only look at symbols with a length of two characters. We can see that while the overall matrix is very sparse, the actual data is relatively dense. The number of rows is relatively small compared to the potential number of rows, which makes sense considering most characters are not used in purely textual data. Furthermore, we can see that the (average) number of symbols that start with the same character, so the average number of entries (columns) in the same row, is relatively low.

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

Table 3.1: The usage of the lookup table in terms of row and column usage. A row is used when there is a 2-byte symbol starting with the character corresponding to the row. The number of columns described in this table refers to the columns used within the same row; in other words, the number of 2-byte symbols that start with the same character.

One data structure that could more efficiently represent this data pattern is an ELL matrix based on the sparse matrix package in ELLPACK [35]. The original matrix can be changed to a NxK matrix, where K is the new number of columns, and all non-zero elements within a row are compacted. An additional matrix of identical size is then used to map the original column locations to new column

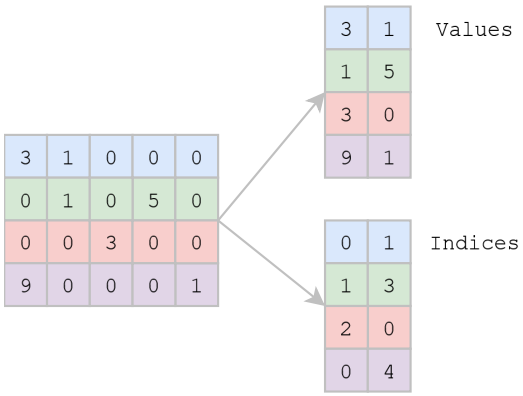


Figure 3.4: The ELL sparse matrix format splits up a matrix into a (smaller) value matrix and a column indices matrix that is identical in size to the value matrix. In the original encoding matrix, a code for symbol AB can be found in row A and column B. In the ELL format, a code can be found by iterating over row A in the indices matrix to find the column that contains the value B. The code is the value in the value matrix at the same row and column if found.

locations. An example illustration can be found in Figure 3.4. This representation changes the memory footprint from $256 * 256$ to $256 * K * 2$, saving space when $K < \frac{N}{2}$. As is the case with the parameter X for the hash table, the parameter K becomes a parameter to balance the ratio and memory footprint. We will find its effect on the compression ratio in Chapter 5.

While the ELL format leads to a significant reduction in size, the matrix is still sparse, storing more than 200 empty rows. Additionally, a GPU uses 32 banks to address shared memory, meaning a single bank will serve eight rows of this matrix, likely leading to bank conflicts as the characters used in textual data are in close proximity.

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

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

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

Note that the parameters R and K directly map to the row and column usage described above, and will influence the final compression ratio and, indirectly, performance. We have slightly modified the original FSST table generation algorithm to respect the additional constraints defined by these parameters and pick the next best option if a constraint would be violated. We will investigate the effects of these parameters in Chapter 5.

3.3.3. Sliding table for collaborative lookups

Up until now, we have focused on lookup tables that work with a tiling approach where every thread encodes its own chunk of data. However, as we already briefly discussed in Section 3.2.1, we can also parallelize the lookup sub-task. This lookup table focuses on that approach.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

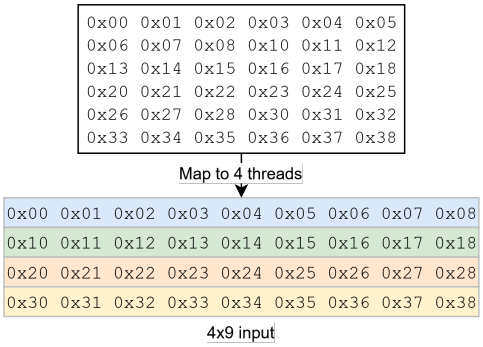


Figure 3.5: The NxM input matrix represents the input data for N threads, where every row contains M bytes each thread will process. This mapping is trivial when the total amount of data is a multiple of M. Otherwise, the input data must be padded to reach the next multiple.

We introduce a new table, called a sliding table. The main idea is that we store symbols in descending order of their length, and then progressively compare all entries. There are 256 possible entries, which results in eight lookup iterations with 32 comparisons per iteration. The symbol table and the lookup process are shown in Listing 2. Note that these lookups will be free of bank conflicts, since the size of every entry in memory is twelve bytes. This results in a bank offset of three per entry, which perfectly wraps around 32 to use all banks exactly once.

Every thread identifies the next symbol based on the current lookup iteration and its ID, after which it will perform an equality check. All threads will then hold a vote to determine which thread has the best match, if any. This is achieved using the ballot functionality. We can determine the best match by simply choosing the thread with a match that has the lowest ID, since the symbols are sorted based on length.

The main advantage of this approach is that we eliminate thread divergence and bank conflicts, and use minimal memory. However, we could reduce the achieved parallelism too much, which means this approach would be fundamentally unfit for encoding. In Section 3.4.3, we will compare the global approaches and determine if this table has any use in the final design.

3.4. Towards a GPU implementation

This section will first elaborate on the data flow through our compression pipeline. We will then discuss possible encoding kernels and their effects on the pipeline. Finally, we will show some preliminary results that we will use to select the most promising kernel that we will optimize further.

3.4.1. Data flow through compression pipeline

Initially, the data to be compressed lives in contiguous memory on the GPU's global memory. The input data can be viewed as an NxM matrix, where N is the number of tiles or threads, and M is the number of bytes every thread will process. In other words, the matrix contains a row for every thread, and every row consists of all the bytes the thread will process. The input data format can be seen in Figure 3.5.

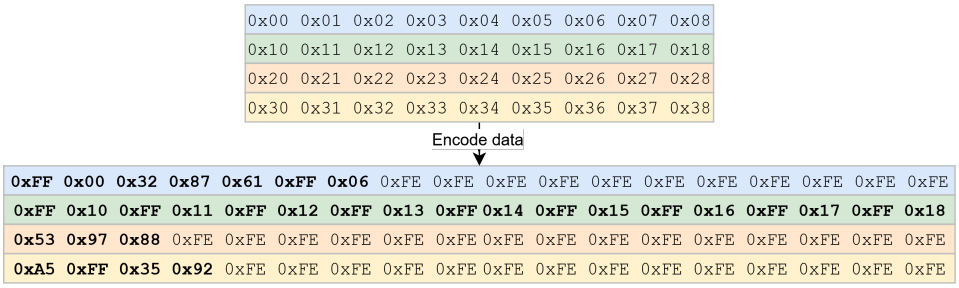
This data then needs to be compressed and written into a compressed buffer. Since we run the table generation on the CPU, we first need to move a sample of the data from the GPU global memory to system memory. We then generate the symbol tables in a multi-threaded fashion, generating *super tables* and the corresponding optimized data structures.

Once the encoding tables are generated, we move the tables from system memory to a GPU buffer. We can then run an encoding kernel that transforms the NxM input matrix into a $Nx2M$ matrix. In the output matrix, every row contains the compressed row of the input data. The output data has twice as many columns because the worst-case result of FSST is a 2x increase in size, when every symbol would be escaped. All unused entries in a row will be filled with a reserved padding character, which will be filtered out in a later phase.

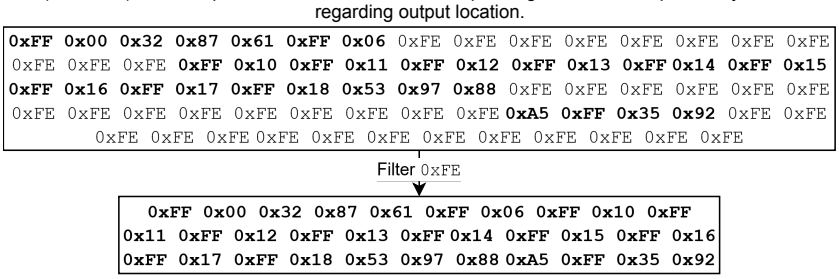
Finally, all padding has to be removed from the output data. The implementation details depend heavily on the encoding kernel's output format, but for the sake of this overview, this step can be viewed as a generic stream filter. The overall pipeline is illustrated in Figure 3.6.

Listing 2 All data structures used in the sliding symbol table, and the lookup process using a voting mechanism and intra-warp data transfers.

```
struct ComparableSmallSymbol {
    uint32_t val1 = 0, val2 = 0; // lsb, msb
    uint16_t metadata = 0; // ignoredBytes:3,code:8,length:4
    // Metadata/helper functions omitted for brevity...
    bool match(Symbol s) {
        uint64_t relevant_val = s.val.num & (0xFFFFFFFFFFFFFFFF >> ignoredBytes() * 8);
        return ((uint64_t)val1 | (uint64_t)val2 << 32) == relevant_val;;
    }
};

struct SymbolSlidingTableData {
    ComparableSmallSymbol symbols[256];
    // Metadata/helper functions omitted for brevity...

    bool attemptMatch(const GPUSymbol& sym, int iter, uint8_t* code, uint8_t* len) {
        const ComparableSmallSymbol s = symbols[32 * iter + threadIdx.x % 32];
        *code = s.code();
        *len = s.length();
        return s.match(sym);
    }
};

BallotResult ballot_cycle(SymbolSlidingTableData& symbol_table, Symbol symbol) {
    uint8_t code = 255, len = 1, lane_id = threadIdx.x % 32;

    for (int i = 0; i < 256 / 32; i++) {
        // Check if this symbol matches our current guess
        const bool match = symbol_table.attemptMatch(symbol, i, &code, &len);

        // Do a ballot to see if any thread found a match
        const uint32_t mask = __ballot_sync(0xFFFFFFFF, match);

        if (mask == 0) {
            continue; // If nobody found a match, attempt next cycle
        }

        // If this thread has the best match, it will have to do the output processing
        const uint8_t best_match_lane = __ffs(mask) - 1;
        uint8_t s_len = len;
        if (lane_id == best_match_lane) {
            // Omitted for brevity: handle write..
        }

        // Synchronize input and output offsets between threads
        s_len = __shfl_sync(0xFFFFFFFF, s_len, best_match_lane);
        return BallotResult{.sym_len = s_len,.output_size = 1};
    }

    if (lane_id == 0) { // Nobody found a match, need to escape..
        // Omitted for brevity: handle write..
    }
    return BallotResult{.sym_len = 1,.output_size = 2};
}
```



(a) The encoding step encodes the NxM to a $Nx2M$ result using the encoding table. The data size within a row may increase to twice the original size due to escapes, and the remaining space is filled with padding data (`0XFE`). The bold (non-`0xFE`) values represent codes. Because of the padding, there is no dependency between blocks regarding output location.



(b) In the final step, all padding is removed, resulting in all codes being concatenated. This filter step reduces the size of the data, achieving compression.

Figure 3.6: The basic pipeline consists of three steps. First, the encoding tables are generated using the FSST algorithm. The next step is to encode the input data. This will increase the size of the data, but most of it will be padding. The final step filters the data stream, which achieves compression and concatenates all block data.

3.4.2. Types of encoding kernels

In general, we introduce three types of encoding kernels. There are different variations within versions, but they are based on the same technique.

First, we have a compaction-based encoding kernel. The main idea behind this kernel is that every thread block has its own output data block that is guaranteed to fit the compressed data. This way, blocks do not write to overlapping locations, and we can concatenate the data in a separate post-processing phase. Threads use a reserved padding symbol to indicate unused memory, and the post-processing phase will filter these padding symbols. In this kernel, every thread maps to its own tile, so threads work on their own data. The parallelism in this kernel is essentially a form of tiling, similar to GSST splits. Some disadvantages of this kernel type are high thread divergence, bank conflicts for hash lookups, and memory operations that will be difficult to coalesce.

Second, we have a collaborative lookup kernel. In this kernel, the threads within a warp encode the same piece of data, focusing on parallelizing the lookup procedure. This is achieved using the sliding table mentioned in Section 3.3.3. This kernel type eliminates thread divergence and bank conflicts and allows trivial memory coalescing for both reads and writes. We must still perform stream filtering in a post-processing phase, similar to the compaction kernel.

Finally, we can use the same encoding process and the compaction kernel, but precalculate all output locations. This means all padding is no longer needed, and we can eliminate the stream filtering. However, this comes at the cost of running the encoding kernel twice, but only writing to memory the second time. Note that this pipeline does not match the pipeline described in Figure 3.6 as no padding is added or filtered, but instead, the blocks are directly written to their precalculated locations. Furthermore, it will still be challenging to coalesce memory operations.

3.4.3. Preliminary encoding results

In the previous sections, we have introduced several types of kernels and corresponding lookup data structures. In theory, we can implement and optimize all of them, but that will lead to a large design space. To limit our design space, we will perform some preliminary tests. These tests involve implementing a basic implementation for all lookup data structures and encoding types, and then benchmarking them.

Version	Table gen (GB/s)	Precomputation (GB/s)	Encoding (GB/s)
List	2.186	0.218	0.998
Count	22.721	4.565	8.203

Table 3.2: The results of the preliminary benchmarks comparing different versions. This test was performed on a development system (RTX 2070, Ryzen 7 5800X) and used 2GB of TPC-H data.

For our initial benchmark, we use a development system that has an RTX 2070 8GB and a Ryzen 7 5800X. We implemented the collaborative lookup kernel, which uses the sliding window introduced in Section 3.6.1. In addition to that, we implemented the encoding kernel that uses the precalculated output locations. The results can be found in Table 3.2. Note that the count implementation uses the multithreaded table generation, while the list implementation does not.

We can see that the collaborative kernel underperforms significantly when compared to the other kernel, even when compared to the single-threaded FSST implementation. This eliminates a sliding-window-based solution.

To determine the best option between using a compaction-based kernel or one that uses precalculated positions, we also perform a small benchmark on the Thrust library. We generate random data and filter out 64 percent of the data, which corresponds to a ratio of about 2.75, using stream compaction. On our development system, we achieve 96.7 GB/s and 65.6 GB/s for the `copy_if` and `remove_if` methods, respectively. The latter function is performed in-place, while the former is not.

This shows that our precalculation kernel would have to match 96.7 GB/s of throughput in order for it to be a better solution than a compaction kernel, assuming the compaction kernel would output the same amount of data as its input. While not impossible, it is likely stream compaction is more performant than encoding due to its inherent issues with thread divergence. For that reason, a compaction-based kernel is the best approach, and we will limit ourselves to a compaction kernel from this point.

3.5. Version summary

We have introduced the general dataflow of our compression pipeline and identified the most promising kernel type in Section 3.4.

In the next sections, we will introduce our optimized design for an accelerated FSST compression pipeline based on a compaction kernel, and in Chapter 5 we will benchmark these versions. In this section, we aim to summarize the most important work of the design and the difference between versions.

To fit the encoding table into shared memory, we introduced two concepts. The first one is heavily based on the ELLPACK format and significantly improves memory usage. The novel second format uses 32-bit integers to create a matching table. This encoding table format is even more compact, possibly at the cost of some compression ratio. The matching table is used by default, but all pipelines can switch to the other with no implementation changes.

We will now summarize the different pipeline versions, which can also be found in Table 3.3. At the third iteration, the versioning diverges into a transposed (with `-T`) and non-transposed (without `-T`) version. This is because the transposed version significantly changes the inner workings of the encoding pipelines. It would be confusing to keep the same versioning, as some later improvements only work for the transposed version, not the non-transposed version, and vice versa.

Version 0 is the initial version that works on the GPU. There were several different proofs of concept, but this version was picked as a baseline. Version 1, introduced in Section 3.6.1 and 3.6.2, improves this pipeline by introducing output packing and the sliding window, which means the kernel directly operates on 32-bit registers instead of many smaller unaligned 8-bit values.

Until now, the changes have only related to the encoding kernel itself. That changes with versions 2 and 3. Version 2, introduced in Section 3.6.3, introduces a transposition of the input data while the encoding tables are generated, which means the encoding kernel can perform coalesced reads. Version 3, introduced in Section 3.7.1, extends this idea and transposes the output data, meaning the encoding kernel can perform coalesced writes. Since transposition stages can be done very efficiently, the gains from coalesced writes are higher than the cost of the transposition stages.

Version 4, introduced in Section 3.7.4, extends the previous version by adding transposition pipelining using dynamic parallelism.

The `-T` branch starts with V3T, which is based on V2 and introduced in Section 3.6.4. This version introduces a different concept for handling output, using ballots and additional filler values to create

a transposed output matrix. This allows for coalesced writes, resulting in lower memory usage and higher throughput, at the cost of higher complexity and a lower compression ratio.

Version 4T, introduced in Section 3.7.2, uses the dense output format of V3T and replaces stream compaction with many direct memory copies, resulting in even higher compression throughput.

Version 5T, introduced in Section 3.7.3, expands on this by performing an additional transposition stage before the memory copy. This allows us to perform a final stream compaction pass to filter out the V3T filler data. The final version achieves the same compression ratio as V4, while almost achieving the same throughput as V4T.

Version	Introduced in Section	Summary
V0	3.6	Base version.
V1	3.6.2	Introduces the use of sliding window and output packing.
V2	3.6.3	Transposes the input data during table generation.
V3	3.7.1	Encoding kernel outputs transposed data. An additional efficient transpose stage transforms the data to the correct structure before compaction.
V4	3.7.4	Adds pipelining to the efficient transpose stage using dynamic parallelism.
V3T	3.6.4	Uses a voting mechanism to output dense and transposed data, with some additional filler data.
V4T	3.7.2	Replaces stream compaction with direct memory copies to remove inter-block padding.
V5T	3.7.3	Uses a pipelined transpose kernel before memory copy, with an additional stream compaction stage that removes interleaved padding.

Table 3.3: Summary of compression pipeline versions and their key modifications.

3.6. Optimizing compaction kernel

This section will elaborate more on the compaction kernel and several optimizations.

We know that the input format of the encoding kernel is an NxM matrix, where a row is the input data of a thread. Since threads progress at different speeds through their column, we must load the row into shared memory to ensure coalesced reads. This does not fit into memory at once, so we emulate a circular buffer and load smaller *chunks* of the tile. We then continue performing encoding cycles until the full tile has been encoded.

An encoding cycle consists of creating a symbol with the first eight bytes in the buffer and then using that symbol to do a lookup in our encoding table. We use the best match to write an output code with a possible escape character and advance the input circular buffer corresponding to the length of the best match. Once fewer than eight bytes are left, the cycle is completed, and the output buffer is written away to memory, ensuring all unused entries are set to the padding symbol. In the final cycle, we also encode the last eight bytes. We refer to this version of the compaction kernel as V0.

3.6.1. Alignment and sliding window

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

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

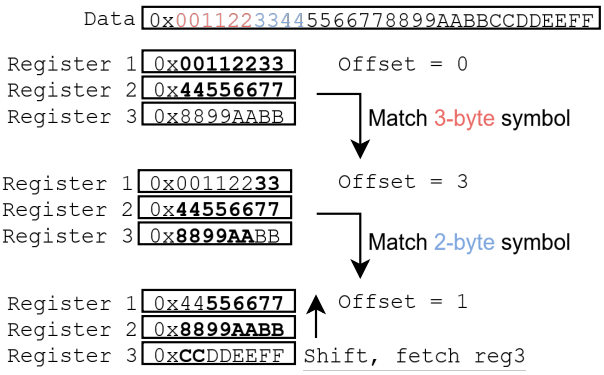


Figure 3.7: The process of using a sliding window to build a view of the active data, which the encoding kernel can use to match on directly. In this example, we show how data moves through the registers as the data in shared memory is processed. Bold numbers are used to show what part of the data is part of the current view.

evaluate multiple (partial) words, we introduce the sliding window.

The sliding window uses three 32-bit registers and keeps track of the reading offset to create a view of the next eight bytes. The effect of the sliding window can be seen in Figure 3.7. In Listing 3, we show how to create a view. We also keep track of the spillover from the previous encoding cycle, as a symbol might overlap chunk borders. When a register is fully encoded, indicated by the offset, we shift the registers once and fetch the next 32-bit word. We also keep track of the spillover from the previous encoding cycle, as a symbol might overlap chunk borders. The spillover is used to create a view when there is still data available. When the spillover has been fully used, i.e., all data from the previous cycle has been encoded, we switch to using the three registers.

3.6.2. Output packing (V1)

In the previous section, we addressed the issue of inefficient shared memory usage and misaligned data access for input data. However, this issue is also present in the handling of output data. Every match iteration of an encoding cycle produces one or two bytes, depending on whether the symbol needs an escape character. This is not naturally aligned to 4-byte boundaries, so an array of bytes is used to allow for this individual assignment.

Writing this array to global memory is incredibly wasteful since every write will lead to a 128-byte transaction. A simple solution would be to transform the byte array to an array of a larger datatype. We would also need to store this array in intermediate shared memory instead of directly writing to global memory. If we choose this type to be a 32-bit integer, we could avoid all bank conflicts by structuring the output data for a thread block in a column-major fashion.

However, storing 32-bit integers means we lose the ability to assign individual bytes easily. For that reason, we need to perform an output packing operation. This process uses some bit logic to set individual bytes in a 32-bit number, which allows us to use the array of 32-bit numbers as if it were an array of 8-bit numbers. Listing 4 shows the output packing process.

Using output packing and the sliding window described in Section 3.6.1, we can now use the original algorithm in V0 but with bigger data types and aligned words. This version of the encoding kernel is called V1, and can be found in Listing 6. The encoding cycle pseudocode can be found in Listing 5.

3.6.3. Coalesced reads (V2)

One important aspect to realize is that the input data has a *row affinity*. In other words, the natural ordering of the data is row-major, which is not ideal for a GPU. Memory loads must be *coalesced* to utilize the full bandwidth. One option would be to use an approach similar to CudaDMA [11], where warps would collaboratively load data, effectively decoupling the compute and memory warps. In our case, this solution would require too much shared memory, as data for all threads would have to be loaded before any thread could progress. Instead, we will interleave the data using a transposition stage in parallel to table generation. This means we change the NxM input matrix to an MxN matrix, which allows us to coalesce memory loads in the encoding kernel. We call this version of the encoding

Listing 3 Sliding window view creation using three registers and an offset, and also a variant that keeps track of the data that spilled over from the previous encoding cycle.

```
uint64_t create_view(uint32_t first_block, uint32_t second_block,
                uint32_t third_block, uint8_t offset, uint8_t len) {
    uint8_t b_from_first = min(len, 4 - offset);
    uint8_t b_from_second = min(len - b_from_first, 4);
    uint8_t b_from_third = min(len-(b_from_first+b_from_second), offset);

    uint64_t first_data = get_first_n(
                            first_block >> offset * 8, b_from_first);
    uint64_t second_data = get_first_n(second_block, b_from_second);
    uint64_t third_data = get_first_n(third_block, b_from_third);

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

uint64_t create_view_spill(uint64_t spill, uint8_t spill_len,
            uint32_t first_block, uint32_t second_block,
            uint8_t len) {
    uint8_t b_from_spill = min(spill_len, len);
    uint8_t b_from_first = min(len - b_from_spill, 4);
    uint8_t b_from_second = min(len-(b_from_spill+b_from_first), 4);

    uint64_t spill_data = get_first_n(spillover, b_from_spill);
    uint64_t first_data = get_first_n(first_block, b_from_first);
    uint64_t second_data = get_first_n(second_block, b_from_second);

    return spill_data | first_data << b_from_spill * 8 |
        second_data << (b_from_spill + b_from_first) * 8;
}
```

Listing 4 The output packing process

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

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

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

Listing 5 Encoding cycle for one chunk in V1.

```
EncodeResult compaction_encode(SymbolMatchTable symbol_table,
                           uint32_t result[out_buf_size][THREAD_COUNT],
                           uint32_t load[tile_buf_size][THREAD_COUNT],
                           uint64_t spillover, uint8_t spillover_len,
                           bool last_chunk) {
    uint8_t out = 0; // Keep track of the number of written bytes
    uint8_t idx = 0; // Keep track of the number of read bytes

    uint32_t first_block = load[0][threadIdx.x]; // Sliding window reg1
    uint32_t second_block = load[1][threadIdx.x]; // Sliding window reg2
    uint32_t third_block = load[2][threadIdx.x]; // Sliding window reg3
    uint8_t first_block_offset = 0; // Sliding window offset
    uint8_t block_offset = n_regs_per_chunk - 1;

    uint16_t search_len =
        n_regs_per_chunk * sizeof(uint32_t); // can look ahead at all available bytes
    uint16_t encode_len = search_len - 7;
    // cannot encode when there are fewer than 8 bytes available, unless in the last chunk

    // First handle spillover
    while (idx < spillover_len) {
        uint64_t symbol = create_view_spill(spillover, spillover_len - idx, first_block,
                                            second_block, 8);
        uint16_t code = symbol_table.findLongestSymbol(symbol);
        uint8_t sym = (uint8_t)code;
        uint8_t sym_len = (uint8_t)(code >> 8);
        uint8_t escape = sym == 255;

        pack_results(result, out, sym);
        if (escape)
            pack_results_local(result, out + 1, get_first_byte(symbol));

        // Update pointers
        out += 1 + escape;
        idx += sym_len;

        // Bookkeeping of spillover data
        spillover >>= 8 * sym_len;
    }

    // Update registers for possible shift after overusage of spillover
    shift_registers(load, idx - spillover_len, &first_block_offset, &block_offset,
                &first_block, &second_block, &third_block);

    // Then handle regular block
    const uint8_t encode_range = last_chunk ? search_len : encode_len;
    while (idx < spillover_len + encode_range) {
        uint64_t symbol = create_view(first_block, second_block, third_block,
                           first_block_offset, min(8, search_len + spillover_len - idx));
        uint16_t code = symbol_table.findLongestSymbol(symbol);
        uint8_t sym = (uint8_t)code;
        uint8_t sym_len = (uint8_t)(code >> 8);
        uint8_t escape = sym == 255;

        pack_results_local(result, out, sym);
        if (escape)
            pack_results_local(result, out + 1, get_first_byte(symbol));

        // Update pointers
        out += 1 + escape;
        idx += sym_len;

        shift_registers(load, sym_len, &first_block_offset, &block_offset, &first_block,
                    &second_block, &third_block);
    }

    // Then create a new spillover
    uint8_t spilled_bytes = search_len + spillover_len - idx;
    uint64_t new_spillover = create_view_spill(first_block, second_block, third_block,
                                           first_block_offset, spilled_bytes);

    return EncodeResult{out, new_spillover, spilled_bytes};
}
```

Listing 6 Basic V1 encoding kernel. The encoding function used is listed in Listing 5.

```
template <typename T>
    requires(alignof(T) == alignof(uint32_t))
void load_metadata_local(T* metadata, uint32_t* smem_target, uint32_t super_block_size) {
    const uint32_t* m = (uint32_t*)&metadata[blockIdx.x / super_block_size];

    for (uint i = threadIdx.x; i < sizeof(T) / sizeof(uint32_t); i += blockDim.x) {
        smem_target[i] = m[i];
    }

    // Symbol table needs to be in shared memory before we can actually start with encoding
    __syncthreads();
}

void gpu_encode_v1(GCompactionMetadata* metadata. const uint8_t* src, uint8_t* dst) {
    __shared__ uint32_t global_size[THREAD_COUNT]; // Keep track of thread outputs
    __shared__ uint32_t input[tile_buf_size][THREAD_COUNT];
    __shared__ uint32_t result[out_buf_size][THREAD_COUNT];

    // Load metadata into shared memory
    __shared__ GCompactionMetadata m;
    load_metadata_local<GCompactionMetadata>(metadata, (uint32_t*)&m, SUPER_BLOCK_SIZE);

    // Active data
    uint64_t spillover = 0;
    uint8_t spillover_len = 0;
    global_size[threadIdx.x] = 0;

    const auto aligned_src = (uint64_t*)(src + BLOCK_SIZE * ((uint64_t)blockIdx.x));

    for (uint32_t chunk_id = 0; chunk_id < n_chunks; chunk_id++) {
        // Step 1: Load into working memory
        uint64_t load1 = aligned_src[n_words_per_tile * threadIdx.x + chunk_id * 2 + 0];
        uint64_t load2 = aligned_src[n_words_per_tile * threadIdx.x + chunk_id * 2 + 1];
        input[0][threadIdx.x] = (uint32_t)load1;
        input[1][threadIdx.x] = (uint32_t)(load1 >> 32);
        input[2][threadIdx.x] = (uint32_t)load2;
        input[3][threadIdx.x] = (uint32_t)(load2 >> 32);

        // Step 2: Run chunked compression on spillover and loaded window
        auto encode_result = compaction_encode(m.symbol_table, result, input, spillover,
                                               spillover_len, chunk_id == n_chunks - 1);

        // Step 3: Update spillover
        spillover = encode_result.spillover;
        spillover_len = encode_result.spillover_len;
        global_size[threadIdx.x] += encode_result.bytes_written;

        // Step 4: Output
        uint8_t* dst_loc = dst + blockIdx.x * (uint64_t)TMP_OUT_BLOCK_SIZE;
        uint32_t* dst_aligned = (uint32_t*)dst_loc;

        for (uint32_t store_id = 0; store_id < out_buf_size; store_id++) {

        dst_aligned[threadIdx.x * tile_out_len_words + chunk_id * out_buf_size + store_id]
            = result[store_id][threadIdx.x];
        }
    }

    // Left out for brevity: output header writing..
}
```

kernel V2. Note that we call it a new version of the encoding kernel, but we also make changes to the overall pipeline.

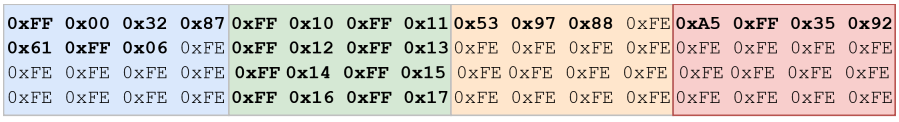
3.6.4. Coalesced output (V3T)

One problem that has not been addressed so far is that all output writes are not coalesced. Similarly to the input format, the output data also has a *row* affinity because every thread generates output for a single tile. This is even more important in the case of writes because we create more data than we ingest; every thread writes twice the amount of data it reads because FSST can *potentially* double the amount of data.

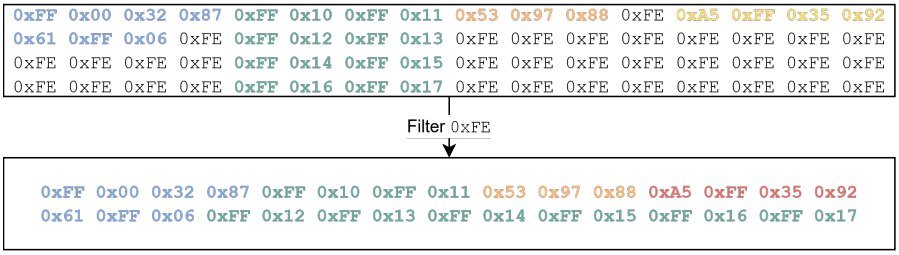
One solution would be to write data collaboratively. In this case, you would group threads within a warp, depending on the amount of output data per thread, and threads would work together to write the data in global memory for every thread in the group. For example, assume every thread has 32 integers to write to global memory. All threads would work together to write the integers from the first thread, all writing a single integer. They would then continue to the 32 integers that belong to the second thread and repeat that process until they have written the integers from the last thread.

Intuitively, that solution coalesces all writes and, therefore, would be faster. While all writes are indeed coalesced, the kernel is not quicker. This is because one critical aspect is that the shared memory is organized in a column-major fashion, such that all shared memory banks are servicing exactly one thread. This eliminates all bank conflicts during the encoding cycles, but would lead to massive bank conflicts when writing data using the collaborative method. All threads would use a single bank, leading to sequential reads for the entire group. This entirely undermines all performance gains from coalesced writes.

Another solution would be to transpose the output data. This means the output data can be seen as a YxN matrix with 32-bit words, where N is the number of threads within a thread block and Y is the number of output words per thread. The decompressor would then need to reconstruct the original output data, which is not a problem as long as the output matrix remains valid. A valid matrix means that all rows are of the same length. Otherwise, creating a 2D structure (matrix) from a 1D memory buffer would be impossible. However, our output matrix is no longer valid after the stream compaction phase because of all the intermediate padding. This is illustrated in Figure 3.8.



(a) The output data of transposing the output of the V2 pipeline would look like a transposed matrix of the output data of Figure 3.6b



(b) When taking the transposed output matrix and applying the stream filter, you get the above invalid matrix. This matrix is invalid because it is infeasible to determine which values belong to which tile (depicted by different colors here). The only way would be to store the amount of data per row per tile, leading to one additional byte for every four output bytes.

Figure 3.8: An illustration of the effects of simply using a transposed output and applying the stream filter. The problem is that the original matrix cannot be reconstructed from the resulting data without adding too much metadata.

To coalesce the output writes using a transposed format without invalidating our output matrix, we need to make significant changes to the handling of the output. In V2, we output data for every cycle, even when there is only padding. This means that our padding will be intertwined with data. To achieve a dense output, i.e., all padding is at the end of a block, we must only write data when there is enough useful data.

We achieve this using a significant change to our encoding kernel: collaborative output writing. In

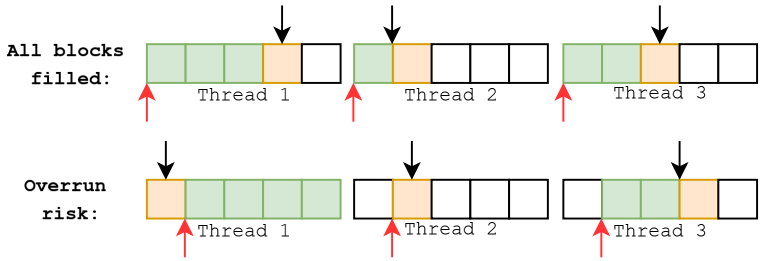


Figure 3.9: Threads keep track of their own local buffer head (marked with black arrows), and their working word (marked orange) and filled blocks (marked green). All threads keep track of the active word in the warp (marked with red arrows). Threads in a warp will decide to flush in two scenarios: when all threads have filled the currently active word with data, or when a thread can potentially overrun the buffer in the next encoding iteration.

Listing 7 The encoding kernel cycle is adjusted to call the voting function after every cycle, such that threads in a warp can determine whether they need to flush or not. This function creates the functionality shown in Figure 3.9.

```
bool must_flush(uint8_t active_out_block, uint8_t current_out) {
    uint8_t current_out_block = current_out / sizeof(uint32_t);

    // If the current output block is not the same as the active block, we can
    // assume we are ahead (as being behind is impossible/illegal)
    bool current_is_ahead = active_out_block != current_out_block;

    bool last_free_block = (current_out_block + 1) % tile_out_word_buf_size == active_out_block;
    bool escape_can_overflow = current_out % sizeof(uint32_t) >= 2;
    bool risk_overflow = last_free_block && escape_can_overflow;

    bool any_thread_risk_overflow = __popc(__ballot_sync(0xFFFFFFFF, risk_overflow)) > 0;
    bool all_thread_ahead = __popc(__ballot_sync(0xFFFFFFFF, current_is_ahead)) == 32;

    return any_thread_risk_overflow || all_thread_ahead;
}
```

order to achieve coalesced writes, we will use the aforementioned transposed output format, and all threads within a warp have to perform writes in the same row at the same time, hence the *collaborative* part.

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

After every iteration in the encoding cycle, threads will hold a vote on whether to initiate a flush or not. If any thread risks overrunning its buffer, all threads will add padding to their local buffer if needed and trigger a flush. A flush will also be triggered if all threads have filled the currently active word, which is the ideal scenario. This process is illustrated in Figure 3.9. After the last encoding cycle has completed, a warp will continue flushing its buffers until all threads within a warp have fully written their data. Additionally, all warps within a block will communicate such that they write the same number of overall flushes to create a valid output matrix. Listing 7 shows the general voting algorithm.

This method ensures all write transactions are coalesced and also eliminates the sequential inter-thread dependency. Imbalances between threads in terms of local compression ratios are mitigated by the voting process.

Using the above voting mechanism, we achieve an entirely different output format that is more dense than that of V2. In V2, the output data is in a row-major format with intermediate padding, whereas this format is column-major, and all padding is at the end of the memory block. This format also uses a new type of padding, which is not removed by stream compaction. This means we achieve coalesced writes, write less data overall, and the output matrix remains valid and can be decompressed at the

0xC8	0x33	0x68	0xFE	0xFE	0xFE	0xFE	0xFE	0x2D	0x7C	0x7F	0xD6	0x7C	0xD6	0xFE	0xFE
0x53	0xE9	0x23	0x48	0xFE	0xFE	0xFE	0xFE	0x55	0xC9	0x8E	0x84	0xC7	0x7B	0x93	0x63
0x2D	0x7C	0xD6	0x7C	0xD6	0xFE	0xFE	0xFE	0x53	0xE9	0x23	0x48	0xFE	0xFE	0xFE	0xFE
0xA0	0xCA	0x89	0x8C	0x70	0x7E	0xDA	0xB1	0x97	0x82	0x08	0x7C	0xA7	0xFE	0xFE	0xFE
0xC2	0x85	0x64	0xFE	0xFE	0xFE	0xFE	0xFE	0x46	0x7C	0x86	0x09	0xFE	0xFE	0xFE	0xFE
0x47	0x6B	0x83	0x63	0x7E	0xFE	0xFE	0xFE	0xCB	0x97	0x35	0x23	0x0C	0xC7	0xFE	0xFE

(a) The output data of V2 creates a $Nx2M$ matrix with every row containing the output of every thread. In this case, this results in a $3x32$ matrix. You can see the effect of the chunking, where every encoding cycle takes four bytes and outputs eight, often adding padding when there are fewer than eight bytes.

0xC8	0x33	0x68	0xFD	0x2D	0x7C	0xD6	0x7C	0xC2	0x85	0x64	0x46
0x2D	0x7C	0x7F	0xD6	0xD6	0x53	0xE9	0x23	0x7C	0x86	0x09	0xFD
0x7C	0xD6	0x53	0xE9	0x48	0xA0	0xCA	0x89	0x47	0x6B	0x83	0x63
0x23	0x48	0x55	0xC9	0x8C	0x70	0x7E	0xDA	0x7E	0xCB	0x97	0x35
0x8E	0x84	0xC7	0x7B	0xB1	0x97	0xFD	0xFD	0x23	0x0C	0xFD	0xFD
0x93	0x63	0xFD	0xFD	0x82	0x08	0x7C	0xA7	0xC7	0xFD	0xFD	0xFD
0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE
0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE	0xFE

(b) This shows the same output data, but then in the format of V3-T. This creates a $\frac{M}{2}x4N$ matrix where the codes are packed much more densely, but sometimes with an additional padding character (`0xFD`) when required when flushing. This allows a decompressor to reconstruct the data from a single thread by reading the corresponding columns and ignoring the special padding character.

Figure 3.10: In this example, we compare the output formats of V2 and V3-T for a simplified case of three threads, each outputting 32 bytes. The regular $Nx2M$ matrix is transformed into a $\frac{M}{2}x4N$, which allows for coalescing all encoder writes. Every flush writes a row of the new output matrix in contiguous memory.

cost of a reduction in compression ratio. The differences in output format are illustrated in Figure 3.10. We call this iteration of the encoding kernel V3T.

3.7. Optimizing overall pipeline

Until now, we have primarily focused on optimizing the encoding kernel, but we can still optimize the overall pipeline. In this section, we will focus on such optimizations.

3.7.1. Transposition stage (V3)

With V3-T, we introduced a dense transposed format. However, this format introduces some overhead, which results in a lower compression ratio. Furthermore, the GSST decompressor is less performant when using a transposed format.

We know that simply transposing the output of V2 speeds up the kernel, but it will result in an invalid matrix after string compaction. To fix this, we can add a transposition stage between encoding and compaction, transforming the data to a row-major format. While intuitively adding stages to transpose the data might feel like added overhead, the improved data access patterns lead to a better overall result. This is partly because memory bandwidth has increased enormously with modern GPUs, so the cost of the transpose before and after the kernel is relatively small compared to the performance gain in the processing kernel itself.

With the added transposition stage, we now have the V3 pipeline. We first transpose the input matrix while we concurrently generate the encoding tables. Once both stages are finished, we run the encoding kernel on the transposed input matrix. After encoding has been completed, we transpose the output matrix and then perform stream compaction.

3.7.2. Utilize dense output packing (V4T)

The dense output format introduced in V3T introduces further acceleration potential. In V3T, we still perform stream compaction to finalize the result. We use stream compaction provided by the Thrust library, specifically the `copy_if` functionality [78]. This kernel provides 360 GB/s of compaction throughput on an RTX 4090 and is already highly optimized.

The only real way to speed up the compaction stage is to run it on less data or use direct memory copies. Both options are not feasible for the non-T versions of our pipeline, but the dense format in V3T opens up the possibility of using direct memory copies instead of stream compaction. Instead of

performing stream compaction, we can copy only the first Y rows of the output matrix, corresponding to the number of flushes. The destination location of all blocks is determined by performing a prefix sum on the number of flushes for all blocks. This version is called V4T.

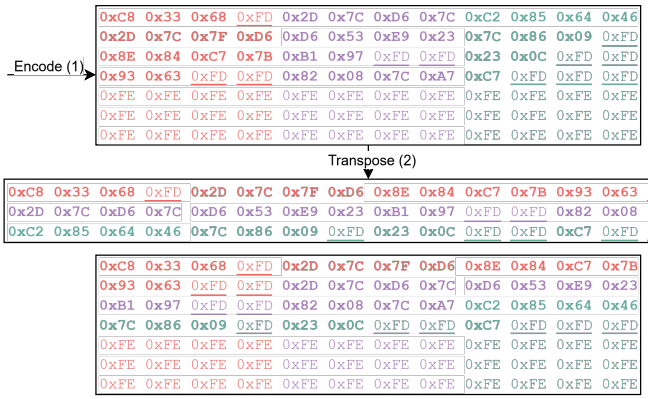
3.7.3. Optimizing for compression ratio (V5T)

One problem that remains with V4T is its lower compression ratio, because of the filler values used to maintain the output matrix. We can address this by applying the same transposition idea used in V3 to the output of V4T. Transposing the output of V4T results in the same output matrix as V3, where every row contains the data of a single tile. We can then apply stream compaction to filter out the filler values to achieve the same ratio as V3. Since V4T generates significantly less data than V3, the transposition and the compaction stage have very high effective throughput. This results in V5T, which achieves the same compression ratio as V3 at a slightly lower throughput than V4T.

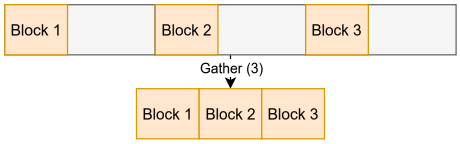
The entire pipeline of V5T is described in Figure 3.11. This figure shows the memory transformations we apply in the different stages of the pipeline. We start with the encoded data, which is the first step. We then transpose this data on a block level similar to V4. The main difference is that we only transpose *useful* data, i.e., data that has been written to by the flushing mechanism. This results in a higher effective throughput for the transposition stage, since we have less data to process. The transposed data for every block is then gathered, based on its output metadata, such that all blocks are in a single contiguous block of memory. We then perform stream compaction on this final continuous block of memory to remove padding that is the result of the balloting mechanism. Similarly to the transposition stage, the compaction stage is significantly sped up by ensuring that we only process useful data. This also means that we get a higher overall throughput when the compression ratio is high, unlike V4, which always performs all operations on the worst-case amount of data.

To minimize the required amount of memory to compress the data, we carefully use a temporary buffer and make use of the fact that we have multiple sequential memory transformations. Figure 3.12 shows how we use the temporary buffer with the memory transformations to swap data between buffers. We encode the input data to a temporary buffer, which we then transpose to the output buffer. Since the temporary buffer is now unused, we use it as the target buffer for our gathering stage. After the gather operation has completed, the output buffer is no longer used, so we directly perform our stream compaction from the temporary buffer to the output buffer. Additionally, we copy our generated headers to the output buffer during compaction. This ensures that we only need a single additional buffer to compress the data, which we will compare to the state-of-the-art compressors in Chapter 5.

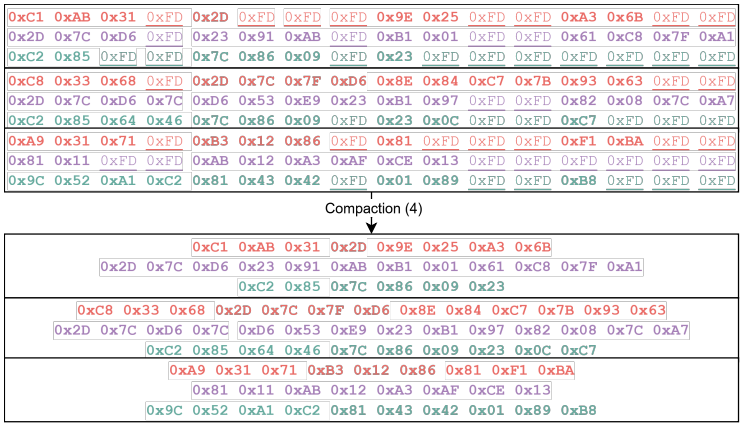
One interesting observation is that V5T is essentially V4T with additional post-processing stages. This means V5T could be *lazily* applied to V4T, i.e., V4T could be initially used to compress data and transmit it over a network, but V5T can then still be applied to data at rest to achieve a higher compression ratio.



(a) The first step, encoding, produces the output data that is typical for the `-T` branch. The V5T pipeline then immediately transposes this output data on the block level using dynamic parallelism. This results in output data more typical for the non-`T` branch, like V3 and V4. Note that we exactly know how much data was produced by tracking the number of flushes, so we only transpose the useful data. This means that the actual output data per block will look like the lower transposed block, while the *effective* output data will look like the upper transposed block.



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



(c) In the final stage, we remove the padding introduced by the voting mechanism. This phase is significantly faster because we now ensure that the compaction stage only has to work with useful data. The gather stage has already filtered all unnecessary padding introduced by the encoding stage. Note that this phase works on all blocks as a whole and does not distinguish between blocks. We kept the block indications in this image, but there aren't any different segments in memory.

Figure 3.11: A simplified overview of our GPU-accelerated compression pipeline. All data belonging to the same tile has the same color. Note that the first two stages of encoding and transposition operate on the block level, and the final two stages of gathering and compaction operate on the entire data stream.

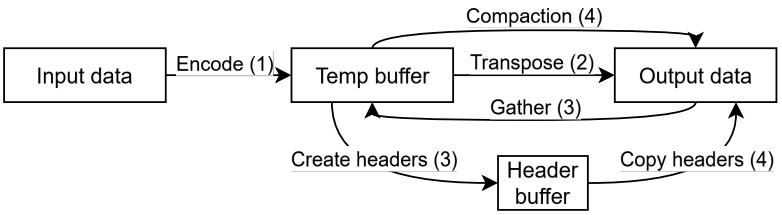
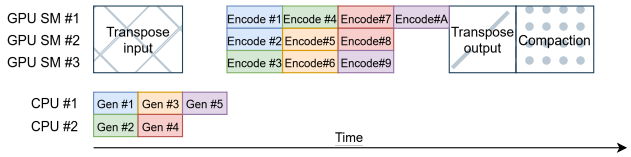


Figure 3.12: The data flow through our pipeline's temporary and destination buffers. The overall memory usage is low because we reuse the temporary and destination buffers for multiple operations.

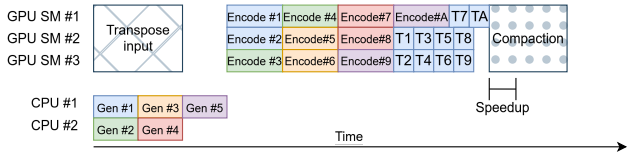
3.7.4. Pipelining (V4)

One technique we can still employ is pipelining. This means we overlap several operations to increase overall performance. Pipelining can improve a process when some resources are not utilized while all dependencies of a set of calculations have been satisfied.

A more concrete example of this happening in our pipeline is with the added transposition stage in V3. Whenever a block has been fully encoded, it can immediately be transposed. However, we only transpose the entire dataset once all blocks have been encoded. Following a scheme like the one illustrated in Figure 3.13 would be ideal.



(a) The V3 pipeline has apparent dependencies between the four stages, only starting on the next stage when the previous one has finished. The encoding only starts when all encoding tables have been generated, and transposing only starts when all blocks have been encoded. The SMs are underutilized at the tail of the encoding kernel because not enough blocks are left.



(b) The V4 pipeline can reduce this tail underutilization by changing the transpose output from a big shared matrix, which all threads use, to a small matrix per block. Every block then launches a transposition kernel using CUDA dynamic parallelism when it has finished encoding. The underutilized SMs can start running transposition kernels in the queue, resulting in an overall speedup.

Figure 3.13: Highlight of the difference between the V3 and V4 pipeline, with the V4 pipeline operating on a different granularity level than V3, allowing for more efficient pipelining. Note that this pipelining potential also exists for the dependency between table generation and encoding.

However, this is mainly a practical limitation because it is difficult for the CPU to know when a block has finished encoding. It only knows when the entire grid finishes. So it would have to make many small grids or add synchronization between the CPU and the GPU. To solve this, we can use dynamic parallelism. Using this CUDA functionality, we can launch grids from within the grid, meaning we can launch a transposing kernel from each block once it is finished encoding. We call this version of the pipeline V4.

Something that is out of scope for this thesis, but could be interesting when using this purely as a compression accelerator, is to use pipelining to hide the latency of loading the file from system memory to GPU memory. When using pipelining, you could already start to encode part of the file when the next part is still being transferred to device memory.

3.8. Data format

To decompress the data, we need to store some metadata that can be used to decompress the file. FSST uses a block format, where the corresponding data follows every block of metadata. However, as the paper's authors note, there is no fixed method of storing metadata as FSST is primarily meant for direct use in databases. This means the metadata would likely be stored elsewhere. The only reason FSST supports a file-based compression cycle is to allow direct comparison to competitors like LZ4.

The same applies to our compressor; we will support a file-based compression, but theoretically, the compressor can be modified to store metadata anywhere. The data format overview we use is shown in Figure 3.14. Since we use stream compaction on the entire data stream, as highlighted in Section 3.7.3, interleaving the metadata throughout the data would be inefficient. This is because we filter specific characters, which might occur in the metadata. For that reason, our metadata is at the start of the file, followed by all blocks of data.

The decompressor mainly dictates the data required in the metadata, but the metadata should contain the FSST decoding table at the very least. To validate the decompressed data, every block header contains its compressed and uncompressed lengths. This can also be used to parallelize decompression. The file header should at a minimum contain the number of blocks, the super table size, and the compressed and uncompressed data stream size.

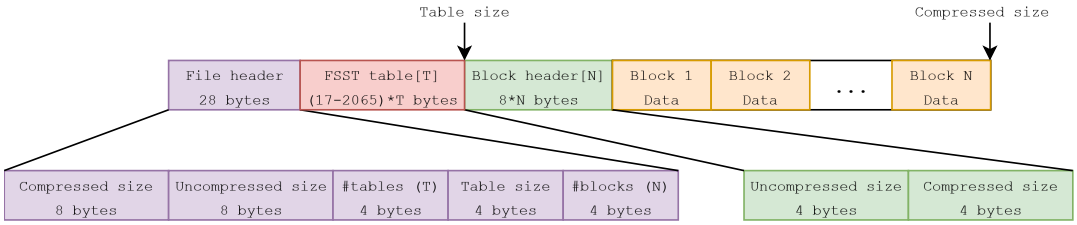


Figure 3.14: The data format used by the V5T compressor. All header data is at the start of memory, followed by all data blocks. The file header contains relevant data to reconstruct the other headers. The FSST table data consists of the decoding table and has a variable size, which is indicated in the file header. For every block, there is a block header with some data that is specific to the corresponding block.

This is enough data for our validation decompressor and any regular CPU decompressor. However, more information might be needed for a GPU decompressor like GSST. We will go into more depth on the required changes to this format and the integration with GSST in Chapter 4.

3.9. Optimizing for hardware

Up until now, we have primarily focused on kernel and pipeline optimizations. However, as shown in Section 2.5.2, the GPU hardware determines your theoretical maximum performance. While this is an obvious statement, we have yet to look at our target hardware. The aim of this thesis is not to develop a solution that is optimized for one hardware architecture, nor is the goal to develop a solution that works on all architectures that CUDA supports. However, we ought to at least investigate our strengths and weaknesses concerning the underlying hardware architectures and make recommendations for future research or making the compressor 'production-ready'.

One such observation is our heavy use of integer operations. Most of our optimizations rely on reducing the number of small (irregular) memory transactions by utilizing 32-bit integer registers for bitwise operations, as can be seen in Figure 3.15. This is not a problem, but GPUs usually focus on floating-point operations. This is why FLOPS is a critical performance metric, and high-performance libraries for linear algebra like cutlass focus on floating-point data types.

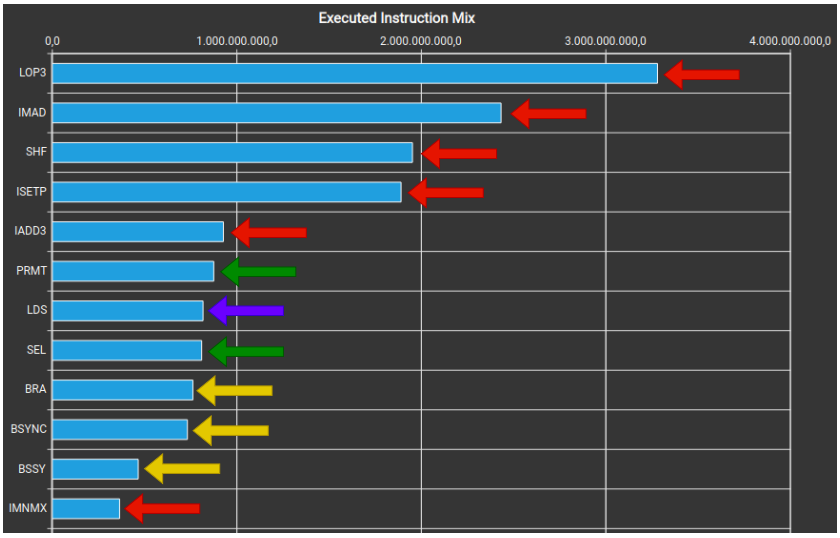


Figure 3.15: The executed instructions by our encoding kernel in the V5T pipeline. Red arrows show integer instructions, green arrows show movement instructions, purple arrows show load/store instructions, and yellow arrows show control instructions. Most operations are indeed integer instructions, which puts a high load on the ALU pipeline. Source: [73]

The reason why this is relevant becomes clear when investigating the underlying SM architecture of recent architectures like Ampere (A100, RTX 30xx), Ada Lovelace (H100, RTX 40xx), and Blackwell (B100, RTX 50xx). Before Blackwell, the SM architecture was optimized for regular shaders, meaning 50 percent of all CUDA cores within an SM would only perform FP32 operations, and the other half

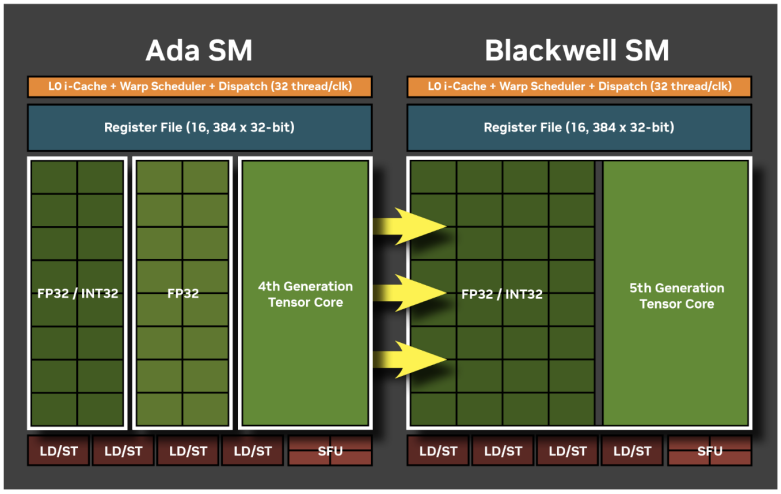


Figure 3.16: Comparing an Ada SM to a Blackwell SM shows the difference in architecture with regards to the number of INT32 capable cores. The Blackwell SM was optimized for neural shaders, which might also benefit INT32-heavy pipelines. Source: [76]

could switch between FP32 and INT32. This might become a bottleneck for a pipeline that heavily uses INT32 instructions like ours.

The introduction of the Blackwell architecture has changed this. Blackwell was optimized for neural shaders, which benefit from more INT32 throughput. For this reason, all CUDA cores in a Blackwell SM can execute INT32 instructions, doubling the number of possible INT32 integer operations compared to Ada. Figure 3.16 shows how the Blackwell SM architecture evolved from Ada's architecture by unifying them with FP32 cores.

If the pipeline is not bottlenecked by the number of available INT32 computations per cycle, this change in architecture will not result in significantly different performance numbers. However, it is essential to confirm this when benchmarking because it might affect adaptation on other architectures.

4

GSST integration

4.1. GSST analysis

In Section 2.4.4, we have already established what GSST is and briefly how it works. In this section, we will go into more depth about its implementation (Section 4.1.1) and its performance characteristics on our benchmark system (Section 4.1.2).

4.1.1. Implementation

We focus on the GSST splits kernel and ignore the other kernels. The reason for that is twofold: the split kernel is the most performant implementation out of all options, and it also matches our compressor output reasonably well.

Figure 4.1 shows the data format of the GSST decompressor. We can see that it more closely follows the FSST format, with interleaved headers and data blocks. The main addition is that the block headers now also contain the lengths of the individual splits, which match the output per thread. Furthermore, the number of splits per block was added, in addition to the uncompressed output length. This output length is used to determine where to place output data when decompressing. This breaks the sequential block location dependency.

The kernel structure is mostly what one would expect given the data format: every block is decompressed by a single thread block, and every thread decompressed a single split and calculates its input and output locations based on the header data and the fact that all splits, except the last one, have a constant uncompressed size. However, GSST also supports using the same thread for multiple splits and the same block for multiple blocks. This means that all thread blocks iterate over all block headers to create a running length of the data, and decompress blocks that belong to that block. This leads to some inefficiencies, but allows for greater flexibility. The original GSST authors found that using a single thread per split and a single thread block per data block results in the largest overall throughput indeed.

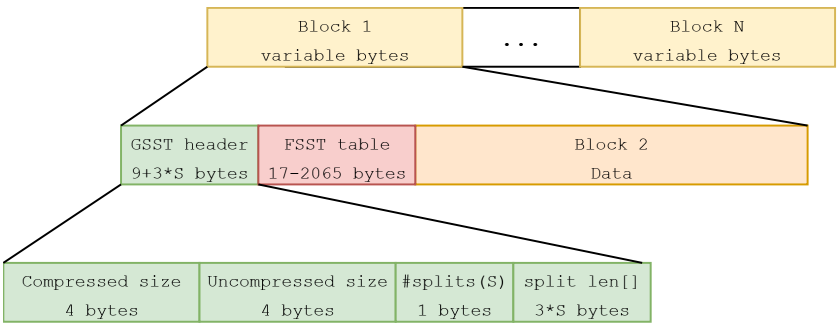


Figure 4.1: The data format used by the GSST decompressor. It is closer to the FSST data format, with only some information regarding the split structure added.

4.1.2. Performance

In the original GSST paper, a decompression throughput of 191 GB/s was reported on an A100. However, this is in the optimal configuration, and when combined with our compressor, there is a high chance we will not run in the optimal configuration for the decompressor. Ideally, we find three parameter configurations: one that favors decompression throughput, one that favors compression throughput, and one that favors overall throughput. This way, the used configuration can be changed depending on the final application.

In order to find these parameter configurations and determine the impact of our modifications on the GSST kernel, we first need a baseline measurement. We will find the baseline throughput by repeating the decompression benchmark provided by GSST on our benchmark system. This system, along with the used datasets, is described in more detail in Section 5.1. We will only use the TPC-H benchmark, since this is the only dataset the GSST compressor properly supports.

Figure 4.2 shows the results of our baseline tests for GSST. This shows that our results align with what the authors found, with the exception that our numbers are different because of different hardware and input size. The most optimal configuration achieves approximately 252 GB/s. The results are relatively sensitive to the configuration. This is especially the case with smaller blocks, which result in throughputs as low as 1 GB/s.

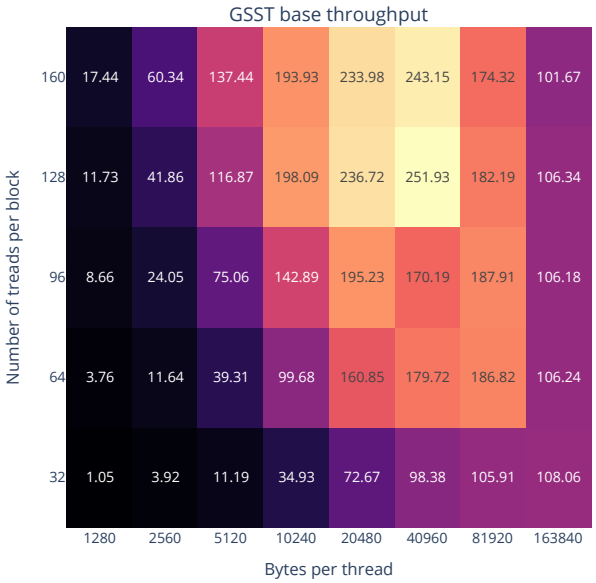


Figure 4.2: The achieved throughput of the GSST decompressor with different configurations, without any changes. This test uses a 2GB TPC-H dataset and runs on an RTX 4090.

4.2. Required modifications

In order to integrate our V5T compressor with the GSST decompressor, the most important step is defining a shared data format. The most involved step is deciding between an interleaved header style like GSST uses, or a fixed header style that we use. After that, we need to modify both pipelines to match this data format.

The reason we use a fixed header style is because of our stream compaction step. This filtering kernel is significantly more performant when applied to the entire dataset, when compared to many smaller blocks. Unfortunately, we cannot apply the filtering kernel on the header data, as we cannot guarantee that our reserved padding character is not used in any of the header data. We would only be able to filter our header data properly when the memory locations are known to the filtering process, as we could then ignore padding characters in memory areas that represent header data. Unfortunately,

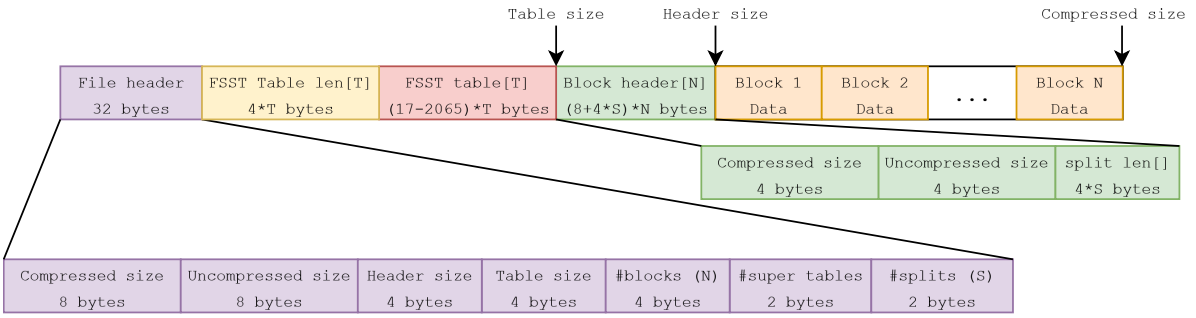


Figure 4.3: The data format used by the combined V5T and GSST (de)compressor. It is close to the V5T data format, with the addition of some header locations, and the split structure has been added to the file and block headers.

this is not possible with the existing libraries. In the case of GSST, there is no technical reason to use the interleaved format. The only reasons are that it is easier to skip entire data blocks, and it keeps the format relatively close to that of FSST. However, remember that FSST explicitly mentioned that the table storage can be flexible, since its main purpose is direct usage in databases. For this reason, adapting the fixed header style is the logical choice from a technical standpoint. Figure 4.3 shows the shared data format for V5T and GSST. This uses the fixed style header format, with the addition of split data in both the block headers and the file header.

The V5T pipeline already closely matches this output format, with the exception of split data. It is trivial to add this data, since every thread is aware of how much data it has produced. GSST requires some more significant changes, as previously it used a single pointer to keep track of the current block location, but the block header and block data are now separated in memory. For this reason, GSST is modified to keep three running pointers: one for the FSST table, one for the block header, and one for the block data. Keeping three running lengths instead of one is slightly inefficient, but greatly simplifies the implementation.

Potential future improvements are removing the running length mechanism and directly accessing the relevant FSST table, block header, and data. This can be achieved by adding direct reference locations to the FSST table and data in the block header. Since the V5T headers are generated in parallel to stream compaction, this should have little to no overhead in the compression pipeline, while it could have a significant impact on the decompression throughput. In addition to that, the decompression pipeline should be made more robust for decompression errors for different datasets. This would allow the full (de)compressor to be more robust overall and be used for end-to-end (de)compression tests for all data types.

5 Results

5.1. Test methodology

In this section, we will describe the system used for most of our tests and describe the datasets we use for the tests in more detail.

5.1.1. Hardware

All performance benchmarks are executed on the same system with the specifications in Table 5.1, unless specified otherwise. For example, if the goal of the experiment is to compare hardware performance. The specifications of the RTX 4090 GPU in the system are shown in Table 5.2. We use the following software versions:

- CUDA 12.8
- NVIDIA driver 570.133.20
- NVIDIA-SMI 570.133.20
- NVIDIA Nsight Systems 2024.6.2.225-246235244400v0
- NVIDIA Nsight Compute 2025.1.1.0
- nvCOMP 4.2.0.11

CPU	AMD Ryzen 9 9950X 16 HW cores, 32 Threads
GPU	NVIDIA RTX 4090 24GB
Memory	2x24GB 8200 MHz, 48GB total
Peripherals	1x Corsair MP700 2TB 1x 5Gb ethernet

Table 5.1: Table showing the hardware specifications of our benchmarking system.

5.1.2. Datasets

We use three datasets to test and tune our compressor pipeline. The choice of data used for tuning and benchmarking purposes will significantly influence the results, which is why we use a wide variety of datasets. We have established that we are focusing on textual data, which means that our datasets also mostly consist of textual data. TPC-H [111] is a good candidate for this, and specifically several string columns such as the `comments` column in the `lineitem` table and the `name` column in the `customer` table. Furthermore, we use location data from the GDelt dataset [30]. Finally, we use part of the DBText corpus used by the original authors of FSST [13], specifically the machine-readable identifiers, which are also common in databases. Table 5.3 shows some statistics and samples of these datasets.

GPU Name	AD102
Architecture	Ada Lovelace
Bus interface	PCIe 4.0 x16
Base clock	2235MHz
Boost clock	2520MHz
Memory clock	1313MHz
Memory size	24GB
Memory type	GDDR6X
Memory bandwidth	1.01TB/s
SM Count	128
L1 Cache (per SM)	128KB
L2 Cache	72MB
CUDA compute cabability	8.9
Blocks per SM	16
Warps per SM	48
Threads per SM	1536
Threads per block	1024

Table 5.2: The hardware specifications of the RTX 4090 GPU used in (most) of our benchmarks. Source: [75, 106]

Name	Example	Symbol lengths								Lookup combinations	
		1	2	3	4	5	6	7	8	Average	Max
lineitem	nal braids nag carefully expres	37	78	25	15	20	21	15	37	3.4	15
customer	Customer#000182752	21	40	53	23	22	27	1	49	4.0	10
locations	Waterkloof, Free State, South Africa	70	69	30	13	12	11	10	29	2.4	12
wiki	Weymouth_New_Testament	73	133	27	6	6	1	2	4	4.2	13
uuid	84c3ba4a-2da5-11e8-885e-87d3525c76d2	19	199	3	17	0	4	1	10	11.7	17
		42	131	19	5	7	9	3	24	4.2	13
hex	http://dblp.l3s.de/d2r/data/publications/...	18	235	0	0	0	0	0	0	13.8	17
urls	Get_Together_(Madonna_song)	80	82	19	9	21	6	7	29	2.7	9
yago	http://fr.dbpedia.org/resource/Le_Grand_...	67	147	31	4	3	1	1	0	4.4	19
urls2		76	109	17	7	12	4	7	22	4.1	12

Table 5.3: The datasets we use and some relevant statistics such as the average symbol lengths and the lookup usage in terms of the number of 2-byte symbols starting with the same character.

5.2. Parameters

In this thesis, we have proposed several compression pipelines, each with its own configurable parameters. During development, we identified a set of parameter values that yield sufficiently high occupancy and avoid obvious misconfigurations. These values were selected based on empirical observation and iterative refinement, rather than through an exhaustive parameter sweep aimed at maximizing compression ratio or throughput.

To evaluate the progression across pipeline designs, we report the baseline compression ratio and throughput for each version. A more detailed performance analysis is reserved for the final version (V5T), where we examine the impact of individual parameters in greater depth. Consequently, this section focuses solely on parameters relevant to the V5T pipeline. The influence of these parameters on compression ratio and throughput is illustrated in Figure 5.1. Note that some parameters, such as `tile_len`, have both a positive and a negative effect on the compression throughput. We will observe and discuss these effects in Section 5.4.2.

We begin with parameters related to the encoding table. The first, `hash_entries`, defines the number of entries in the hash table and corresponds to parameter X introduced in Section 3.3.1. The second, `lookup_rows`, specifies how many unique starting characters can be used for 2-byte symbols. The third, `lookup_combinations`, determines how many symbol combinations are allowed within a single row (i.e., sharing the same starting character). These two lookup parameters were introduced as R and K in Section 3.3.2.

As discussed in Section 3.2.1, work is divided by partitioning the input data into blocks of size `tile_len` $\times$ `num_threads`. Each thread block processes one such data block. To decouple the data block size from the table block size, we employ super block tables, introduced in Section 3.4.1. This design enables independent tuning of data block size for optimal throughput, while adapting the effective table block size to exploit local entropy variations in the input data.

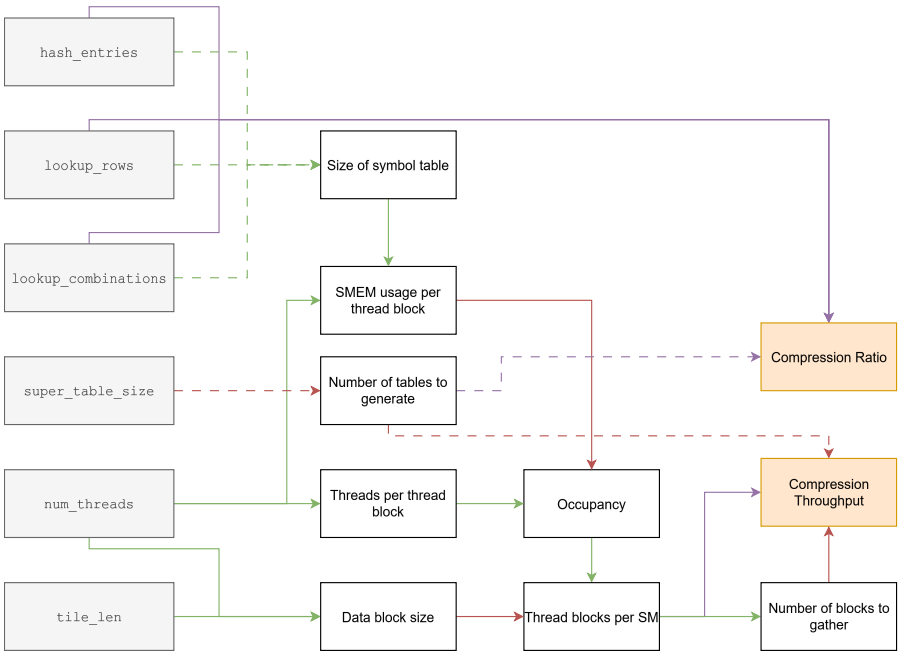


Figure 5.1: The impact of our parameters on the overall compression ratio and throughput. The arrows show how a change in a parameter impacts the overall program. A red arrow means that increasing the source will decrease the sink, while a green arrow means that the sink will also be increased. A purple arrow indicates that the sink will likely also increase, but this depends on external factors and cannot be defined with certainty.

5.3. Lookup table performance

We start by analyzing the effects of the three encoding table parameters on the compression ratio of the three datasets. We achieve this by varying the parameters and observing their effect on the compression ratio. What we hope to see is that the compression ratio converges relatively quickly for all datasets. This means the encoding table can be reduced in size while achieving similar results.

5.3.1. Hash table size

Figure 5.2 shows the effect of changing the number of entries in the hash table. As expected, the compression ratio does not continue increasing when the number of entries is higher, but instead converges to a certain ratio.

For DBText, this happens fairly quickly with only 50 entries at maximum, but it is already close to its converging ratio at 28 entries. GDelt follows the same pattern, but converges later at 94 entries while being saturated at 111 entries.

TPC-H shows an interesting result, where allowing for more entries will decrease the compression ratio at some point. This is mostly because of the customer data and the way the table generation algorithm works. The customer data is essentially just a static string concatenated with an increasing number. Table generation will sample part of the data and prefer longer symbols over shorter ones. This results in long symbols capturing specific numbers, while it would be more beneficial to have shorter symbols that can be used to efficiently encode an increasing number. For the TPC-H dataset, the number of entries continues to grow to 175, while it already reaches optimal compression ratios at 118 entries.

Besides looking at the average of the datasets, it is also interesting to see how specific types of data behave. For this, we can analyze the individual datasets within the DBText collection. We can see that hexadecimal numbers, like the `hex` and `uuid` datasets, barely use the hash table at all, and almost all other datasets are saturated relatively quickly. The only dataset not doing so is the `urls` dataset, which contains many URLs containing the page title. This suggests the hash table is barely used for machine-readable data, while textual data like TPC-H, GDelt, and `urls` allows for longer symbols.

Overall, we can conclude that we can reduce the number of hash entries by half without sacrificing any performance in terms of compression ratio. Or in the case of data like that of TPC-H, the

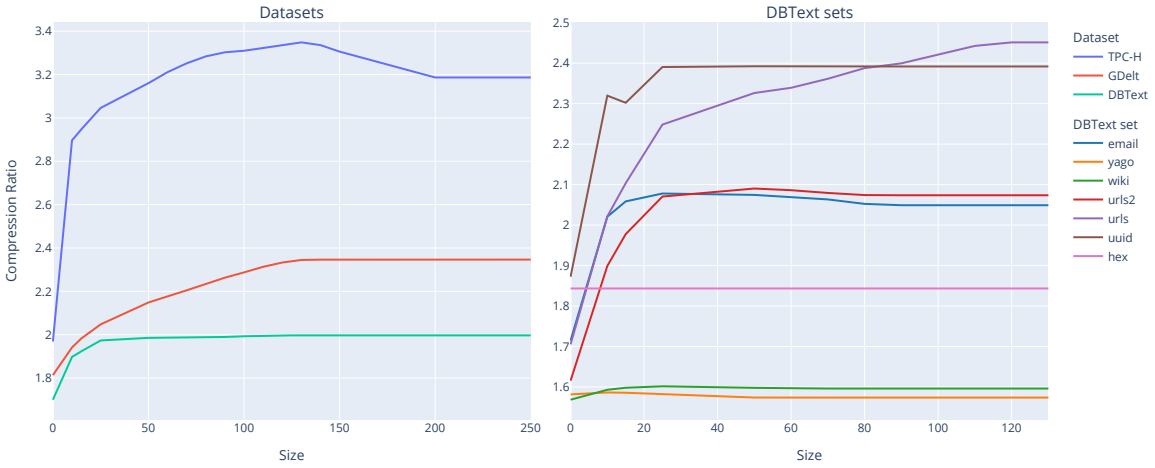


Figure 5.2: The effect of varying the number of entries in the hash table on the resulting compression ratio. It is clear that the hash table can be smaller without sacrificing significant accuracy. The left graph shows the results for all datasets, while the right graph shows the results for individual DBText sets.

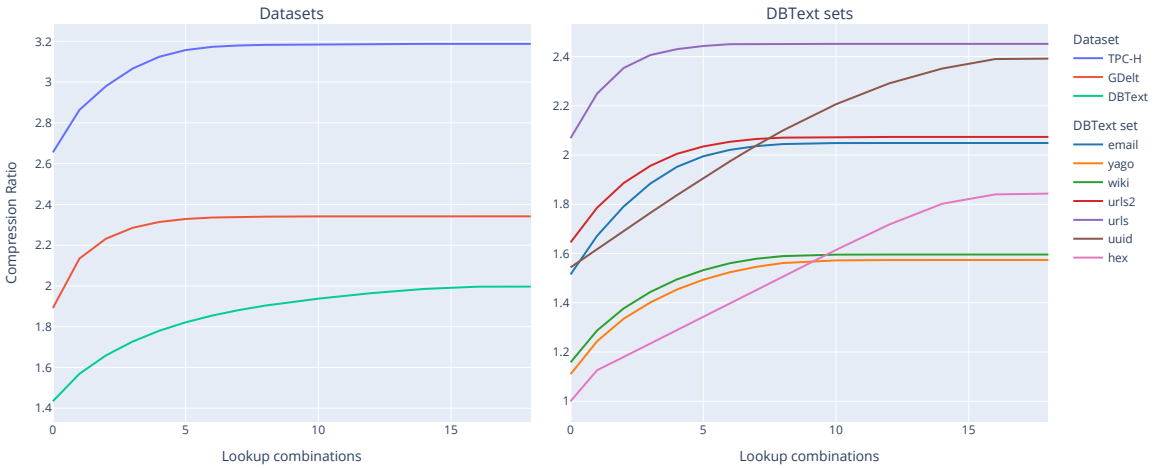


Figure 5.3: The effects of the `lookup_combinations` parameter, which limits the number of possible 2-byte symbol combinations with the same starting character. The textual datasets follow the expected pattern, but DBText deviates from this pattern since it contains machine-readable data. This is due to the two datasets that contain hex data: `hex` and `uuid`.

compression ratio could even increase.

5.3.2. ELL table lookup

We will first consider the main parameter of the ELL-based encoding table described in Section 3.3.2, which corresponds to parameter `lookup_combinations`. This parameter dictates the number of columns in the ELL sparse matrix format.

Figure 5.3 shows the effect of this parameter on the compression ratio for all three datasets. We have already established that the maximum amount of column usage is relatively high compared to the average column usage in Table 5.3, so we also expect the compression ratio to saturate relatively early.

We can now also confirm that the compression ratio saturates relatively close to this average usage. For textual data, the compression ratio saturates at approximately six columns, indicating that in typical string data, there is limited benefit in supporting more than six 2-byte symbols that begin with the same character

The DBText dataset deviates slightly from the general trend, with compression ratio saturation occurring at approximately 13 columns. All datasets follow the previously observed pattern, with the

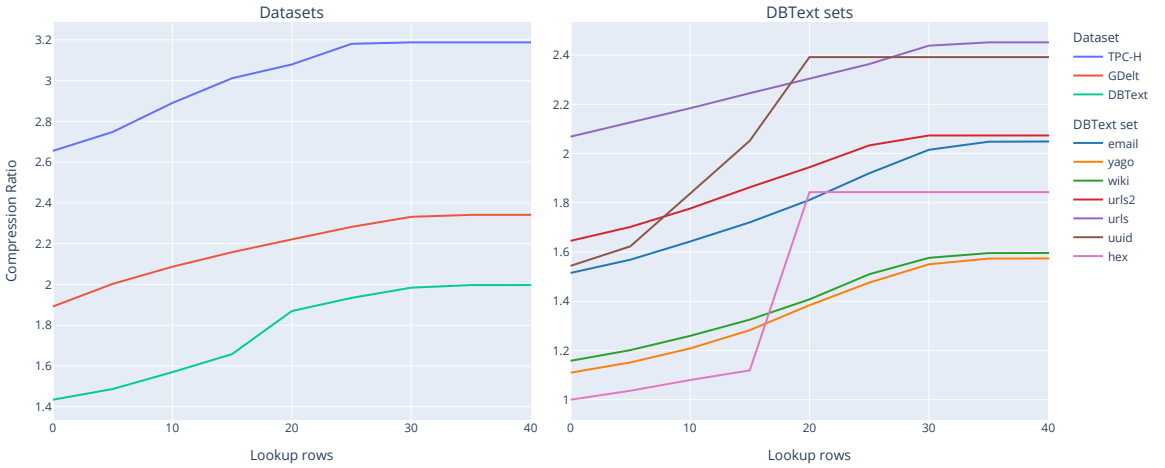


Figure 5.4: The effect of modifying parameter `lookup_rows` on the achieved compression ratio. This parameter limits the number of characters that 2-byte symbols can start with. For all datasets, the compression ratio saturates when most of the (common) characters in the alphabet can be covered, in addition to some special characters.

exception of the `hex` and `uuid` datasets. This deviation is expected, as these datasets are composed almost entirely of hexadecimal values. Specifically, their contents consist of characters matching the pattern `[0-9A-F]+`, which results in a large number of 2-byte symbols. Since there are 16 possible hexadecimal values, the compression ratio continues to increase steadily until all 16 combinations are supported. This pattern aligns precisely with the observed results.

The goal of the ELL table is to achieve close to the same level of accuracy as the original table while reducing its memory footprint. However, the primary goal is not to minimize memory usage at all costs, but to enable a high-throughput compression pipeline that retains compression efficiency. Although a smaller encoding table can improve performance by increasing occupancy due to reduced shared memory usage, this is not universally true. In practice, the main performance benefit is binary: the table either fits within shared memory or it does not.

The original table exceeds the shared memory limits. In contrast, an ELL-based table configured with a `lookup_combinations` value of 8 fits within shared memory. This configuration results in a 96.88 percent reduction in memory footprint, at the cost of only a 1.63 percent average decrease in compression ratio.

5.3.3. Match table lookup

We will now consider the match table described in Section 3.3.2. This encoding table format also uses the `lookup_combinations` parameter that the ELL format uses, with the only limitation that it has to be a multiple of two. Additionally, the match table is also limited in terms of the maximum number of different starting characters, described by the `lookup_rows` parameter. To avoid bank conflicts, this parameter should be a multiple of 32, which corresponds to the number of available banks.

These two parameters are inherently linked, so their linked effects must be investigated. However, we will first investigate the effect of `lookup_rows` without any limitation on the number of allowed combinations described by `lookup_combinations` to see where the compression ratio saturates. We will then use these approximate saturation points to do a combined parameter sweep and identify a local maximum, or rather, try to identify a global trend.

As shown in Figure 5.4, the compression ratio for all datasets approaches saturation at approximately 30 rows. This behavior is expected, as this configuration permits nearly all common alphabetic characters to be used as starting characters, while still reserving additional rows for special characters.

The sharp increase observed in the DBText collection between 15 and 20 rows can be attributed to the characteristics of specific subsets within the collection, particularly the hex-based datasets. As previously discussed, datasets composed predominantly of hexadecimal values place a strong reliance on the lookup table and typically consist of approximately 17 distinct characters. When the number of supported rows in the lookup table falls below this threshold, the encoder is unable to represent all

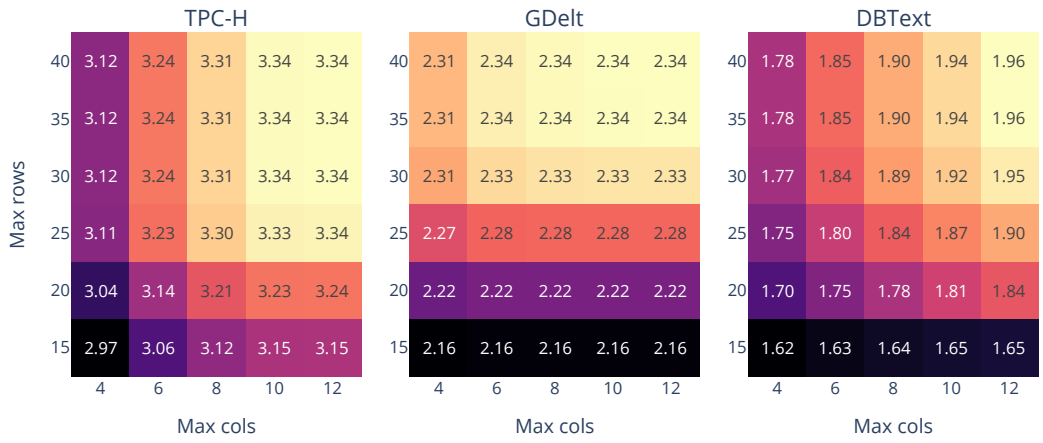


Figure 5.5: The effect of varying the number of rows and columns in the lookup table on the resulting compression ratio.

symbol variations effectively, which leads to a significant reduction in compression ratio. Therefore, limiting the number of rows to fewer than the number of unique characters in the dataset imposes a constraint that directly impacts compression effectiveness.

With these results, we have established the approximate range for the `lookup_rows` parameter. We can then combine that with the approximate range for the `lookup_combinations` parameter found in the previous section to do a 2D parameter sweep. We will repeat the same tests as before to see the impact on the compression ratio. Note that the number of hash table entries is limited to get an accurate estimate of the compression ratio.

Figure 5.5 shows the resulting compression ratios for the TPC-H, GDelt, and DBText datasets. We will briefly discuss the results from every dataset, and then find a set of parameters that work well for all datasets.

Overall, the TPC-H dataset works well with the additional limits, only suffering significantly when extremely limited in terms of `lookup_rows` or `lookup_combinations`. The relative compression ratio is only lower when there are very limited rows or columns available, or when the number of total symbols is limited. When `lookup_rows` is higher than approximately 25 and `lookup_combinations` is 6 or more, the match table performs very well for the TPC-H dataset.

The GDelt dataset seems to be mostly limited by the `lookup_rows` parameter, requiring 25 or more. Increasing `lookup_combinations` does not have a meaningful impact beyond 6.

Finally, analyzing the results of DBText provides some interesting results. Based on the initial research into the landscape of CPU and GPU compressors in Chapter 2, we have already concluded that the machine-readable data in the DBText dataset provides some challenges for existing compressors. It is no different for our compressor, where we have a reduction in compression ratio across the board. Instead of finding a 'break-even' point like for TPC-H and GDelt, we have to determine what is an acceptable decrease in the compression ratio.

We have already determined that the DBText dataset will require at least 17 possible starting characters and 17 possible combinations to achieve the saturation compression ratio. This is because of the hexadecimal data in this dataset. This means a lower number of lookup combinations will always result in a lower compression ratio, even if many rows are available. The other data in this dataset behaves more like the TPC-H and GDelt datasets.

For this reason, the `lookup_rows` should be at least 30 or higher, and `lookup_combinations` should be at least 8, but higher is better. The question of whether a higher value `lookup_combinations` is acceptable depends highly on the amount of available shared memory. Every increase of 2 additional combinations takes up $\texttt{lookup\_rows} * 4$ bytes. This is acceptable if the space is available, but not be acceptable if the occupancy is lowered as a result.

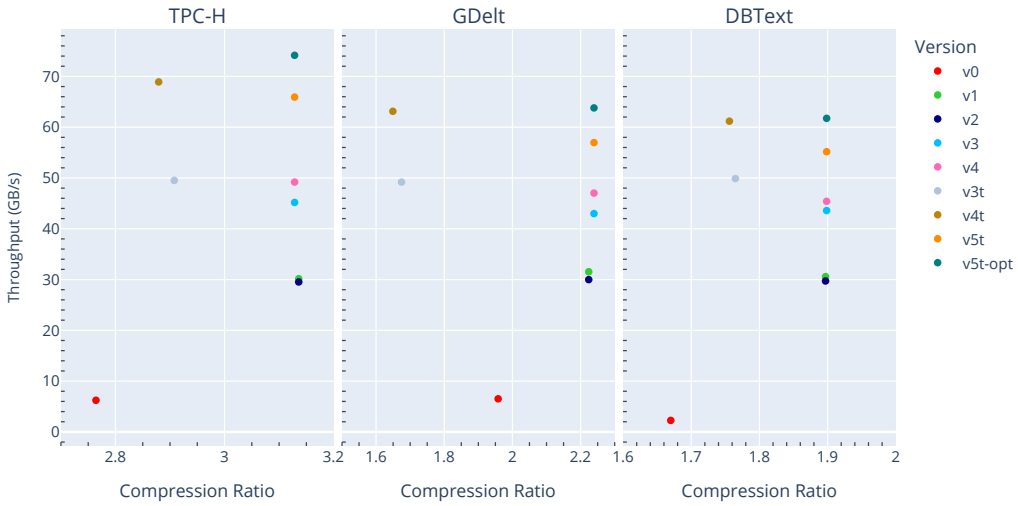


Figure 5.6: The difference in performance between all the pipelines in terms of compression ratio and throughput. Note that `v5t-opt` refers to the parameter-optimized version of the regular V5T pipeline, which we will identify in Section 5.4.2.

As a baseline, 32 lookup rows and 8 lookup combinations provide acceptable results. When using these parameters, we use $1024 + 12 * 128 = 2560$ bytes for the hash table and $32 * 8 * 2 + 256 = 768$ bytes for the lookup table, a reduction of 84 percent and 99 percent compared to FSST, respectively. We will investigate in Section 5.4.3 whether we can slightly modify these parameters without lowering throughput.

5.4. Pipeline performance

In this section, we will show the evolution of our pipelines in terms of compression ratio and throughput, and we will also do a parameter sweep on the best-performing pipeline to achieve the highest throughput for our benchmarking GPU. For our performance tests, we will use the baseline parameters established in Section 5.3, but we will also investigate if we can modify these parameters for a higher compression ratio without sacrificing pipeline throughput. The effects of the modified SM architecture in the Blackwell architecture, as described in Section 3.9, will be investigated in Section 5.7. Finally, we will revisit the GPU compression landscape and discuss where our pipeline stands in Section 5.4.4.

5.4.1. Pipeline evolution

In Chapter 3, we have implemented seven different pipelines. We have the regular compaction pipelines with versions V1, V2, V3, and V4. We also created a pipeline that uses an internal voting scheme to output a compact transposed format with version V3T, V4T, and V5T. This section will focus on the evolution between these versions.

Figure 5.6 shows the difference in performance between these pipelines in terms of compression ratio and throughput. Note that we already added the parameter-optimized variant of V5T for the sake of comparison. Table 5.4 summarizes the incremental speedup when compared to our own baseline, its previous version, and FSST. We will compare our best pipeline to state-of-the-art GPU-based compressors in Section 5.4.4.

The difference between V0 and V1 is significant, both in compression throughput and in compression ratio. The throughput can be explained by the major efficiency improvements from the sliding window and output packing. These significantly impact our effective memory bandwidth. The compression ratio can be explained by the addition of super tables and a different block size. This results in more accurate symbol tables and fewer symbol splits due to small blocks.

The difference between V1 and V2 is interesting. Remember that this is where we added the trans-

Version	Ratio	Throughput (GB/s)	Speedup to previous	Speedup to V0	Speedup to FSST	Speedup to FSST (MT)
V0	2.67	7.52	N/A	N/A	4.27	0.65
V1	3.14	30.16	4.01	4.01	17.11	2.62
V2	3.14	29.52	0.98	3.93	16.74	2.57
V3	3.13	45.17	1.53	6.01	25.62	3.93
V4	3.13	49.18	1.09	6.54	27.90	4.28
V3T	2.91	49.51	1.68	6.58	28.08	4.31
V4T	2.88	68.90	1.39	9.16	39.08	5.99
V5T	3.13	65.92	0.96	8.77	37.39	5.73
V5T-opt	3.13	74.15	1.12	9.86	42.06	6.45

Table 5.4: The iterative improvements of our pipelines compared to their predecessor, our baseline version, and FSST. Note that `v5t-opt` refers to the parameter-optimized version of the regular V5T pipeline, which we will identify in Section 5.4.2.

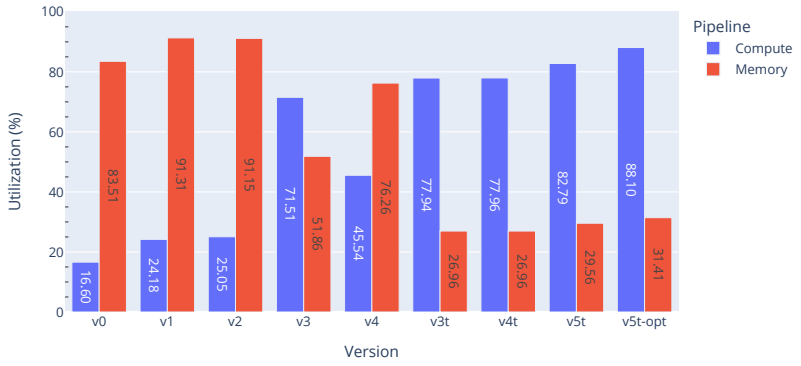


Figure 5.7: The difference in GPU utilization in terms of memory and compute usage. It clearly shows the evolution between versions, and how the kernels evolved to focus on compute. Note that this just focuses on the encoding utilization, so these numbers do not reflect the utilization of the entire pipeline. Note that `v5t-opt` refers to the parameter-optimized version of the regular V5T pipeline, which we will identify in Section 5.4.2.

position stage to the input data, while the tables are being generated. While this does increase the performance of the encoding kernel itself, it will result in an overall slowdown when the transposition takes more time than the table generation. We have a powerful CPU in our test system, so multi-threaded table generation is incredibly fast in terms of throughput. This leads to a performance regression for V2.

The V3 pipeline results in a considerable 53 percent increase in overall throughput by adding a transposition stage to the output of the encoding kernel. This ensures the encoding kernel can perform coalesced writes. The V3T pipeline also branches from V2, but achieves coalesced output writes by implementing a warp-wide synchronization scheme. The V3T pipeline achieves a 68 percent increase in performance, but at the cost of a 7.3 percent reduction in compression ratio. The `-T` branch shifts the main bottleneck of the encoding kernel to compute, since we use memory very efficiently now. This can be seen in Figure 5.7.

The V3 pipeline is further improved by V4 by applying the pipelining technique on the transposition stage. This results in a further increase in throughput of 9 percent. Overall, the V4 pipeline achieves a similar compression ratio to the baseline, but an increase in performance of 554 percent and 63 percent compared to V0 and V1, respectively.

The `-T` branch continues with V4T. This version changes the gather stage of the pipeline by replacing the single stream compaction with targeted direct memory copies. It allows thread blocks to synchronize their output, such that the CPU triggers copies on a block-level granularity. This results in a 39 percent increase in throughput compared to the V3T pipeline.

Unfortunately, the `-T` branch still suffers from a lower compression ratio. This is because of the added padding to maintain a valid output matrix, as described in Section 3.6.4. Furthermore, the final

Threads	Registers	SMEM	Occupancy	Blocks/SM	Threads/SM	Max blocks RTX 4090
32	48	5650	0.3125	15	480	1920
64	48	7700	0.4583	11	704	1408
96	48	9760	0.5625	9	864	1152
128	48	11810	0.5833	7	896	896
160	48	13870	0.625	6	960	768

Table 5.5: Different thread configurations for the encoding kernels, and the resulting resource usage and occupancy.

output of the compressor is still a transposed output format, which is harder to decompress [113]. For that reason, version V5T of the pipeline was introduced. This version introduces yet another transposition stage with an additional compaction stage, which eliminates all padding and creates an identical output format as the non-`T` version at the cost of some throughput. Compared to V4T, this version has a 4 percent decrease in throughput but a 9 percent increase in compression ratio. This results in V5T having an identical compression ratio to V4, but with a relative increase in throughput of 34 percent.

By optimizing the parameters, we can further optimize V5T. We will explore the optimal parameters in the next sections, but for the sake of completeness, we will already discuss the final results here. Compared to the regular variant, the optimized variant of V5T achieves a 12 percent increase in throughput. Overall, this variant achieves an increase in throughput of 886 percent and 146 percent compared to V0 and V1, respectively, making it superior to the non-`T` versions.

5.4.2. Modifying work division

To determine the optimal throughput, we perform a parameter sweep for the number of bytes every thread processes (`tile_len`) and the number of threads within a thread block (`num_threads`). We modify the super table configuration for every test such that every data block configuration keeps the same table block size, ensuring differences in compression ratio are only because of the different data block sizes.

Before we look at the results, we can already reason about what we expect to happen when changing the data size. First, changing the number of threads within one group will use more shared memory, since we use more memory to read data and keep temporary results. In turn, this will lower the number of active blocks per SM. However, the total number of threads per SM might still increase, which is beneficial for throughput. Furthermore, the final aggregation step in the V5T pipeline benefits from having fewer blocks to process because of the individual device memory copy. However, as described in Section 3.6.4, all warps within a thread block will have to synchronize to ensure they all output the same amount of data. This means that a larger number of warps within a single block might lead to more padding, lowering the throughput of the final gather and compact stage.

Modifying the amount of data processed by each thread generally produces similar effects on the final V5T stage. However, there is an additional consideration that must be taken into account. If the data blocks become too large, the number of thread blocks available for encoding may no longer be sufficient to fully saturate the streaming multiprocessors (SMs).

Table 5.5 presents the occupancy levels of our encoding kernel, as measured using Nsight Compute, and indicates how many thread blocks can execute concurrently per SM. Table 5.6 lists the number of thread blocks required to encode files of varying sizes. Based on these tables, one might expect a sharp decline in throughput when larger data blocks are used, due to underutilization of the SMs.

While occupancy is not the only determinant of overall throughput, it remains an important factor. File size and GPU configuration can have a substantial impact on performance, particularly when parameters are not adjusted dynamically. In this thesis, we adopt static configuration parameters. However, in practice, dynamically adapting these parameters to match hardware characteristics and input sizes would likely yield improved performance.

We will now consider the actual measured throughput to check our hypothesis. Figure 5.8 shows the measured throughput for the TPC-H, GDelt, and DBText datasets, respectively. These results follow our expectation that larger data block sizes are beneficial to the overall compression throughput, given that all SMs can still be saturated with thread blocks.

We can see that, at least for these datasets, the benefits of having more warps coordinate their output before the aggregation stage outweigh the additional compaction costs due to extra padding. However, more warps will lead to a degradation in performance eventually due to the ever-increasing shared memory usage. This does show there might be potential to use cooperative groups [38] to create groups of warps, as opposed to grouping all warps within a thread block. This would achieve

Threads	File size	Total blocks per config						
		1280	2560	5120	10240	20480	81920	163840
32	2GB	48320	24160	12080	6040	3020	**755**	**377**
64	2GB	24160	12080	6040	3020	1510	**377**	**188**
96	2GB	16106	8053	4026	2013	**1006**	**251**	**125**
128	2GB	12080	6040	3020	1510	**755**	**188**	**94**
160	2GB	9664	4832	2416	1208	**604**	**151**	**75**
32	4GB	96640	48320	24160	12080	6040	**1510**	**755**
64	4GB	48320	24160	12080	6040	3020	**755**	**377**
96	4GB	32213	16106	8053	4026	2013	**503**	**251**
128	4GB	24160	12080	6040	3020	1510	**377**	**188**
160	4GB	19328	9664	4832	2416	1208	**302**	**151**

Table 5.6: The number of active thread blocks per different encoding configuration. All numbers in bold are lower than the possible number of active blocks based on the occupancy, which means there will definitely be idle SMs for those configurations.

the same goal of reducing load on the aggregation stage, but would not create large thread blocks with a high shared memory requirement.

The impact of insufficient SM saturation becomes evident when the number of bytes per thread exceeds 81920 bytes for a 2GB input file. This observation is further confirmed by increasing the input file size, where a similar performance pattern emerges. However, in this case, the point of sharp performance degradation shifts to a higher threshold.

The effect on individual stages of the pipeline can also be examined in more detail. Figure 5.9 illustrates the performance trends of the two main stages: encoding and compaction. The encoding stage applies the encoding table to transform the input data, while the compaction stage aggregates the output from all blocks and removes padding. The figure provides additional confirmation of the earlier hypothesis. The compaction stage benefits significantly from both smaller and larger data blocks, whereas the encoding stage suffers a performance decline when there are too few thread blocks to fully saturate the SMs. This interaction between the stages results in an optimal range that yields the highest overall throughput.

Ideally, the tile size should scale dynamically with the input file size in order to maintain optimal throughput across varying datasets. Additionally, increasing the number of warps per thread block positively impacts the compaction stage, but negatively affects the encoding stage due to increased pressure on shared memory. One possible approach to mitigate these negative effects and further improve performance is the use of cooperative groups, as previously discussed.

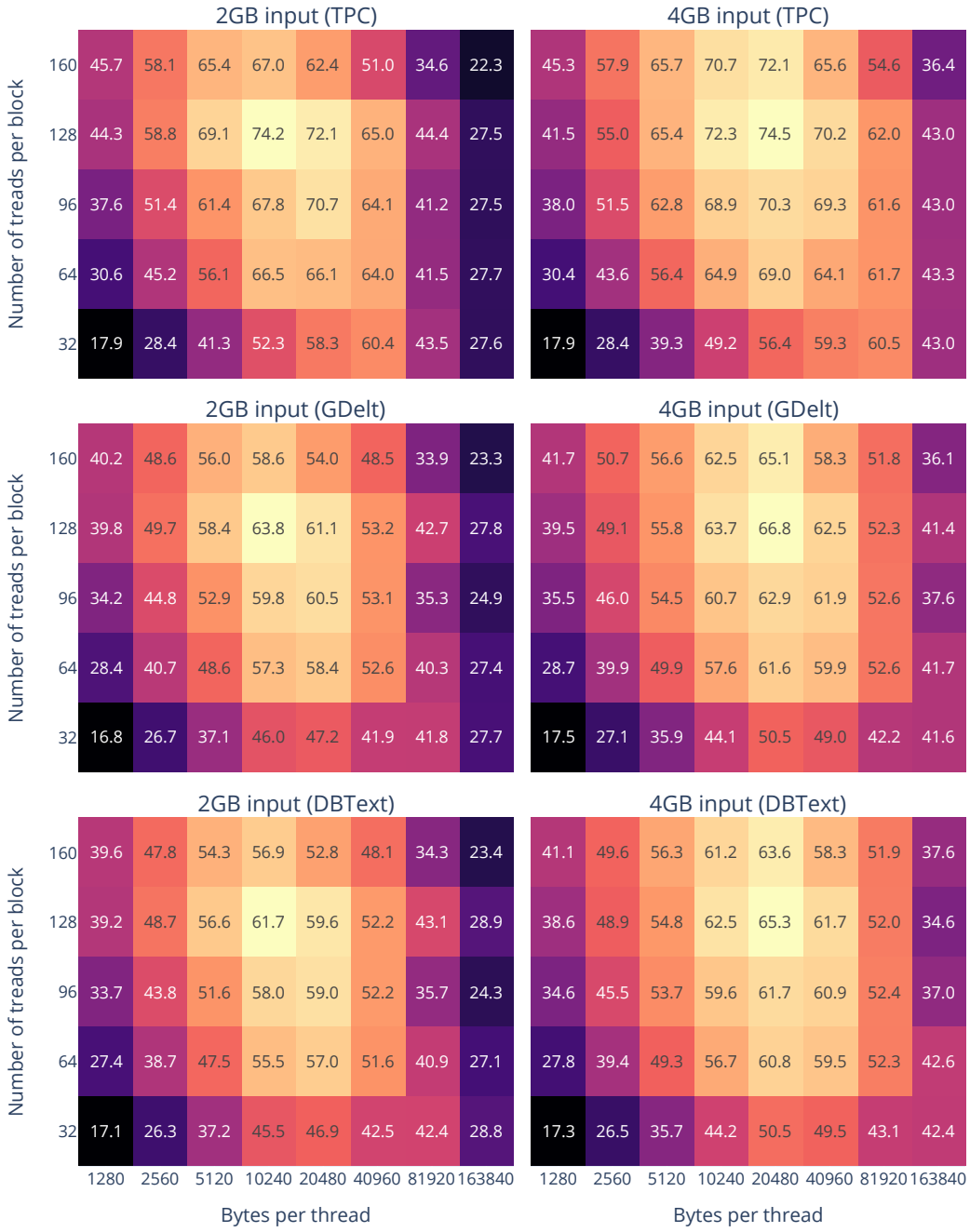


Figure 5.8: Heatmap of the overall compression throughput of the V5T pipeline when varying the two main work division parameters: the number of bytes per thread, and the number of threads per thread block. This test is performed on all datasets and for multiple file sizes to observe the effect.

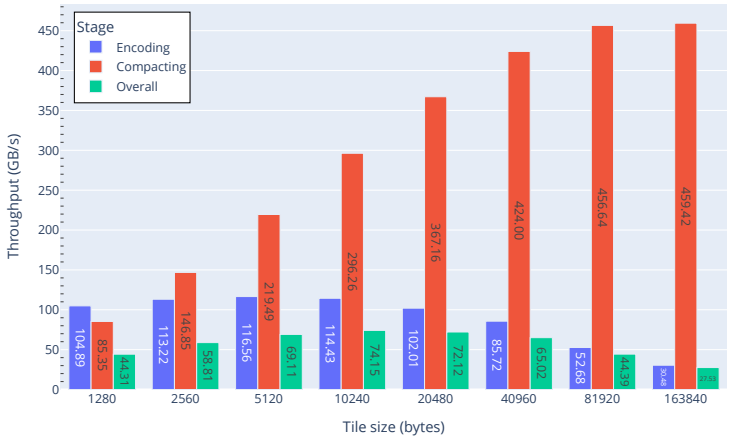


Figure 5.9: The evolution of the throughput of the individual stages in the pipeline. This figure shows the evolution of the pipeline on the TPC-H dataset using 128 threads per block. It clearly shows how the overall performance is limited by the encoding kernel as the data block size increases, whereas the compaction stage is the bottleneck with many small data blocks.

5.4.3. Optimizing compression ratio

Besides optimizing the work division parameters for throughput, we also need to consider the impact on the compression ratio. Ideally, there is little to no effect, meaning we can focus on optimizing only one parameter: throughput. With the introduction of *super tables* in Section 3.2.1, the compression ratio should barely be affected since the table generation algorithm uses the same amount of data to generate symbols. Figure 5.10 shows the compression ratio as a result of the work division parameters, and confirms that the compression ratio is minimally affected.

Based on the table parameters identified in Section 5.3, we have already achieved acceptable ratios. However, if the increased shared memory pressure does not lower the occupancy, we might be able to increase the compression ratio further. Figure 5.11 shows the effect on overall pipeline throughput as a result of higher shared memory usage when modifying the number of columns, which corresponds to the `lookup_combinations` parameter. This shows that a reduction in throughput of roughly three percent for all datasets results in an increase in compression ratio of roughly two percent for machine-readable data. For this reason, we will not modify the parameters beyond the ones defined in Section 5.3.

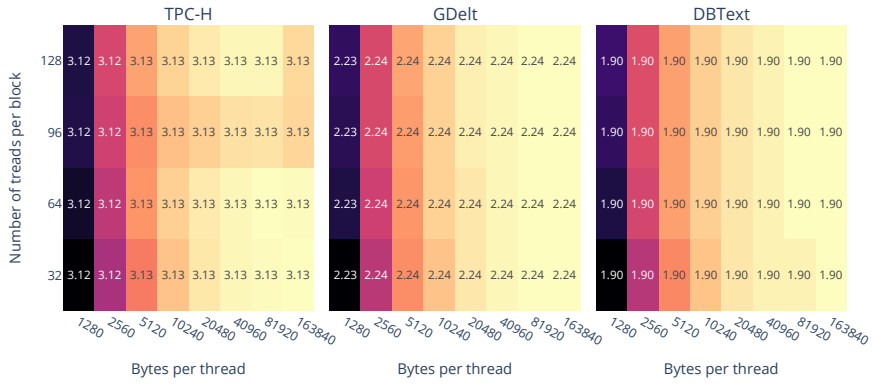


Figure 5.10: The effect of varying the work division parameters on the resulting compression ratio.

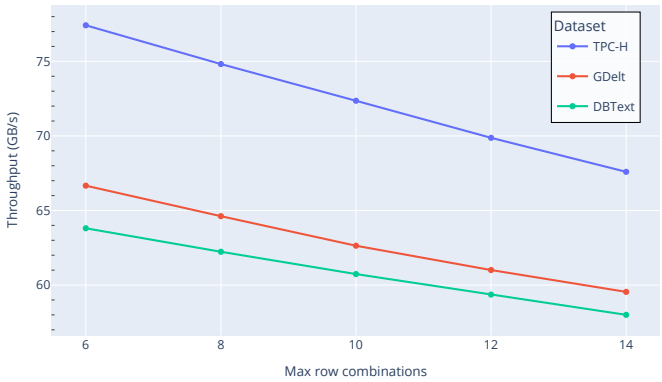


Figure 5.11: The effects on overall throughput of changing the maximum number of columns, as a consequence of lower occupancy.

5.4.4. Redefining the GPU compression landscape

In this section, we will revisit the GPU compression landscape that we defined in Section 2.3.2 and add our own pipeline. Furthermore, we will examine how these results can be related to the original problem of GPU memory offloading posed in Section 1.2. We will only focus on the V5T pipeline, which is the most performant overall pipeline.

Figure 5.12 shows the current view of the state-of-the-art compressors, as mentioned in Section 2.3.2. Our best pipeline, V5T, is marked with an orange star. V5T is part of the Pareto front, i.e., there exists no compressor that is both faster and compresses more, for all datasets, meaning that our pipeline is part of the new state-of-the-art.

For the TPC-H dataset, this concretely means that we outperform GPULZ, LZ4, and Snappy in terms of both compression ratio and throughput. We are more performant than (G)Deflate, but achieve a lower compression ratio. The GDelt dataset does not work nicely for the static symbol table, which results in a relatively low compression ratio compared to LZ4 and Snappy. We are, however, still part of the Pareto front. The final dataset, DBText, is challenging for all compressors except the ones that operate on the data as if it were non-textual data, like Bitcomp and ANS. We are still part of the Pareto front, but for machine-readable data, this pipeline has a very narrow band where it will be the best option.

We also added the results for the original FSST paper to the graph to compare overall compression ratios. We achieve nearly identical compression ratios, except for the machine-readable data as explained in Section 5.3, while still achieving a considerable speedup. Even when compared to a multithreaded CPU implementation, we achieve a 6.45x speedup as shown in Table 5.4.

To make these results easier to graph and relate back to our original problem statement, we will also present the results (for the TPC-H dataset) in terms of effective throughput. Based on Equation 2.5 defined in Section 2.1, we can determine the effective throughput for a compression algorithm on a link with a given bandwidth, given its achieved compression ratio and compression throughput. Using this equation, we can directly compare the compression algorithms for this scenario. Figure 5.13 shows the results for a selected group of compression algorithms.

We have marked three important points in this graph with red dotted lines. The first line shows the threshold at which our pipeline is the absolute fastest; given the link bandwidth is higher than 3.5 GB/s, our pipeline will outperform all of NVIDIA's compressors in addition to GPULZ and LC3. Below this threshold, ZSTD achieves a higher throughput due to its very high compression ratio.

The next important point is located at 25.5 GB/s. This is where NVIDIA's flagship compressor, ANS, starts to outperform our pipeline. This means that our pipeline is the best available compressor when the transfer bandwidth is between 3.5 and 25.5 GB/s.

However, if the nvCOMP compressors are not an option because of their high memory usage, the main comparison is between using our pipeline and using no compression at all. This crossover point is

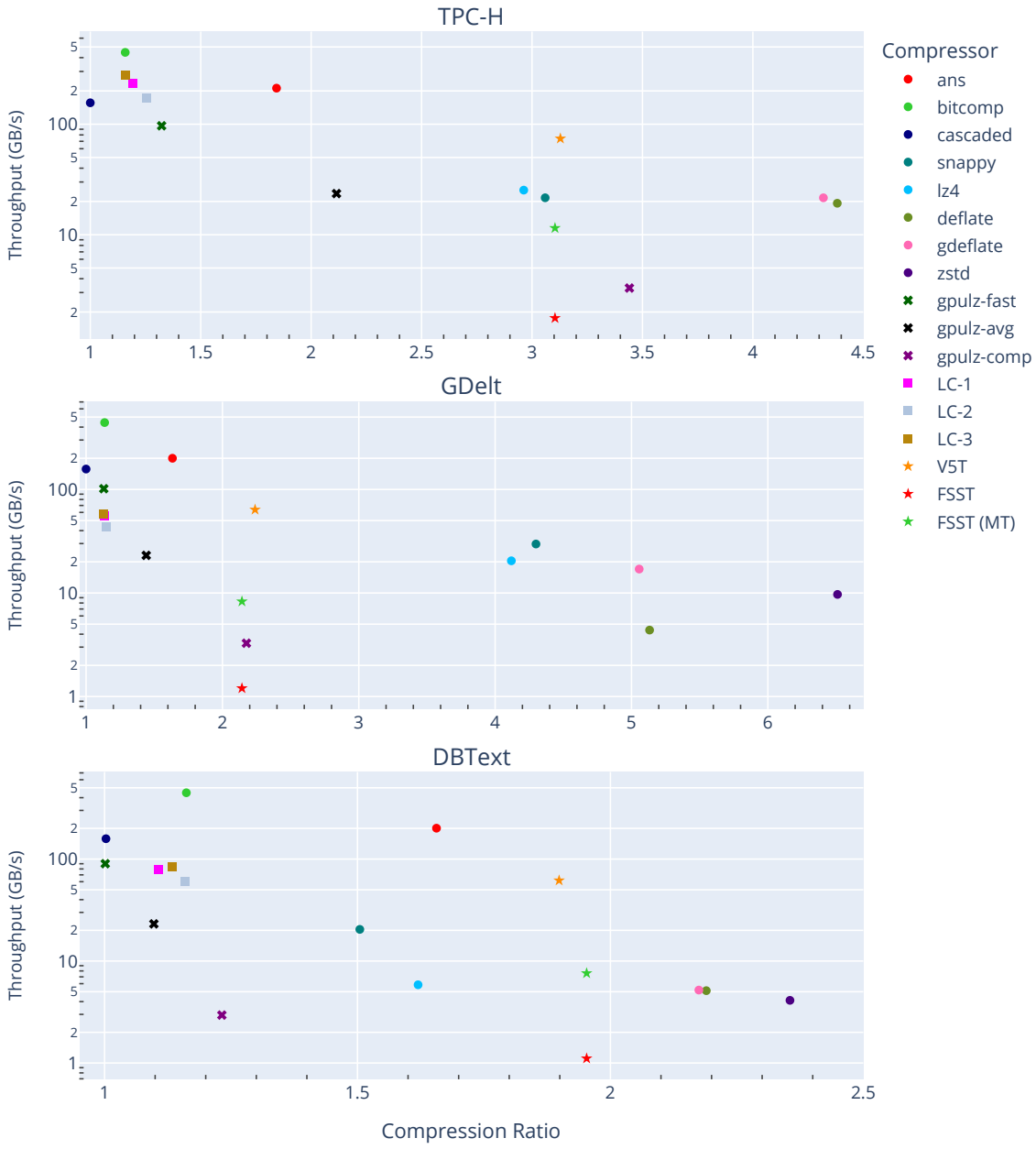


Figure 5.12: We compare our proposed compression pipeline to the nvCOMP compressors, GPULZ, compressors generated with the LC framework with a different number of stages, and the original FSST algorithm, regarding compression ratio and throughput. All benchmarks were completed on the same machine (RTX 4090 with Ryzen 9 9950X) and used the same 2GB files. The V5T compressor, marked with an orange star, is a Pareto point in all datasets.

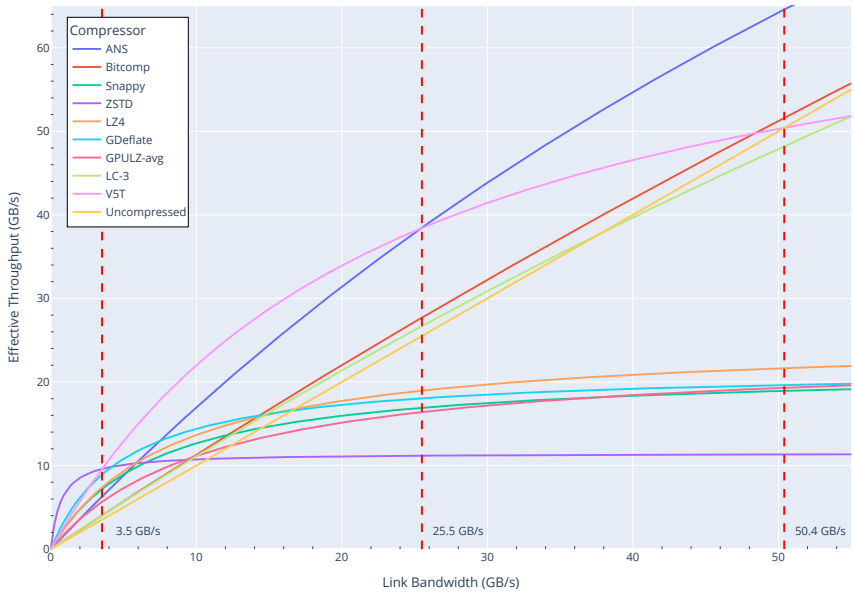


Figure 5.13: The effective bandwidth of compressing data before transmitting it over a link with a given bandwidth. The effective throughput is based on Equation 2.5 defined in Section 2.1. Our pipeline is the best available option when the link bandwidth is between 3.5 and 25.5 GB/s, and beats an uncompressed transmission when the link bandwidth is lower than 50.4 GB/s.

marked by the third line, which is located at 50.4 GB/s. At that bandwidth, our compressor throughput starts to become a limiting factor. From this point, a higher effective throughput can be achieved using no compression. Meaning our compressor outperforms a plain uncompressed link when the bandwidth is lower than 50.4 GB/s, which covers the theoretical limits of PCIe 4.0 and approaches those of PCIe 5.0.

5.5. Resource consumption

In Section 1.2, we have established some questions related to the problem scope. One of the points was to investigate the performance of our pipeline in terms of several metrics. We have discussed the two primary metrics in Section 5.4.2 and 5.4.3, but we haven't discussed memory usage and energy consumption yet. We will first discuss memory usage and then briefly discuss energy consumption.

We have established that memory usage is important for compression purposes, as the usefulness of a compressor is severely limited when it has very high memory consumption. During the process of developing our pipelines, we have mainly focused on the two primary metrics, but we have always tried to keep memory usage in mind. To compare to the state-of-the-art in terms of memory usage, we have measured the global memory usage (VRAM) for several compressors while compressing several files of our TPC-H benchmark data. The results are shown in Figure 5.14. It can be seen that our compressor is competitive with the state-of-the-art, as we use less memory than all of the nvCOMP compressors and GPULZ. The compressor generated with the LC-Framework uses less memory, but is unable to compress files larger than 2 GB. Because of some internal overhead, the difference between V5T and the other compressors becomes more significant once the input size grows.

In the case of our pipelines, the main memory usage comes from temporary buffers required to store intermediate data, as outlined in Section 3.7.3. Besides using a small amount of temporary storage for metadata, the main memory requirements for every pipeline are defined by their temporary data buffers and are listed in Table 5.7. This shows that the `-T` branch of pipelines also uses less memory, besides being more performant.

Besides the memory usage, we also measured the energy consumption using the *nvidia-smi* tool.

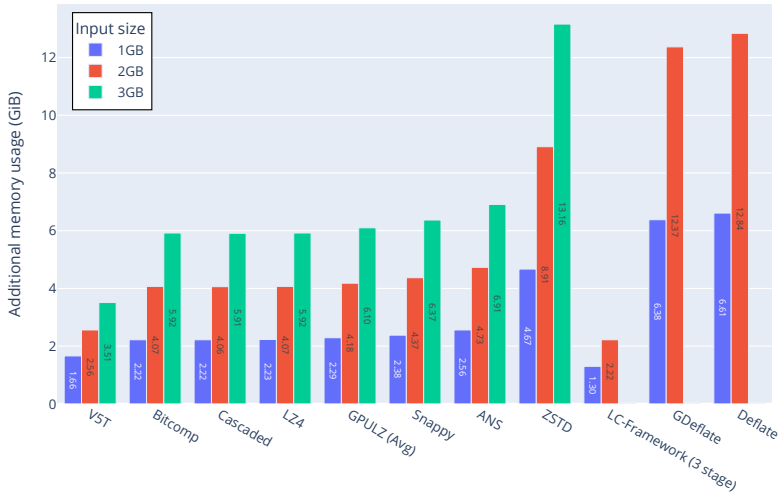


Figure 5.14: Memory usage, measured with `nvidida-smi`, excluding input and output buffers of our compression pipeline, compared to other state-of-the-art compressors. The compressors are sorted based on their memory usage, and empty columns indicate that a compressor ran out of memory or failed to compress the file.

Version	Data buffers	Aux buffers
V0	2x	6.4Kib per table
V1	3x	3Kib per table, 8 bytes per block
V2	3x	3Kib per table, 8 bytes per block
V3	5x	3Kib per table, 8 bytes per block
V4	5x	3Kib per table, 8 bytes per block
V3T	2x	3Kib per table, 16 bytes per block
V4T	2x	3Kib per table, 16 bytes per block
V5T	1x	3Kib per table, 10 bytes per block

Table 5.7: The memory usage per pipeline version in terms of data buffers and auxiliary buffers. The size of the data buffers is defined in terms of the data input size. Note that the aux buffers do not include buffers used by Thrust to perform stream compaction.

Compressor	Peak usage (W)	Time (ms)	Energy usage (Ws)
LC3	84.09	7.10	0.60
ANS	139.06	9.60	1.33
V5T	363.44	26.89	9.77
GPULZ	148.30	85.68	12.71
Snappy	314.78	162.9	51.28
LZ4	385.36	190.73	73.5

Table 5.8: The energy usage for different compressors, sorted by overall energy usage in Ws (J) to compress a single 2GB file. The peak usage was determined with the `nvidida-smi` tool, with a reported accuracy of about 5 percent.

This tool uses the GPU's onboard measurements, which are accurate to within 5W according to *nvidia-smi* (by querying the manual with the `'--help-query-gpu'` flag). However, more in-depth research shows this inaccuracy is closer to 5 percent [124]. Furthermore, not all board power is drawn by the compressor, since this system is also running a normal operating system with a desktop manager. Nevertheless, by running multiple compression cycles on the same machine under the same circumstances, the results should be usable.

Table 5.8 shows the results of our experiment with different compressors. For this test, we looked at the peak energy usage over multiple runs and calculated the overall energy usage per compression cycle based on this usage and the approximate time it took based on the achieved compression throughput. The test is not ideal, as different compressors use different benchmark suites, but even with a high overall error margin, we can say something about the general usage characteristics. In general, our pipeline has a high peak usage approaching the maximum energy limit for the RTX 4090, similar to the behaviour of Snappy and LZ4. GPULZ and ANS are similar in their usage, while LC3 clearly has the lowest peak usage. However, because our pipeline runs significantly faster than GPULZ, Snappy, and LZ4, our overall energy consumption is lower. Overall, LC3 and ANS are clearly the most efficient compressors.

5.6. Overall (de)compression performance with GSST

We have now investigated the performance of our compressor, but we have yet to investigate the performance of the full (de)compressor that is a combination of our V5T compressor and the GSST decompressor. We will do so in this section.

Figure 5.15 shows the throughputs achieved by the compressor (V5T), decompressor (GSST), and the combined throughput, respectively. The combined throughput is calculated using the equation $TH_{\text{combined}} = (\frac{1}{TH_C} + \frac{1}{TH_D})^{-1}$. The first observation is that the configuration favoring compression throughput did not change when compared to the results obtained in Section 5.4.2. Furthermore, the configuration favoring decompression throughput is different than the one favoring compression, as expected. However, the behaviour of the decompression kernel is similar to that of the compression pipeline, but the negative effect of small blocks is more pronounced. This shows that the optimal performance range is narrower for decompression than it is for compression, i.e., the decompression kernel is more sensitive to improper parameter tuning. The combined throughput shows that the maximum throughput is mostly limited by the compression pipeline, while it inherits the narrow performance band from the decompression kernel. Fortunately, the balanced profile, which maximizes combined throughput, is very similar to the other two profiles. For all profiles, 128 threads is optimal. The balanced and compression profiles are essentially equal, while the optimal decompression uses a tile size twice that of the balanced profile.

The V5T compression pipeline suffers from a 5 percent reduction in throughput with the modified data format. The compression ratio is not significantly impacted, with a reduction of only 0.05 percent due to the additional metadata in the headers. We slightly modified the GSST kernel to incorporate our data format, and in the process, we slightly optimized the parsing of header data. This resulted in a marginal increase in throughput of 7 percent, showing there is likely more to gain from improving the general kernel structure.

We will repeat the test in Section 5.4.4 to show the effective bandwidth over a link, this time incorporating both compression and decompression throughput. Figure 5.16 shows the results, and we can see that the combined (de)compressor has the highest effective throughput when the link bandwidth is between 3.1 GB/s and 20.1 GB/s. Below 3.1 GB/s, ZSTD dominates all other compressors because of its high compression ratio. When the link bandwidth is higher than 20.1 GB/s, ANS achieves a higher effective bandwidth, but at the cost of almost twice the amount of memory usage. If the link bandwidth is

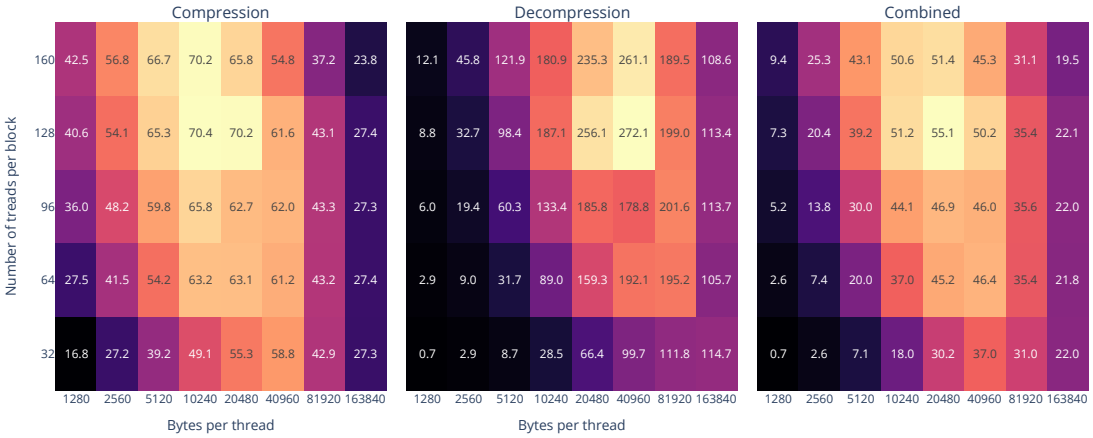


Figure 5.15: Overall compression throughput for the (de)compressor that combines V5T and GSST. The decompression kernel has very high maximum throughput, but a narrow performance band. The compression kernel has a lower maximum throughput, but is less sensitive to the configuration and provides a more balanced throughput. The benchmark uses a 2GB data file with TPC-H data, and is performed on our RTX 4090 system.

less than 37.5 GB/s, the combined (de)compressor is faster than using an uncompressed data transfer. For reference, the theoretical maximum bandwidth of a PCIe 4.0 x16 link is 32 GB/s.

To conclude, the combined compressor achieves considerable throughput without significantly affecting the compression ratio. We propose two configurations that can be used by an end-user to suit the target application: one configuration favors decompression throughput, and the other uses a more balanced profile that maximizes combined throughput and also has high compression throughput. The balanced configuration offers a competitive effective throughput over a wide range of link bandwidths, beating the state-of-the-art for all links between 3.5 and 20.1 GB/s. This corresponds to a network link of 28 to 300 Gbps, which corresponds to modern high-performance networks.

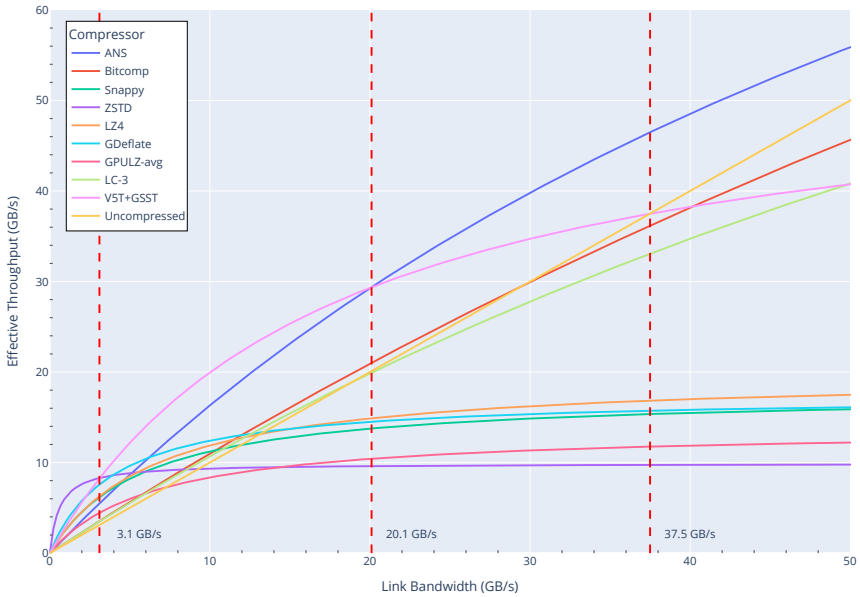


Figure 5.16: The effective throughput of a transmission where the data is first compressed, then transmitted over a link with limited bandwidth, and then decompressed. Our combined compressor outperforms all other available compressors when the link bandwidth is between 3.1 and 20.1 GB/s, and achieves a higher effective throughput when the link bandwidth is lower than 37.5 GB/s. This corresponds with 300 Gbps networking and PCIe 4.0 x16 busses.

5.7. Using the Blackwell architecture

As discussed in Section 3.9, the V5T pipeline makes extensive use of the ALU pipeline on the Streaming Multiprocessors (SMs), due to its high usage of integer instructions. To evaluate whether architectural changes in the SMs of the Blackwell generation positively impact our pipeline, we benchmarked the V5T encoder and several nvCOMP compressors on the RTX 5090.

Table 5.9 presents the achieved (de)compression throughputs for both the RTX 4090 (Ada architecture) and the RTX 5090 (Blackwell architecture). The table also includes the relative speedup between the two GPUs. Note that we only list the encoding throughput for V5T, as we are specifically interested on the effects on the GPU kernel, not the pipeline as a whole.

The relative performance improvements vary significantly between compressors. Generally, there are compressors that seem to achieve about a 25 percent increase in compression throughput, and compressors that achieve a roughly 75 percent increase. To assess whether these improvements are due to architectural changes in the RTX 5090 or other factors, we first need to estimate the expected performance gain from this new generation.

One way to approximate this baseline is by comparing theoretical compute performance. The RTX 5090 delivers a peak FP32 throughput of 104.9 TFLOPS, while the RTX 4090 achieves 82.58 TFLOPS [106, 107], indicating a roughly 27 percent improvement. Alternatively, synthetic benchmarks such as 3DMark suggest an average performance uplift of around 41 percent [91], while in-game performance comparisons report a more modest 23 percent increase [108]. Although none of these benchmarks directly map to our use case, they suggest that the RTX 5090 offers an approximate 30 percent improvement in raw compute performance over its predecessor.

This expected 30 percent improvement aligns well with the observed performance scaling of compressors like Snappy and Cascaded. However, our pipeline, in addition to several nvCOMP compressors, performs better than what the generational difference alone would predict. Unfortunately, due to the proprietary nature of nvCOMP and the lack of access to performance counters on the RTX 5090 system, it is difficult to determine exactly which aspects of these compressors lead to their improved performance on Blackwell. Nevertheless, since these compressors rely heavily on integer arithmetic, it is plausible that the modified SM architecture is at least partially responsible for this behaviour.

Compressor	Compression throughput RTX 4090 (GB/s)	Compression throughput RTX 5090 (GB/s)	Decompression throughput RTX 4090 (GB/s)	Decompression throughput RTX 5090 (GB/s)	Speedup (comp/decomp)
V5T-GSST (enc)	102.88	172.96	256.14	303.84	1.68 / 1.19
ANS	209.56	389.69	427.82	735.63	1.86 / 1.72
Bitcomp	449.79	813.87	419.20	466.70	1.81 / 1.11
Cascaded	155.95	204.51	374.55	695.44	1.31 / 1.86
Deflate	8.36	14.88	36.03	47.46	1.78 / 1.32
GDeflate	8.65	14.97	82.66	60.64	1.73 / 0.73
LZ4	10.07	18.08	89.57	125.19	1.80 / 1.40
Snappy	12.17	14.68	82.35	116.49	1.21 / 1.41
ZSTD	5.08	0.00	65.07	0.00	0.00 / 0.00

Table 5.9: Achieved (de)compression throughputs for several pipelines on both the RTX 4090 and RTX 5090. The variation in relative speedups between compressors suggests that factors beyond the generational performance difference contribute to the observed results. Only the encoding throughput is reported for V5T-GSST.

6 Conclusion

6.1. Answer to research questions

In this thesis, we explored the realm of GPU-accelerated compression using CUDA on NVIDIA GPUs. The goal of the thesis was to accelerate an existing CPU-based compression scheme using NVIDIA GPUs and to integrate the resulting implementation with an existing decompression system to advance the state-of-the-art in data compression. The original research questions posed in Chapter 1 define the scope of our thesis and can be used to measure our success in our exploration. We will now revisit and answer these questions using the proposed design and results introduced in the thesis.

- **How can we GPU-accelerate an existing string compression scheme for use cases such as the described query engine memory offloading?**

  After background research in Chapter 2, the FSST compressor was deemed the most suitable algorithm to accelerate. Both because of its use of static tables and the fact that a GPU-accelerated decompressor already exists. In Chapter 3, we have introduced a design for an FSST-based compression pipeline, maximizing our throughput using the unique architecture of GPUs. We can apply tiling to create parallelism within a single block, which matches the SIMT programming model of GPUs. By efficiently using shared memory and 32-bit registers with our sliding window, matching table, and output packing, we maximize the throughput of our encoding kernel. We can further optimize the overall throughput by optimizing our effective memory bandwidth using our voting mechanism, which allows for efficient data gathering. We minimize the impact of our pipeline on the compression ratio by employing an additional transposition stage in combination with stream compaction.

- **Can we implement this scheme fully on-chip to use it for memory offloading? Would a heterogeneous solution be sufficient or preferred?**

  The only step ill-suited for an on-chip implementation is the table generation, due to its usage of priority queues and high inherent divergence between tables. For this reason, a heterogeneous solution that also utilizes the CPU is better suited for the overall pipeline. A multithreaded CPU implementation can reach very high throughput. This is partially because we do not require expensive data transfers from the GPU to the CPU for this step, since we only require a small sample of the original data to be present on the CPU itself to generate accurate tables.

- **Can we integrate our compressor with an existing GPU-accelerated decompressor for optimal performance?**

  The GSST decompressor uses a static symbol table to decompress data, which matches the FSST static table. In Chapter 4, we introduced a standard data format and modified both GSST and our V5T pipeline to integrate them into a single high-throughput (de)compressor. The results in Chapter 5 show that we can indeed achieve both high compression throughput and decompression throughput, while maintaining a high compression ratio. We identified three optimal profiles: one favoring compression throughput, one favoring decompression throughput, and one

balanced profile. These three configurations can be used to ensure the (de)compressor can be optimized for the end-user.

- **How can we tune our parameters to achieve optimal key performance metrics such as compression ratio, throughput, and memory usage?**

  We defined a parameter mapping for all our parameters in Chapter 5. Then we performed a parameter sweep to get the optimal parameters for both our lookup tables and the compression pipeline. The results of these experiments were used to evaluate compression ratio, throughput, memory usage, and energy consumption. The analysis demonstrated that the proposed approach achieves an excellent compression ratio and throughput, and lies on the Pareto frontier with respect to these two metrics. Additionally, it consumes significantly less memory than comparable compressors. Energy usage also remains within acceptable bounds when compared to state-of-the-art methods.

- **What speedup can we get from this GPU-accelerated compression scheme?**

  In terms of compression ratio, we outperform ANS, Bitcomp, Cascaded, and GPULZ consistently for all datasets. For TPC-H and DBText, we achieve slightly higher compression ratios than LZ4 and Snappy, while they have a higher compression ratio for the GDelt location data. When considering overall compression throughput, we outperform GPULZ and all nvCOMP compressors except for ANS, Bitcomp, Cascaded, and all compressors generated with the LC framework. This ranges from a 2.8x increase compared to Snappy to a 7.9x increase compared to ZSTD. Overall, our compressor is part of the Pareto front for every dataset, pushing the state-of-the-art further towards ideal compression. We achieve nearly identical compression ratios to FSST, except for the machine-readable data, while achieving a speedup of 42.06x. Even when compared to a multithreaded CPU implementation, we achieve a 6.45x speedup.

In conclusion, we have introduced a GPU-accelerated compression pipeline that is on the Pareto front and therefore pushes the state-of-the-art. This compression pipeline is based on the FSST compressor and has a throughput 42.06 times that of the single-threaded version and 6.45 times that of the multithreaded implementation. We achieved a compression throughput of 74 GB/s with a compression ratio of 3.13 for the TPC-H dataset. Combined with the fact that we have very low memory consumption, we are an attractive option for modern GPU-based data applications. Our compressor outperforms an uncompressed transmission on links with a bandwidth up to 50 GB/s when omitting decompression, making our pipeline a feasible option for both memory offloading and data shuffling. Furthermore, we have integrated our compression pipeline with the GSST decompressor and achieved a combined throughput of 55 GB/s. Concretely, this means we outperform uncompressed data transfer on links with a bandwidth up to 37.5 GB/s, considering a complete transmission consisting of compression, transfer, and decompression. The source code of this thesis, including all compressor versions and the integrated version with GSST, will be available on GitHub [6].

6.2. Discussion and future work

We have achieved all our goals for this thesis, but there are always points to improve. In this section, we will elaborate on some topics that we believe can further enhance the quality of the resulting compression pipeline. We believe there are two types of general improvements. We have concrete technical improvements that either potentially enhance performance or directly contribute to making the resulting pipeline ready for a production system. Additionally, additional experiments would further establish our pipeline's effectiveness for a broader range of datasets and applications. We will first discuss some (concrete) technical improvements, and then focus on future work that can potentially enhance the robustness of the compression pipeline and experimental results.

6.2.1. Technical improvements

In Section 5.4.2, we observed that more warps per thread block positively affect the gathering stage. This makes perfect sense, as gathering a few large blocks is more efficient than gathering many smaller blocks. At the end of the encoding stage, all warps within a thread block will synchronize and ensure they have flushed the same number of times. This allows the gathering and compaction stages to treat the output of several warps as a single block, which enhances overall performance. However,

adding more warps to a thread block will eventually decrease the throughput of the encoding stage. This is because the shared memory usage of a block grows linearly with the number of warps, and the occupancy decreases as a result. One possibility to mitigate this is to use cooperative groups to allow warps in different thread blocks to synchronize their number of flushes. This decouples the shared memory pressure from the number of warps that can synchronize, potentially increasing the overall throughput. However, the amount of padding will increase when more warps synchronize their number of flushes, which will lower the throughput of stream compaction. Some experimentation will have to be performed to find the optimal amount of synchronization between warps.

In the same section, we have analyzed the throughput per stage as a function of the work division. This shows that the throughput of our encoding and gathering stage primarily determines our overall performance. Unfortunately, these stages have somewhat conflicting preferences in terms of the number of blocks and their lengths. The main requirement for the encoding stage is that there are enough blocks to prevent idle SMs and low occupancy, while the gathering block prefers fewer blocks. When the input size changes, so will the optimal work division. For our thesis, we used static parameters that are optimized for our input size. Ideally, these parameters are updated based on the input size. This would require some minor modifications to the encoding kernel, and experiments will have to be performed to define a function that transforms the input size to ideal parameters. Ideally, this function would also take the underlying hardware into account.

We filter out a special character (`0xFE`) in our current pipeline since we use it as temporary padding. While this character is not used in regular ASCII text and is also an unused character in UTF-8, it can still technically be used by non-textual data or textual data using special characters. This character should not be filtered if it is present in the input data to maintain the property that the compressor is a lossless compressor. In the current implementation, we enforce this property by rejecting data with a reserved character, as we deemed a mitigation strategy for this limitation to have an insignificant effect on the final results. However, this limitation must be solved before the compressor can be used in a production-ready system. It can be mitigated by introducing two fixed entries in the symbol table: one mapping from the reserved character (`0xFE`) to any other character, and one reserved mapping that matches nothing and maps to the reserved character. This achieves the goal of mapping all occurrences of the reserved character to an unused character, and ensuring that there cannot exist a reserved character that is not padding. The cost of this mitigation will be two symbols, reducing the available symbols from 255 to 253. In Section 3.3, we have already observed that the symbol table is not fully utilized, so this mitigation strategy will have minimal impact on the compression ratio.

In Section 3.7.4, we introduced pipelining the transposition stage. However, when using the compressor to compress data that is not present on the GPU, pipelining can also be used to hide the transfer latency from system memory to GPU global memory. This means the GPU can start compressing part of the data when the transfer is completed. Additionally, this allows the GPU to compress data streams that do not fit in memory.

Finally, in Chapter 4, we provide the steps to integrate GSST with our compressor. However, the GSST kernel contains some inefficiencies that can be addressed to potentially further increase performance. This involves removing the running length mechanism, at the cost of using a fixed mapping between data blocks and CUDA thread blocks. Additionally, the overall robustness of GSST should be improved to support more diverse datasets and at least match the capabilities of our compressor. Only then can the combined (de)compressor be used in production-ready workloads.

6.2.2. Enhancing robustness

Our development system uses an RTX 2070, and our benchmarking system uses an RTX 4090. While these GPUs are commonly used, many more exist, and new ones are developed every few years. In our thesis, we focused on the hardware accessible to us and tuned our parameters to this hardware as a result. It would be interesting to see the effect of these parameters on different consumer and server-grade GPUs. This would tell us something about our performance portability, i.e., how well our results transfer to different hardware architectures. The results of these experiments can also be used to dynamically adapt parameters depending on the GPU used.

In theory, our pipeline can be used for any data processing application. However, the main target application is big data analytics, since these applications handle many terrabytes of data and require high-throughput compression on GPUs. It would be very interesting to integrate our (de)compression pipeline into a library such as RAPIDS [64] to perform direct comparisons between our compressor and

the state-of-the-art in the context of data analytics.

Bibliography

[1] Daniel Abadi et al. “The design and implementation of modern column-oriented database systems”. In: *Foundations and Trends® in Databases* 5.3 (2013), pp. 197–280.

[2] Andy Adinets. *CUDA Dynamic Parallelism API and Principles*. 2014. URL: https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/.

[3] Azim Afroozeh and Peter Boncz. “The fastlanes compression layout: Decoding> 100 billion integers per second with scalar code”. In: *Proceedings of the VLDB Endowment* 16.9 (2023), pp. 2132–2144.

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

[5] Nauman Ahmed et al. “GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data”. In: *BMC bioinformatics* 20 (2019), pp. 1–20.

[6] Tim Anema. *Thesis Git Repository*. 2025. URL: https://github.com/timanema/msc-thesis-public.

[7] Michael Armbrust et al. “Spark sql: Relational data processing in spark”. In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

[8] Zaid Al-Ars, Saiyi Wang, and Hamid Mushtaq. “SparkRA: enabling big data scalability for the GATK RNA-seq Pipeline with Apache Spark”. In: *Genes* 11.1 (2020), p. 53.

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

[10] Noushin Azami et al. *LC Git Repository*. 2024. URL: https://github.com/burtscher/LC-framework.

[11] Michael Bauer, Henry Cook, and Brucek Khailany. “CudaDMA: optimizing GPU memory bandwidth via warp specialization”. In: *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. 2011, pp. 1–11.

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

[13] Peter Boncz, Thomas Neumann, and Viktor Leis. “FSST: fast random access string compression”. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2649–2661.

[14] Peter A Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution.” In: *Cidr*. Vol. 5. 2005, pp. 225–237.

[15] Sebastian Breß. “The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS”. In: *Datenbank-Spektrum* 14 (2014), pp. 199–209.

[16] Sebastian Breß et al. “GPU-accelerated database systems: Survey and open challenges”. In: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV: Selected Papers from ADBIS 2013 Satellite Events* (2014), pp. 1–35.

[17] Michael Burrows. “A block-sorting lossless data compression algorithm”. In: *SRS Research Report* 124 (1994).

[18] Franck Cappello et al. “Use cases of lossy compression for floating-point data in scientific data sets”. In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1201–1220.

[19] Periklis Chrysogelos et al. “HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines”. In: *Proceedings of the VLDB Endowment* 12.5 (2019), pp. 544–556.

[20] U Cisco. “Cisco annual internet report (2018–2023) white paper”. In: *Cisco: San Jose, CA, USA* 10.1 (2020), pp. 1–35.

[21] Yann Collet. *LZ4 v1.10.0 - Multicores edition*. 2024. URL: https://github.com/lz4/lz4/releases/tag/v1.10.0.

[22] Gordon V. Cormack and R Nigel S Horspool. “Data compression using dynamic Markov modelling”. In: *The Computer Journal* 30.6 (1987), pp. 541–550.

[23] Voltron data. *Theseus, The Enterprise SQL Engine*. URL: https://voltrondata.com/theseus.html.

[24] Peter Deutsch. *DEFLATE compressed data format specification version 1.3*. 1996.

[25] Jarek Duda et al. “The use of asymmetric numeral systems as an accurate replacement for Huffman coding”. In: *2015 Picture Coding Symposium (PCS)*. IEEE. 2015, pp. 65–69.

[26] Joao Ferreira et al. “Estimating the environmental impact of data centers”. In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2018, pp. 1–4.

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

[28] Philip Gage. “A new algorithm for data compression”. In: *The C Users Journal* 12.2 (1994), pp. 23–38.

[29] Abhijeet Gaikwad and Ioane Muni Toke. “GPU based sparse grid technique for solving multi-dimensional options pricing PDEs”. In: *Proceedings of the 2nd workshop on high performance computational finance*. 2009, pp. 1–9.

[30] GDELT. *The GDelt project*. URL: https://www.gdeltproject.org/.

[31] Jens Glaser et al. “High-throughput virtual laboratory for drug discovery using massive datasets”. In: *The International Journal of High Performance Computing Applications* 35.5 (2021), pp. 452–468.

[32] Jens Glaser et al. “Scaling SQL to the Supercomputer for Interactive Analysis of Simulation Data”. In: *Smoky Mountains Computational Sciences and Engineering Conference*. Springer. 2021, pp. 327–339.

[33] Solomon Golomb. “Run-length encodings (corresp.)” In: *IEEE transactions on information theory* 12.3 (1966), pp. 399–401.

[34] Chris Gregg and Kim Hazelwood. “Where is the data? Why you cannot debate CPU vs. GPU performance without the answer”. In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE. 2011, pp. 134–144.

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

[36] Mark Harris. *How to Overlap Data Transfers in CUDA C/C++*. 2012. URL: https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/.

[37] Mark Harris and Kyrylo Perelygin. *Cooperative Groups: Flexible CUDA Thread Programming*. 2017. URL: https://developer.nvidia.com/blog/cooperative-groups/.

[38] Mark Harris and Kyrylo Perelygin. *Cooperative Groups: Flexible CUDA Thread Programming*. 2017. URL: https://developer.nvidia.com/blog/cooperative-groups/.

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

[40] HEAVY.AI. *HeavyDB, A Revolutionary GPU-Accelerated Database & Analytics Platform*. URL: https://www.heavy.ai/product/heavydb.

[41] Holger Hellmuth and Joern Klauke. “POWER NX842 Compression for Db2”. In: *IBM Corporation, Sep* (2017).

[42] Matthew Herland, Taghi M Khoshgoftaar, and Richard A Bauder. “Big data fraud detection using multiple medicare data sources”. In: *Journal of Big Data* 5.1 (2018), pp. 1–21.

[43] Benjamín Hernández et al. “Performance evaluation of python based data analytics frameworks in summit: Early experiences”. In: *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI: 17th Smoky Mountains Computational Sciences and Engineering Conference, SMC 2020, Oak Ridge, TN, USA, August 26-28, 2020, Revised Selected Papers 17*. Springer. 2020, pp. 366–380.

[44] Martin Hilbert and Priscila López. “The world’s technological capacity to store, communicate, and compute information”. In: *science* 332.6025 (2011), pp. 60–65.

[45] David A Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[46] David Meirion Hughes et al. “Ink-compact: In-kernel stream compaction and its application to multi-kernel data visualization on general-purpose gpus”. In: *Computer Graphics Forum*. Vol. 32. 6. Wiley Online Library. 2013, pp. 178–188.

[47] Kurt Jacobson et al. “Music personalization at Spotify”. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. 2016, pp. 373–373.

[48] Hosagrahar V Jagadish et al. “Big data and its technical challenges”. In: *Communications of the ACM* 57.7 (2014), pp. 86–94.

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

[50] P Kavitha. “A survey on lossless and lossy data compression methods”. In: *International Journal of Computer Science & Engineering Technology* 7.03 (2016), pp. 110–114.

[51] Fabian Knorr, Peter Thoman, and Thomas 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*. 2021, pp. 1–14.

[52] Alexandros Labrinidis and Hosagrahar V Jagadish. “Challenges and opportunities with big data”. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 2032–2033.

[53] N Jesper Larsson and Alistair Moffat. “Off-line dictionary-based compression”. In: *Proceedings of the IEEE* 88.11 (2000), pp. 1722–1732.

[54] Carson K Leung et al. “Big data analytics for personalized recommendation systems”. In: *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE. 2019, pp. 1060–1065.

[55] Jing Li et al. “Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics”. In: *Proceedings of the VLDB Endowment* 9.14 (2016), pp. 1647–1658.

[56] Shancang Li, Li Da Xu, and Shanshan Zhao. “5G Internet of Things: A survey”. In: *Journal of Industrial Information Integration* 10 (2018), pp. 1–9.

[57] Fangzheng Lin et al. “Recoil: Parallel rans decoding with decoder-adaptive scalability”. In: *Proceedings of the 52nd International Conference on Parallel Processing*. 2023, pp. 31–40.

[58] Yuan Lin and Vinod Grover. *Using CUDA Warp-Level Primitives*. 2018. URL: `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`.

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

[60] Matt Mahoney. *Large Text Compression Benchmark*. 2024. URL: `https://www.mattmahoney.net/dc/text.html`.

[61] Matthew V Mahoney. “Adaptive weighing of context models for lossless data compression”. In: (2005).

[62] Sergey Melnik et al. “Dremel: interactive analysis of web-scale datasets”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 330–339.

[63] Vukasin Milovanovic, Devavret Makkar, and Gregory Kimball. *Boosting Data Ingest Throughput with GPUDirect Storage and RAPIDS cuDF*. 2022. URL: https://developer.nvidia.com/blog/boosting-data-ingest-throughput-with-gpudirect-storage-and-rapids-cudf/.

[64] Vukasin Milovanovic, Devavret Makkar, and Gregory Kimball. *Boosting Data Ingest Throughput with GPUDirect Storage and RAPIDS cuDF*. 2022. URL: https://developer.nvidia.com/blog/boosting-data-ingest-throughput-with-gpudirect-storage-and-rapids-cudf/.

[65] Maryam Moazeni, Alex Bui, and Majid Sarrafzadeh. “Accelerating total variation regularization for matrix-valued images on GPUs”. In: *Proceedings of the 6th ACM conference on Computing frontiers*. 2009, pp. 137–146.

[66] Alistair Moffat. “Implementing the PPM data compression scheme”. In: *IEEE Transactions on communications* 38.11 (1990), pp. 1917–1921.

[67] Mark Nelson and Jean-Loup Gailly. “The data compression book 2nd edition”. In: *M & T Books, New York, NY* (1995).

[68] Craig G Nevill-Manning and Ian H Witten. “Identifying hierarchical structure in sequences: A linear-time algorithm”. In: *Journal of Artificial Intelligence Research* 7 (1997), pp. 67–82.

[69] Lennart Noordsij et al. “Parallelization of variable rate decompression through metadata”. In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2020, pp. 245–252.

[70] NVIDIA. *Cascaded Compression*. URL: https://docs.nvidia.com/cuda/nvcomp/cascaded.html.

[71] NVIDIA. *nvCOMP library*. URL: https://developer.nvidia.com/nvcomp.

[72] NVIDIA. *nvCOMP performance*. 2024. URL: https://web.archive.org/web/20240225035645/https://developer.nvidia.com/nvcomp.

[73] NVIDIA. *NVIDIA Ampere GPU and Ada Instruction Set*. URL: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#nvidia-ampere-gpu-and-ada-instruction-set.

[74] NVIDIA. *Nvidia Blackwell Architecture Technical Overview*. 2024. URL: https://resources.nvidia.com/en-us-blackwell-architecture.

[75] NVIDIA. *NVIDIA RTX ADA GPU ARCHITECTURE*. 2022. URL: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf.

[76] NVIDIA. *NVIDIA RTX BLACKWELL GPU ARCHITECTURE*. 2024. URL: https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf.

[77] NVIDIA. *NVLINK*. URL: https://www.nvidia.com/en-us/data-center/nvlink/.

[78] NVIDIA. *Thrust documentation - copy_if*. URL: https://nvidia.github.io/cccl/thrust/api/function_group__stream__compaction_1gafd4cd96b998ad2b3c336be1e24dc1f67.html.

[79] Molly A O’Neil and Martin 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*. 2011, pp. 1–7.

[80] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. “Hybrid transactional/analytical processing: A survey”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1771–1775.

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

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

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

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

[85] Jeongmin Park et al. “CODAG: Characterizing and Optimizing Decompression Algorithms for GPUs”. In: *arXiv preprint arXiv:2307.03760* (2023).

[86] Himali Patel et al. “Survey of lossless data compression algorithms”. In: *International Journal of Engineering Research and Technolgy* 4.4 (2015), pp. 926–929.

[87] Ritesh A Patel et al. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[88] Max Plauth and Andreas Polze. “GPU-based decompression for the 842 algorithm”. In: *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. 2019, pp. 97–102.

[89] Meikel Poess and Raghunath Othayoth Nambiar. “Energy cost, the key challenge of today’s data centers: a power consumption analysis of TPC-C results”. In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1229–1240.

[90] Joachim Probst et al. “Optical coherence tomography with online visualization of more than seven rendered volumes per second”. In: *Journal of biomedical optics* 15.2 (2010), pp. 026014–026014.

[91] Dik Reinoud. *Videokaartbenchmarks - Synthetische tests: 3DMark*. 2025. URL: `https://tweakers.net/best-buy-guide/videokaarten/benchmarks#synthetisch`.

[92] David Reinsel-John Gantz-John Rydning, John Reinsel, and John Gantz. “The digitization of the world from edge to core”. In: *Framingham: International Data Corporation* 16 (2018), pp. 1–28.

[93] John Rydning and Michael Shirer. “Data creation and replication will grow at a faster rate than installed storage capacity, according to the IDC global DataSphere and StorageSphere forecasts”. In: *International Data Corporation* (2021).

[94] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[95] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. “A study of the fundamental performance characteristics of GPUs and CPUs for database analytics”. In: *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 2020, pp. 1617–1632.

[96] Anil Shanbhag et al. “Tile-based lightweight integer compression in GPU”. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 1390–1403.

[97] K Shyni and Manoj Kumar KV. “Lossless LZW data compression algorithm on CUDA”. In: (2013).

[98] Md Abu Bakar Siddik, Arman Shehabi, and Landon Marston. “The environmental footprint of data centers in the United States”. In: *Environmental Research Letters* 16.6 (2021), p. 064017.

[99] Vaclav Simek and Ram Rakesh Asn. “Gpu acceleration of 2d-dwt image compression in matlab with cuda”. In: *2008 Second UKSIM European Symposium on Computer Modeling and Simulation*. IEEE. 2008, pp. 274–277.

[100] Evangelia Sitaridi et al. “Massively-parallel lossless data decompression”. In: *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE. 2016, pp. 242–247.

[101] Przemyslaw Skibinski. *lzbench*. 2025. URL: `https://github.com/inikep/lzbench`.

[102] Samuel S Stone et al. “Accelerating advanced MRI reconstructions on GPUs”. In: *Proceedings of the 5th conference on Computing frontiers*. 2008, pp. 261–272.

[103] James A Storer and Thomas G Szymanski. “Data compression via textual substitution”. In: *Journal of the ACM (JACM)* 29.4 (1982), pp. 928–951.

[104] Bharat Sukhwani et al. “High-throughput, lossless data compresion on FPGAs”. In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2011, pp. 113–116.

[105] Matthieu Tardy and Carter Edwards. *Controlling Data Movement to Boost Performance on the NVIDIA Ampere Architecture*. 2020. URL: `https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/`.

[106] Techpowerup. *NVIDIA GeForce RTX 4090*. 2022. URL: https://www.techpowerup.com/gpu-specs/geforce-rtx-4090.c3889.

[107] Techpowerup. *NVIDIA GeForce RTX 5090*. 2024. URL: https://www.techpowerup.com/gpu-specs/geforce-rtx-5090.c4216.

[108] Techpowerup. *NVIDIA GeForce RTX 5090 Founders Edition Review - The New Flagship*. 2025. URL: https://www.techpowerup.com/review/nvidia-geforce-rtx-5090-founders-edition/33.html.

[109] Adam Thompson and CJ Newburn. *GPUDirect Storage: A Direct Path Between Storage and GPU Memory*. 2019. URL: https://developer.nvidia.com/blog/gpudirect-storage.

[110] Ashish Thusoo et al. "Hive: a warehousing solution over a map-reduce framework". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[111] TPC. *TPC-H Version 2 and Version 3*. URL: https://www.tpc.org/tpch/.

[112] Kalyan Veeramachaneni et al. "AI^ 2: training a big data machine to defend". In: *2016 IEEE 2nd international conference on big data security on cloud (BigDataSecurity), IEEE international conference on high performance and smart computing (HPSC), and IEEE international conference on intelligent data and security (IDS)*. IEEE. 2016, pp. 49–54.

[113] Robin Vonk. "GSST: High Throughput Parallel String Decompression on GPU". MA thesis. TU Delft, 2024.

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

[115] Yuuki Watanabe and Toshiki Itagaki. "Real-time display on Fourier domain optical coherence tomography system using a graphics processing unit". In: *Journal of Biomedical Optics* 14.6 (2009), pp. 060506–060506.

[116] André Weißenberger and Bertil Schmidt. "Massively parallel ans decoding on gpus". In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.

[117] André Weißenberger and Bertil Schmidt. "Massively parallel inverse block-sorting transforms for bzip2 decompression on GPUs". In: *Proceedings of the 53rd International Conference on Parallel Processing*. 2024, pp. 856–865.

[118] Wikipedia. *Roofline model*. 2025. URL: https://en.wikipedia.org/w/index.php?title=Roofline_model&oldid=1280397259.

[119] Frans MJ Willems, Yuri M Shtarkov, and Tjalling J Tjalkens. "The context-tree weighting method: Basic properties". In: *IEEE transactions on information theory* 41.3 (1995), pp. 653–664.

[120] Samuel Webb Williams, Andrew Waterman, and David A Patterson. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. rep. Technical Report UCB/EECS-2008-134, EECS Department, University of …, 2008.

[121] Ian H Witten, Radford M Neal, and John G Cleary. "Arithmetic coding for data compression". In: *Communications of the ACM* 30.6 (1987), pp. 520–540.

[122] Haicheng Wu et al. "Kernel weaver: Automatically fusing database primitives for efficient gpu computation". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2012, pp. 107–118.

[123] Annie Yang et al. "MPC: a massively parallel compression algorithm for scientific data". In: *2015 IEEE International Conference on Cluster Computing*. IEEE. 2015, pp. 381–389.

[124] Zeyu Yang, Karel Adamek, and Wesley Armour. "Part-time Power Measurements: nvidia-smi's Lack of Attention". In: *arXiv preprint arXiv:2312.02741* (2023).

[125] Bobbi W Yogatama, Weiwei Gong, and Xiangyao Yu. "Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS". In: *Proceedings of the VLDB Endowment* 15.11 (2022), pp. 2491–2503.

[126] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. "The Yin and Yang of processing data warehousing queries on GPU devices". In: *Proceedings of the VLDB Endowment* 6.10 (2013).

[127] Matei Zaharia et al. “Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing”. In: *9th USENIX symposium on networked systems design and implementation (NSDI 12)*. 2012, pp. 15–28.

[128] Boyuan Zhang et al. “Gpulz: Optimizing lzss lossless compression for multi-byte data on modern gpus”. In: *Proceedings of the 37th International Conference on Supercomputing*. 2023, pp. 348–359.

[129] Kang Zhang and Jin U Kang. “Graphics processing unit accelerated non-uniform fast Fourier transform for ultrahigh-speed, real-time Fourier-domain OCT”. In: *Optics express* 18.22 (2010), pp. 23472–23487.

[130] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.

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

A

ADMS Paper

This appendix includes the version of the paper submitted to ADMS 2025.

High Throughput GPU-Accelerated FSST String Compression

Tim Anema[1], Joost Hoozemans[2], Zaid Al-Ars[1], H. Peter Hofstee[1,3]
[1]Delft University of Technology, Delft, Netherlands; [2]Voltron Data, US; [3]IBM, Austin, TX, US
tim.anema@hotmail.nl

ABSTRACT

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

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

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

PVLDB Reference Format:
Tim Anema[1], Joost Hoozemans[2], Zaid Al-Ars[1], H. Peter Hofstee[1,3]. High Throughput GPU-Accelerated FSST String Compression. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

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

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

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

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

This paper has the following contributions:

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

The paper is organized as follows. We will touch upon related work and general GPU development background in Section 2. We will then analyze the acceleration potential of FSST and possible inhibitors in Section 3. We will then provide a memory-efficient encoding table in Section 4, and use it in the encoding kernel implementation in Section 5. The overall compression pipeline will be discussed in Section 6, and we will evaluate its performance in Section 7. Finally, we will conclude in Section 8 and discuss some potential future work.

2 BACKGROUND

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

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

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

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

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

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

2.1 FSST

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

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

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

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

2.2 GSST

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

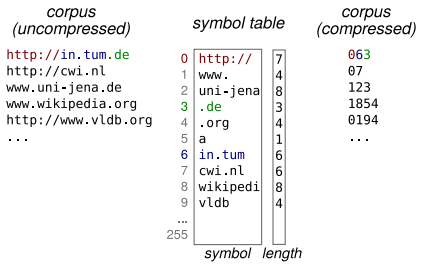


Figure 1: An example of FSST compression. The uncompressed data is encoded to a (smaller) format using a static dictionary. Source: [6]

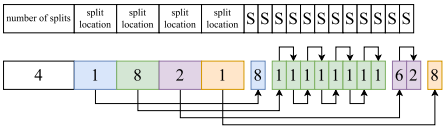


Figure 2: The split format GSST uses. Every block is divided into splits, which individual threads will process. Source: [36]

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

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

2.3 GPU development

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

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

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

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

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

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

3 ACCELERATION POTENTIAL OF FSST

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

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

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

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

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

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

4 MEMORY-EFFICIENT ENCODING TABLE

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

4.1 Data properties

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

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

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

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

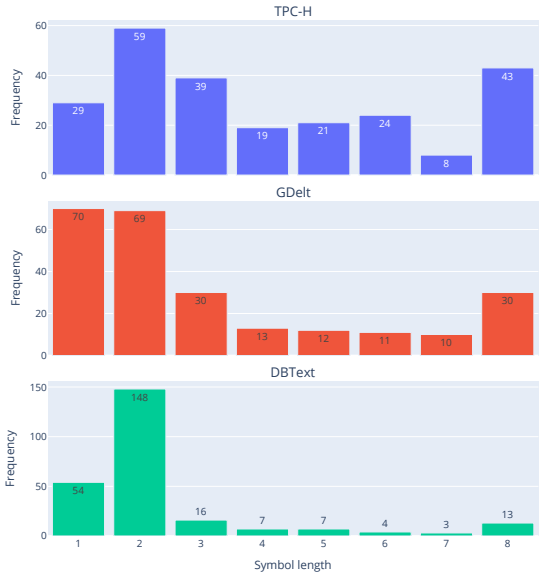


Figure 3: This histogram shows the spread of symbols regarding their length for three datasets: TPC-H, GDelt, and DBText. In general, we can see that the DBText corpus heavily uses short symbols, while the other datasets also use longer symbols more often.

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

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

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

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

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

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

4.2 Modifying the hashtable

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

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

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

4.3 Efficient short symbols

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

4.3.1 ELL matrix. One data structure that could more efficiently represent this data pattern is an ELL matrix based on the sparse matrix package in ELLPACK [14]. The original matrix can be changed to a NxK matrix, where K is the new number of columns, and all non-zero elements within a row are compacted. While the ELL format leads to a significant reduction in size, the matrix is still

sparse, storing more than 200 empty rows. Additionally, a GPU uses 32 banks to address shared memory, meaning a single bank will serve eight rows of this matrix, likely leading to bank conflicts as the characters used in textual data are in close proximity.

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

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

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

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

5 ENCODING KERNEL DESIGN

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

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

5.1 Applying tiling

After the tables have been created, the encoding stage will start. Encoding is done on a block level, i.e., every FSST block can be

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

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

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

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

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

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

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

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

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

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

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

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

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

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

encoded separately. This is the first level of parallelism and maps fairly naturally to a CUDA thread block. To create parallelism within a (thread) block, we utilize the tiling technique. We will split the data within a block into multiple tiles, which map to a single thread. This means a single thread works on a small contiguous block of memory, which is part of the original data block, and all threads in the thread block work in parallel to encode a single data block.

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

5.2 Inter-block dependencies

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

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

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

5.3 Sliding window

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

As mentioned before, a naive implementation performing byte-level operations leads to many bank conflicts and underutilizes the shared memory banks, which are capable of 32 bits per clock cycle.

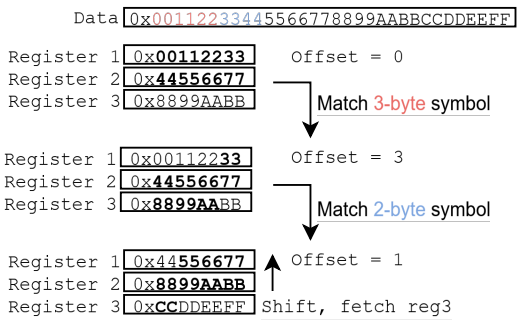


Figure 4: The process of using a sliding window to build a view of the active data, which can be used by the encoding kernel to directly match on. In this example, we show how data moves through the registers as the data in shared memory is processed. Bold numbers are used to show what part of the data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

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

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

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

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

We mitigate this by requesting 32 bits, or four characters, at a time from shared memory, and we also organize the input buffer as a column-major matrix. This means we view the input data for a thread block as a NxM matrix, where N represents the number of threads (or tiles) within a thread block and M the number of 4-byte integers representing the data of a single tile. Shared memory will then contain a $X * N$ matrix, where X represents the chunk size. All data for a single tile will be stored in a column in this matrix, completely eliminating bank conflicts.

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

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

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

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

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

Listing 3: The output packing process

5.4 Output packing

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

5.5 Ensuring coalesced writes

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

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

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

After every iteration in the encoding cycle, threads will hold a vote on whether to initiate a flush or not. If any thread risks overrunning its buffer, all threads will add padding to their local buffer if needed and trigger a flush. A flush will also be triggered if

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

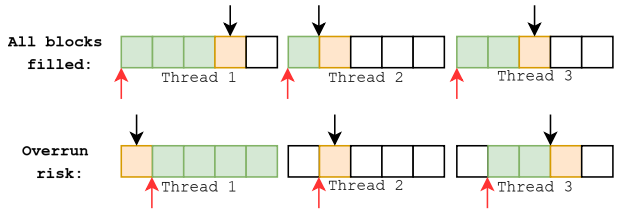


Figure 5: Threads keep track of their own local buffer head (marked with black arrows, on byte level), and their working block (marked orange) and filled blocks (marked green). All threads keep track of the active block in the warp (marked with red arrows, on block level). Threads in a warp will decide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

all threads have filled the currently active block, which is the ideal scenario. After the last encoding cycle has completed, a warp will continue flushing its buffers until all threads within a warp have fully written their data. Additionally, all warps within a block will communicate such that they perform the same number of overall flushes to create a valid output matrix.

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

6 COMPRESSION PIPELINE

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

6.1 Gathering data

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

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

6.2 Improving output format

The compression pipeline is now complete, but still has two issues. Both issues are caused by the transposed format. The first issue is that we (partially) lose FSST's ability to perform random access

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

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

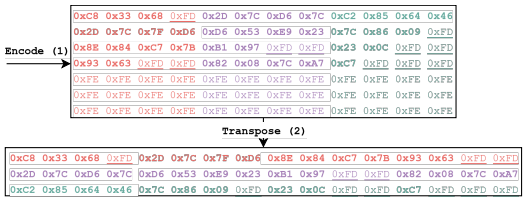
In addition to this, we can now fix the second issue by performing stream compaction on the transposed data to remove the interleaved padding. Since the encoded data for a single tile is now in contiguous memory, we no longer need to maintain identical output lengths for all tiles.

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

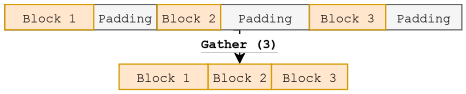
6.3 Optimized pipeline

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

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



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



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

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

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

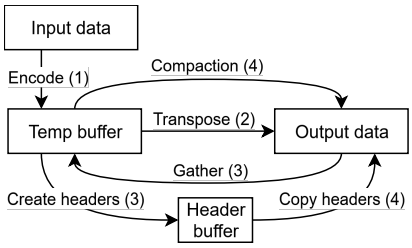


Figure 7: The data flow through the temporary and destination buffers in our pipeline. The overall memory usage is low because we reuse the temporary and destination buffers for multiple operations.

6.4 Ensuring compatibility with GSST

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

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

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

7 EVALUATION

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

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

7.1 Encoding table performance

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

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

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

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

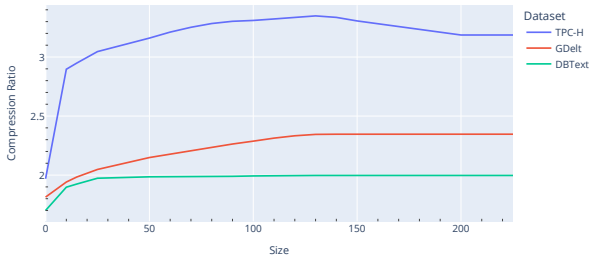


Figure 8: The effect of varying the number of entries in the hashtable on the resulting compression ratio. It is clear that the hash table can be smaller without sacrificing significant accuracy.

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

Figure 10 shows the effect on overall pipeline throughput as a result of higher shared memory usage when modifying the number of columns. When combined with the effect on the compression ratio shown in Figure 9, these results suggest that a throughput reduction of approximately 3 percent across all datasets leads to an approximate 2 percent increase in compression ratio for machine-readable data.

7.2 Accelerated compression throughput and ratio

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

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

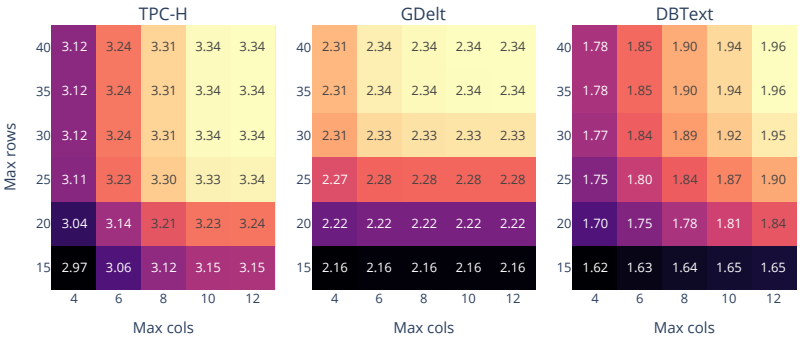


Figure 9: The effect of varying the number of rows and columns in the lookup table on the resulting compression ratio.

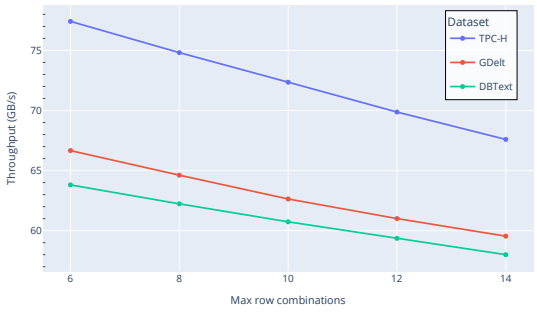


Figure 10: The effects on overall throughput of changing the maximum number of columns, as a consequence of lower occupancy.

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

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

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

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

7.3 Performance analysis per stage

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

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

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

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

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

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

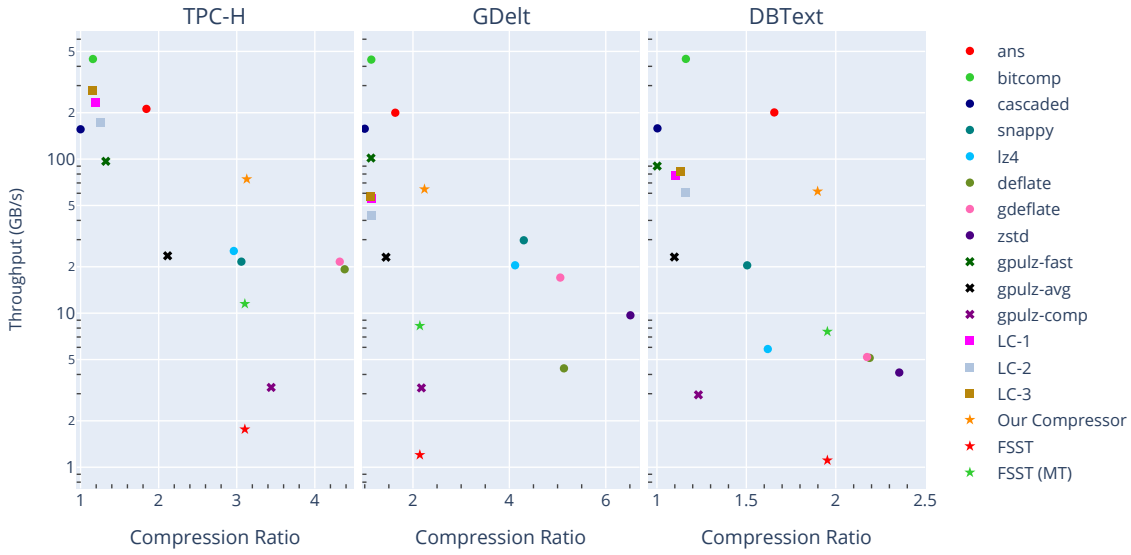


Figure 11: We compare our proposed compression pipeline to the nvCOMP compressors, GPULZ, compressors generated with the LC framework with a different number of stages, and the original FSST algorithm, regarding compression ratio and throughput. All benchmarks were completed on the same machine (RTX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

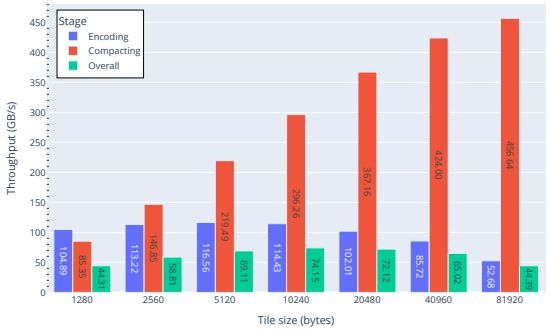


Figure 12: The throughput for the two major stages (encoding and compaction) and how they are influenced by the tile size.

7.4 GPU memory consumption

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

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

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

8 CONCLUSION AND FUTURE WORK

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

This positions our compressor as a Pareto optimal compressor that can be used in GPU-accelerated database systems and other

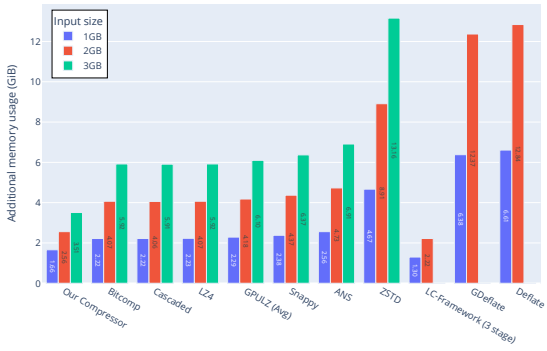


Figure 13: Memory usage, measured with `nvidia-smi`, excluding input and output buffers of our compression pipeline, compared to other state-of-the-art compressors. The compressors are sorted based on their memory usage, and empty columns indicate that a compressor ran out of memory or failed to compress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

areas where large amounts of textual data need to be efficiently compressed with competitive compression ratios.

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

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] TPC. [n.d.]. *TPC-H Version 2 and Version 3.* https://www.tpc.org/tpch/
[36] Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. 2025. GSST: Parallel string decompression at 191 GB/s on GPU. In *Proceedings Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (Rotterdam, Netherlands). ACM.
[37] Zheng Wang, Chenxi Wang, and Lei Wang. 2023. Dpubench: An application-driven scalable benchmark suite for comprehensive dpu evaluation. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 3, 2 (2023), 100120.
[38] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Martin Swany, Dingwen Tao, and Franck Cappello. 2023. Gpulz: Optimizing lzss lossless compression for multi-byte data on modern gpus. In *Proceedings of the 37th International Conference on Supercomputing*. 348–359.
[39] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.
[40] Yuan Zu and Bei Hua. 2014. GLZSS: LZSS lossless data compression can be faster. In *Proceedings of Workshop on General Purpose Processing Using GPUs*. 46–53.