G-CasDec: General Cascaded Decompression on GPUs

YONGQI ZHUO*, Tsinghua University, China
XINYU ZENG*, Tsinghua University, China
HUANCHEN ZHANG, Tsinghua University, China and Shanghai Qi Zhi Institute, China
MINGYU GAO, Tsinghua University, China and Shanghai Qi Zhi Institute, China

GPU-accelerated analytical query processing is often limited by both GPU device memory capacity and host-to-device data transfer time. Modern data compression techniques, such as cascaded lightweight compression, can mitigate these issues. However, existing designs all exhibit critical tradeoffs on compression ratios, decompression speed, and extensibility. They either apply aggressive compression without addressing execution irregularity, or maintain strictly aligned data layouts that sacrifice compressibility. Also, none of them provides an extensible way to easily add new schemes. To address these issues, we present G-CasDec, an optimized framework to enable efficient decompression on GPUs for general cascaded compression schemes. G-CasDec incorporates several key techniques, including aligned batching for aggressive compression of variable-length data yet with efficient decompression, shared memory reuse and register-level fusion to improve resource utilization, and a set of bytecode instructions to orchestrate a generic decompression flow. G-CasDec outperforms state-of-the-art designs with 2.8× higher compression ratios and 2.0× faster decompression, and is also extensible to arbitrary cascade schemes.

CCS Concepts: • **Information systems** → **Data compression**; *Database query processing*.

Additional Key Words and Phrases: GPU data compression; cascaded compression; GPU data analytics

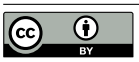
ACM Reference Format:
Yongqi Zhuo, Xinyu Zeng, Huanchen Zhang, and Mingyu Gao. 2026. G-CasDec: General Cascaded Decompression on GPUs. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 276 (September 2026), 26 pages. https://doi.org/10.1145/3837114

1 Introduction

The rapid ascent of Graphics Processing Units (GPUs) over the last decade has fundamentally redefined the landscape of data-intensive computation, with their capabilities expanding at a rate that significantly outpaces the CPUs. This explosion in parallel processing power has made GPUs promising for data analytics, with GPU-accelerated query processing systems emerging in the last decade in both academia [11, 13, 27, 34, 35] and industry [8, 9, 14, 30]. However, GPU analytics is often limited by several issues. First, the GPU device memory capacity is much smaller than the CPU host memory, e.g., only 32 GB on NVIDIA RTX 5090. When the dataset size exceeds this limit, offloading data to the host side incurs significant performance cost and programming complexity. Second, even if the dataset fits in the GPU memory, sometimes it still needs to be transferred from the CPU over the slow PCIe bus when integrated into an end-to-end data processing pipeline. Third,

*Equal contribution.

Authors' Contact Information: Yongqi Zhuo, Tsinghua University, Beijing, China, zhuoyq21@mails.tsinghua.edu.cn; Xinyu Zeng, Tsinghua University, Beijing, China, zeng-xy21@mails.tsinghua.edu.cn; Huanchen Zhang, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China, huanchen@tsinghua.edu.cn; Mingyu Gao, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China, gaomy@tsinghua.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/9-ART276
https://doi.org/10.1145/3837114

some queries access a large amount of data, making them bound by GPU memory accesses. These limitations motivate us to explore *data compression* for GPU analytics, where the data are stored in a highly compressed format and decompressed on the GPU before being processed. Compression not only allows larger datasets to fit within the GPU memory, but also accelerates PCIe transfers and GPU memory accesses in inline query execution by moving less data around.

The state-of-the-art compression paradigm for GPU analytics follows a *cascaded* approach [22, 28], where lightweight compression algorithms are recursively applied on compressed data to further increase compressibility. However, we observe that existing cascaded decompression frameworks face several critical tradeoffs. Early work like Tile-Based [28] directly ported cascaded compression from CPUs without considering GPU-specific factors like control flow overheads and memory access alignment. Consequently, although it achieved state-of-the-art data compression ratios, its **decompression speed** on the GPU was quite low, even underperforming the uncompressed design. Later designs, represented by FastLanes [3–5], introduced a set of parallel-friendly optimizations to improve decompression parallelism and memory bandwidth utilization. But this performance boost came with a significant sacrifice in **compression ratios**. For example, some compression algorithms would generate compressed data in shorter yet variable lengths than the input. FastLanes decided not to further compress such variable-length results to keep stringently aligned data layouts that facilitate parallel processing and coalesced data accesses. Furthermore, prior designs have not addressed the difficulties of GPU programming and CUDA kernel optimization, which are well known to be tricky and burdensome. They all required manual implementation of new kernels to support new compression algorithms and their combinations. As a result, they only supported a limited set of predefined cascaded compression schemes, lacking **generality and extensibility**.

In this work, we aim to break the tradeoff and improve the Pareto frontier between *compression ratios* and *decompression speed*, and at the same time introduce an *extensible* approach to supporting *general* cascaded compression schemes. We thereby design G-CasDec, an optimized GPU decompression framework that achieves the above goals. G-CasDec implements a single, generic CUDA kernel to support arbitrary cascaded compression, instead of manually writing separate kernels for each scheme. It adopts a general decompression flow to conduct a bottom-up traversal on the tree of cascaded compression, where the output decompressed data from the child nodes become the input to the parent node for further decompression. Each node in the cascade tree uses the parallel-friendly compression algorithms inherited from FastLanes [3, 4] to ensure high efficiency on the GPU hardware.

The key innovation of G-CasDec lies in the efficient interaction between parent and child nodes in the cascade tree. One of these challenges is the variable-length intermediate results between tree nodes. Previous designs either overlook the irregularity in parallel execution and memory accesses induced by such variable data lengths [28], or choose not to compress them further and lose compressibility [3, 4]. G-CasDec proposes an effective *aligned batching* scheme that combines several shorter compressed vectors of variable lengths into an aligned batched vector, with modest padding to match the fixed hardware granularity. This enables aligned memory accesses and regular parallel computations on GPUs for high performance, while also supporting variable-length compression for high compressibility.

Another issue is the space to store the intermediate results between tree nodes, for which we prefer to use fast on-chip storage like GPU shared memory and registers. Reusing this precious buffer space across cascade tree nodes is desirable. But with the aforementioned aligned batching, there could be multiple shorter vectors in a batch that are decompressed together but consumed at different times, complicating buffer management. G-CasDec first decomposes the cascade tree into multiple *chains* at the places of variable-length intermediate data, and proposes *chain-based shared memory reuse* along with a *chain-based depth-first decompression flow* to reduce shared memory

Table 1. **Memory hierarchy on NVIDIA RTX 5090** [16, 20, 26].

Memory	Capacity	Bandwidth	Latency
GMEM	32 GB	1.79 TB/s	~ 800 cycles
L2 Cache	96 MB	8 TB/s	~ 300 cycles
SMEM	100 kB per SM	51 TB/s	~ 30 cycles
REG	256 kB per SM	/	/

consumption. Furthermore, G-CasDec adopts *register-level scheme fusion* within each chain to forward intermediate results at the register level, which is even faster than the shared memory. Such efficient utilization of GPU on-chip buffers is key to fast decompression.

Finally, extending to more compression algorithms and cascaded schemes in G-CasDec requires an easy-to-use programming interface. We define a set of *bytecode instructions* that specify the execution flow for an arbitrary cascade tree, e.g., which compression algorithm is used in each node and how they are cascaded. The bytecode is attached to the compressed data and, at runtime, is interpreted and dispatched by the generic decompression kernel, thus avoiding separate kernel implementation for every new scheme and ensuring extensibility with little programming effort. We further apply a few *composable compiler optimization passes*, such as scheme fusion (mentioned above) and buffer elision, on this bytecode.

We compare G-CasDec with the state-of-the-art GPU decompression frameworks, i.e., Tile-Based [28] and FastLanes [3, 4], as well as several earlier designs. G-CasDec outperforms them in *both* compressibility and performance. For compressibility, G-CasDec achieves 2.8× and 1.3× higher compression ratios than FastLanes and Tile-Based, respectively, and is within 3.2% of the CPU-based design that does not consider massive parallel decompression. For decompression speed, G-CasDec outperforms FastLanes and Tile-Based by 2.0× and 4.3× on average, respectively. When executing end-to-end queries [24], G-CasDec exhibits 1.1× to 1.9× speedups over the best GPU baseline. When additionally accounting for the PCIe transfer time from the host, G-CasDec is still 1.0× to 1.4× faster. We open source G-CasDec at https://github.com/tsinghua-ideal/G-CasDec.

The paper makes the following contributions:

- We identify the critical tradeoffs between compression ratios and decompression speed, and the limitations of generality and extensibility in existing designs for GPU-based cascaded decompression.
- We design G-CasDec, an optimized framework to enable efficient decompression on GPUs for general cascaded compression schemes. G-CasDec outperforms existing designs in terms of both compressibility and performance, and is also extensible with little programming effort.
- We propose several key techniques in G-CasDec, including (1) aligned batching to support efficient decompression of variable-length data, (2) shared memory reuse and register-level scheme fusion in a chain-based decompression flow to improve resource utilization, and (3) a set of bytecode instructions and compiler passes to orchestrate arbitrary cascade tree decompression in a generic kernel.

2 Background & Motivation

We first introduce the GPU basics in Section 2.1, with a specific focus on NVIDIA's architecture and the CUDA programming model. Then we discuss data compression techniques used in databases in Section 2.2, particularly the state-of-the-art (SOTA) cascaded compression schemes. Section 2.3 describes the prior designs for cascaded compression on GPUs and summarizes their limitations.

2.1 GPU Architecture

CUDA programming model. CUDA uses the single-instruction multiple-thread (SIMT) execution model. In this paradigm, developers define scalar GPU *kernels* that execute across thousands of concurrent threads. The architecture organizes these threads into independent, parallel-executing *thread blocks*. At the hardware level, the GPU partitions each thread block into *warps*. Each warp contains 32 threads that are scheduled in lock-step [23].

GPU hardware and occupancy. A GPU comprises independent *Streaming Multiprocessors (SMs)*. Each SM has four warp schedulers, each capable of issuing instructions to a warp per cycle. Since the active threads in a warp execute the same instruction in lock-step, the warp serves as the GPU's finest scheduling granularity.

Although GPUs offer massive thread-level parallelism, individual thread execution is in-order and high-latency. When an instruction encounters a hazard, e.g., a memory load dependency between adjacent instructions, the warp *stalls*. To keep instructions being issued, the scheduler adopts the *latency-hiding* technique by context-switching from a stalled warp to another eligible, ready warp. This is vital for maximizing hardware utilization, including computation and memory bandwidth.

However, warps on an SM share finite hardware resources (e.g., registers and shared memory). The usage of these resources by each warp is determined by the kernel, so the total number of warps on an SM, termed *occupancy*, is limited. Higher occupancy, i.e., more warps, typically offers more opportunities for latency-hiding. Nevertheless, when more resources are given to each warp, techniques such as pipelining and prefetching can also improve performance even when occupancy is low. Thus we should balance SM occupancy and per-warp resource usage.

GPU memory hierarchy. All SMs on a GPU share the *global memory* (GMEM), enhanced with a unified L2 cache (a.k.a., last-level cache). Locally, each SM contains fast on-chip SRAM (static random-access memory) that is configurably partitioned between the L1 cache and the *shared memory* (SMEM). Although both offer similar latencies, SMEM is a software-managed, programmable data buffer. Additionally, each thread has a fixed set of *registers* (REGs). Table 1 summarizes the capacity, bandwidth, and latency across the memory hierarchy on a typical GPU.

Because GMEM has the highest latency, optimizing accesses to it with strict *data alignment* is critical. Upon issuing a GMEM load or store, the hardware coalesces all the addresses into discrete naturally-aligned 32-, 64-, or 128-byte *memory transactions*. For example, a warp accessing 32 contiguous 4-byte words triggers a single 128-byte transaction. In contrast, random access patterns can reduce throughput by up to 32× due to more transactions. Consequently, achieving coalesced memory accesses is a primary objective in GPU kernel optimization.

SMEM (typically ~100 kB) is essential for high-performance kernels. It enables both custom data-caching strategies and low-latency scratch buffers allowing data exchange among threads within the same thread block. However, excessive SMEM usage hurts occupancy. Similarly, while REGs provide the fastest access, their scarcity requires precise allocation to maintain high occupancy.

2.2 Lightweight Compression

Compression schemes. Lightweight compression schemes (i.e., LWCs or encodings) are widely used in modern columnar data formats and databases [1, 19, 29, 37, 39]. They achieve fast decompression speed while maintaining high compression ratios. *Bit-Packing* (BP) [2] represents integers using only the minimum required bitwidth. *Run-Length Encoding* (RLE) [2] compresses contiguous sequences of identical values into pairs consisting of the *running value* and the *running length* (i.e., repetition count). *Constant* (Const) is a special case of RLE where every value in a sequence is the same. *Dictionary Encoding* (Dict) [2, 40] replaces frequent values with designated compact integer

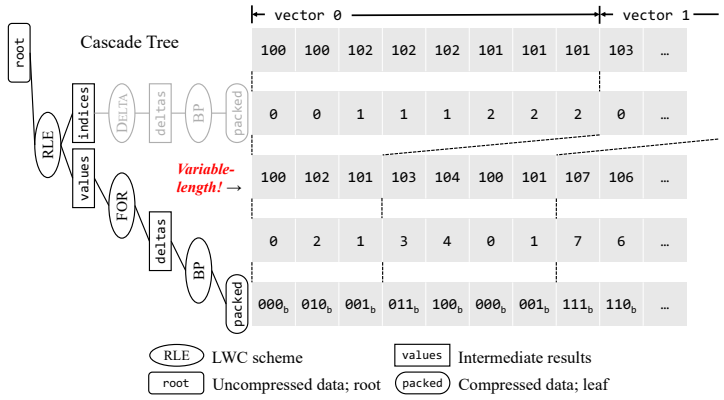


Fig. 1. **A cascade tree with level-1 (FastLanes-)RLE, level-2 FOR and DELTA (omitted), and level-3 BP.** – Each value sequence on the right corresponds to the original/intermediate data along the cascade tree on the left; e.g., the 1st sequence is the uncompressed data at the root, the 2nd and 3rd are the indices and running values of RLE for the root's children, etc. Some sequences, e.g., running values, are variable-length.

codes. *Delta Encoding* (DELTA) [18] records only the differences between adjacent values, achieving compression when combined with Bit-Packing. *Frame-Of-Reference* (FOR) [40] encodes values as small deltas relative to the frame/block's base value (typically the minimum). *Patched Encoding* (PATCH) [40] separately stores the outlier values from the majority of the data so the majority data can be better compressed with other encodings. *Frequency Encoding* (FREQUENCY) [17, 25] isolates the most frequent element and stores the remaining values serially, which is highly effective for skewed distributions.

Cascaded compression. Modern systems often apply the above schemes in a cascaded fashion, i.e., recursively apply an additional LWC layer to the output of a previous scheme. For example, Apache Parquet [7] first applies DICT to the values and then compresses the resulting codes using a hybrid of RLE and BP. Similarly, NVIDIA nvCOMP [22] supports cascading RLE and DELTA. However, these systems typically rely on pre-defined LWC combinations and lack the flexibility to adapt to diverse data patterns.

BtrBlocks [17] introduces a framework for arbitrary multi-level cascading. It recursively selects the optimal scheme via a sampling-based approach. For a given sequence, BtrBlocks evaluates the candidate LWC schemes on a sampled dataset to identify the one that yields the highest compression ratio. This process repeats on the outputs of the last scheme until the cascading level reaches a predefined threshold. Figure 1 shows an example *cascade tree*.

Decompression involves a bottom-up traversal of the cascade tree. Each node ingests the output from its children (or the input compressed data) and decompresses it according to its scheme. To improve performance, a scheme fusion may execute multiple nodes in a single pass. The process of cascaded decompression requires *scratch buffers* to store intermediate results, such as the decoded FOR values required as the running values of the parent RLE node in Figure 1. Writing intermediate results to a scratch buffer is called *materialization*.

2.3 Prior Cascaded Decompression on GPUs

Crystal [27] and Crystal-opt [11] establish the foundation for GPU-based query execution. By making each thread block process a vector of tuples across selected columns, they achieve SOTA

performance on uncompressed columnar data [24]. Nonetheless, many queries remain memory-bound [36], making columnar data compression essential. NVIDIA nvCOMP [22] is a general-purpose GPU compression library, but its cascaded mode supports only two LWC schemes, RLE and Delta, with limited compression ratios.

Cascaded decompression relies heavily on scratch buffers to materialize intermediate results. Unlike CPU-based frameworks such as BtrBlocks, GPU implementations must explicitly manage buffer placement to optimize SMEM and register utilization. Shanbhag et al. [28] introduced the first tile-based decompression model on GPUs, where threads in a thread block collectively decompress a *vector* of tuples at a time. We name this work Tile-Based in this paper. By inlining decompression into query execution, it bypasses the off-chip memory bandwidth bottleneck by avoiding GMEM materialization. However, while tile-based, the programming model, is effective, Tile-Based, the prior work, lacks extensibility because it requires manual implementation for a restricted set of fused schemes. Moreover, Tile-Based adopts CPU-centric data layouts and algorithms, without regard for GPU-critical factors such as GMEM alignment. This leads to suboptimal GPU kernels that fail to saturate the GPU memory bandwidth and are thus slower than the uncompressed Crystal-opt baseline [5, 28].

FastLanes focuses on parallelizing LWC decompression for modern hardware. Its CPU implementation [3, 4] employs SIMD-friendly optimizations to enable independent "lane" processing on the elements of a *vector*. For example, FastLanes-RLE replaces the running lengths with the *indices* into the running values; e.g., on the vector 100, 100, 102, 102, 102, 101, 101, 101, it uses the indices 0, 0, 1, 1, 1, 2, 2, 2 to point to the running values 100, 102, 101, rather than the standard running lengths 2, 3, 3. These indices can be readily compressed by Delta in a cascaded way. They also propose FastLanes-Delta that pre-computes the prefix sums called *base* at each lane to better parallelize decompression. RLE is thus made fully parallel-friendly. While these designs are portable to GPUs by treating each SIMD lane as a GPU thread, the current FastLanes-GPU [5] implementation remains a preliminary effort, supporting only BP and Delta and lacking cascaded decompression support.

A significant drawback of FastLanes is its rigid 1024-value vector granularity, in which all data are decompressed. While such a decision prioritizes decompression speed by avoiding hardware-unfriendly alignment and dynamic loops, it sacrifices compression ratios due to its inability to further compress variable-length intermediate results produced by LWC schemes, such as the running values of RLE, values of Dict, and exception values of Frequency and Patch. Take Figure 1 as an example again. After the first level of RLE, the running values are variable-length, which FastLanes *cannot* further compress with FOR and BP as shown in the figure. Prior work, including BtrBlocks, Tile-Based, and nvCOMP, demonstrates that cascaded compression of those variable-length intermediate values improves the compression ratio significantly.

Summary of limitations. Existing GPU cascaded decompression designs are preliminary and face critical limitations on the **generality** (i.e., only support a few hard-coded cascades), **compressibility–performance tradeoff** (i.e., either provide poor compression ratios or slow decompression), and/or **extensibility** (i.e., require manual programming effort to support new cascades). Table 2 compares Tile-Based and FastLanes against our approach. Both existing frameworks support only limited, pre-defined cascades that are tailored to specific datasets. This lack of generality degrades performance on diverse workloads. For instance, several columns in the Public BI dataset [31] can achieve up to a 2.13× higher compression ratio with more flexible LWC combinations than those supported by FastLanes, as in Figure 2. Furthermore, extending these frameworks is difficult because adding a new cascade requires manually implementing and optimizing complex CUDA kernels from scratch.

Table 2. **Comparison between prior work and G-CasDec, and how G-CasDec overcomes prior limitations.**

	Tile-Based	FastLanes	G-CasDec (Ours)
Generality	✗ Only three cascades supported.	✗ CPU version only supports cherry-picked cascades; GPU version only supports BP.	✓ Automatic support for arbitrary cascades with efficient workflow of *chain-based depth-first decompression.*
Compression Ratio	✓ High, but manually tuned on specific datasets.	✗ Low. Variable-length intermediate data are not compressed. GPU version only supports BP.	✓ High. Can compress variable-length intermediate data with *aligned batching.*
Decompr. Speed	✗ Low, due to unaligned loads and dynamic loops.	✓ High, with parallel-friendly optimizations.	✓ High, with parallel-friendly optimizations, *chain-based shared memory reuse*, and *register-level scheme fusion.*
Extensibility	✗ Kernels for different cascades are manually implemented.	✗ CPU version separately implements each cascade; GPU version only implements BP.	✓ Low programming labor, with a generic kernel controlled by *bytecode instructions*, and composable *compiler optimization passes.*

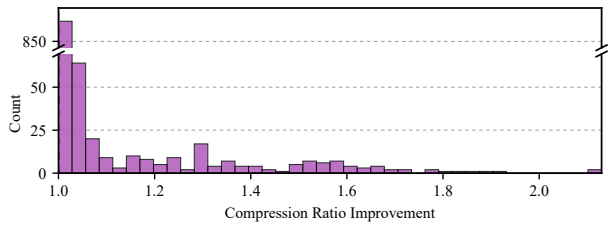


Fig. 2. **Compression ratio improvement from flexible cascades.** – Histogram over all 1072 numeric columns in Public BI, comparing G-CasDec against the same implementation restricted to FastLanes-CPU's set of cascade trees.

Figure 3 quantifies the performance-compressibility tradeoff using the `lo_orderdate` and `lo_custkey` columns from the Star Schema Benchmark (SSB) [24]. Although Tile-Based achieves high compression ratios, its decompression throughput is low. FastLanes, in contrast, offers higher throughput via specific parallel-friendly optimizations but suffers from low compression ratios due to its lack of cascaded compression for variable-length intermediate values. FastLanes-GPU exhibits poor compression ratios because it only supports BP and Delta. Even an enhanced baseline, GPU-NoVarLen, which supports a wide LWC scheme pool but excludes variable-length compression similar to FastLanes-CPU, fails to match the compression ratios achieved by our solution.

3 G-CasDec Design Overview

In this work, we aim to enable efficient decompression on GPUs for columnar data under general cascaded compression schemes. We have the following design goals:

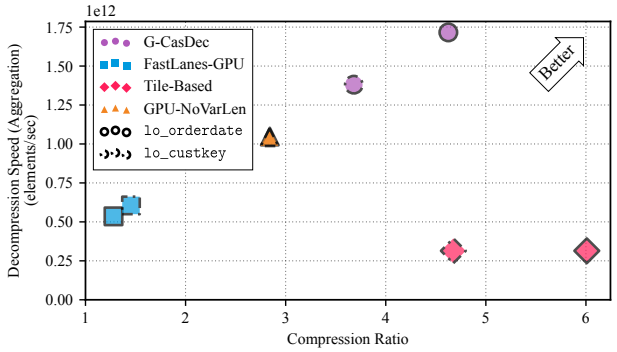


Fig. 3. **Compression ratio and decompression speed tradeoff.** – Tested on two columns in the `lineorder` table from SSB between Tile-Based, FastLanes-GPU, a basic GPU design without variable-length compression (GPU-NoVarLen), and our design G-CasDec. FastLanes-CPU would have similar compression ratios as GPU-NoVarLen but is much slower.

Generality The design should present a general yet efficient framework for decompression of arbitrary cascade trees.

Compressibility The design should achieve SOTA compression ratios without trading off performance. Particularly, it needs to support *compression of variable-length intermediate results* in the cascade tree with fast decompression.

Performance The design should achieve SOTA decompression speed. Particularly, GPU resources like *shared memory and registers must be automatically and systematically managed*, maximizing data reuse while balancing resource usage against SM occupancy.

Extensibility The design should be extensible — adding a new LWC and/or a new cascaded scheme should be simple, *without excessive manual programming effort*. Moreover, such extensibility cannot hurt performance, i.e., the design should be able to *automatically generate optimized GPU kernels* for arbitrary cascaded decompression.

To meet these goals, we design G-CasDec, an optimized framework that realizes general and efficient cascaded decompression on GPUs. G-CasDec builds upon the SOTA FastLanes algorithm [3–5], and presents a general framework that allows for automatic extension and easy programming to support arbitrary cascaded schemes. It also significantly improves the Pareto frontier between compression ratios and decompression speed, by enabling compression of variable-length intermediate results while efficiently utilizing available GPU on-chip resources.

The core workflow of G-CasDec that supports arbitrary cascade trees, which we call *chain-based depth-first decompression*, is presented in Algorithm 1. As in all other cascaded decompression algorithms, G-CasDec decompression performs a bottom-up traversal of the cascade tree to recursively decompress the intermediate results and feed them to upper levels in the cascade tree. Specifically, G-CasDec adopts post-order depth-first search (DFS), decompressing the children of a node first in a depth-first manner (the Decode function in Algorithm 1). For example, following the cascade tree in Figure 1, the bottom-row sequence is decompressed by BP, whose output is next decompressed by FOR. The generated running values are then combined with the indices decompressed from the sibling path, and fed to RLE decompression to restore the original data sequence in the top row.

Similar to FastLanes [3, 4], in G-CasDec we group 1024 values into a *vector*. Only one vector of data at a time is decompressed by one warp (32 threads) in a thread block. Following BtrBlocks [17] and FastLanes-CPU [4], we group 64 vectors into a *chunk* (65536 values), sharing the same cascade

tree. The vectors in a chunk are iteratively decompressed using the main loop in Lines 16 to 18. Different chunks may exhibit different value patterns and can choose different cascaded schemes.

A key feature of G-CasDec is the support for variable-length intermediate data compression to improve **compressibility**, with efficient decompression enabled by the design of *aligned batching* (Lines 4 to 10 in Algorithm 1). Specifically, we batch several compressed vectors of variable lengths into an aligned and padded batch of a fixed length that matches the hardware granularity. This leads to aligned memory accesses to maximize GMEM bandwidth utilization, and static-extent loops that can be unrolled by the compiler to reduce control flow overheads (Section 4).

Aligned batching causes multiple vectors in the batch to be materialized altogether in the scratch buffers, but consumed at different times, complicating the management of SMEM and registers. To better manage on-chip resources for **performance**, G-CasDec decomposes the cascade tree into multiple *chains* breaking at nodes producing variable-length data. Then we *reuse shared memory space* in each chain to improve SM occupancy. We further propose *scheme fusion* in chains to leverage the fastest registers (Lines 10 and 13 in Algorithm 1). The chain decomposition and the fusion together determine our overall *chain-based depth-first* flow (Section 5).

Finally, we propose a set of *bytecode instructions* dispatched by a generic kernel, to efficiently realize the complicated control flow in Algorithm 1 for arbitrary cascaded schemes. The bytecode-based design allows for high **extensibility** in G-CasDec without the need to write specialized kernels. For example, we can use *composable atoms* to fuse multiple LWC schemes with minor programming effort (Section 6).

Algorithm 1 Chain-based depth-first decompression

```
1:  function DECODE(n, i)
2:      ▷ n is the current cascade tree node                                   ◁
3:      ▷ i is the index of the current vector to be decompressed              ◁
4:      if c ← var-len-child(n) exists then                          ▷ Section 4
5:          j ← ⌊ n.breaks[i] / vec_size ⌋                          ▷ Vector index for c
6:          j' ← if i > 0 then ⌊ n.breaks[i−1] / vec_size ⌋ else -1  ▷ Previous vector index for c
7:          if j ≠ j' then                                          ▷ Need immediate decompression
8:              DECODE(c, j)                                        ▷ Otherwise data are ready
9:          offset ← n.breaks[i] mod vec_size
10:         var_in ← scratch[chain_id(c)] + offset                  ▷ Section 5.2
11:     if c ← same-len-child(n) exists then
12:         DECODE(c, i)
13:         in ← scratch[chain_id(n)]                               ▷ Section 5.3
14:     out ← decompress n, supplying var_in and in if any
15:     scratch[chain_id(n)] ← out

16: for each i from 0 to 63 do                                      ▷ Main loop for a chunk
17:     DECODE(root, i)                                             ▷ Decode one vector at a time
18:     Do computation on vector i in scratch[chain_id(root)]
```

3.1 Basic Settings

In alignment with common prior practices, we briefly list some basic settings for G-CasDec here.

Compressor. In line with common practice in this field [5, 28], our study focuses on optimizing the decompressor executed on the GPU, and leaves the compression to a basic CPU implementation that organizes input data offline. The compressor recursively enumerates all the cascaded options containing up to 4 schemes, and selects the one with the highest compression ratio (though for DICT

with more than 128 values, we apply a size penalty of 10% to trade off for speed). Some basic speed optimizations are applied: we enforce a canonical order between two LWC schemes if they commute without compression ratio difference, and we maintain a running best size for backtracking. We briefly report the compression time in Section 8.2.1. More advanced techniques such as compression by sampling in BtrBlocks [17] and FastLanes-CPU [3, 4] can be readily implemented in the future.

Data types. Regarding supported data types, we mainly deal with numeric columns of integer types, which is a common setting aligned with previous designs [5, 27, 28]. We discuss how to support other data types in Section 7 and showcase the extension to floating-point numbers in Section 8.4.

LWC schemes. The current implementation of G-CasDec supports all the LWC schemes introduced in Section 2.2: BP, Const, FOR, RLE (the FastLanes version that uses indices into running values), Dict, Delta, Patch, and Frequency. New schemes can be easily added with a registration interface in G-CasDec, since there is no mutual dependency among schemes (Section 7).

4 Aligned Batching for Variable-Length Intermediate Data

As mentioned in Section 2.3, variable-length intermediate results compressed from some LWC schemes like RLE introduce a dilemma: not compressing them leads to poor compression ratios in FastLanes, while compressing them results in low performance in Tile-Based. In this section, we first dive into the problems with these prior designs in more detail (Section 4.1), and then discuss how we are able to push the Pareto frontier between compression ratio and decompression performance to the next level via our novel aligned batching technique (Section 4.2).

4.1 Problems with Prior Designs

The problem with Tile-Based [28] originates from the fact that some LWC schemes, such as RLE, produce *shorter and variable-length* compressed output [12]. For example, in Figure 1, the running values 100, 102, 101 are significantly shorter than the original vector of length 8. As we compress further down the cascade tree, the data length becomes increasingly shorter. Moreover, the length is determined by the data value pattern rather than statically known, and may therefore also vary significantly across different vectors. During decompression, those short and variable-length data would result in short and dynamic loops, which hurt performance in two aspects. First, short and dynamic loops on GPUs can introduce large control flow overheads and decrease instruction-level parallelism (e.g., data loads cannot be hoisted by the compiler), reducing computing unit utilization and leading to worse performance. Second, shorter and variable-length data lead to unaligned memory accesses, making parts of the data blocks fetched in a memory access transaction (e.g., 128 B) unused and thus wasting precious GMEM bandwidth. These two effects combined hinder the performance of Tile-Based.

In contrast, FastLanes-GPU [5] restricts each loop to the stringent size of a vector (i.e., 1024), in the hope of eliminating the short and dynamic loops and the induced performance issues. Specifically, a vector of 1024 elements is the finest granularity in decompression, which meets the 128 B GMEM alignment requirement. Such a vector length is also sufficient to well amortize the loop control flow cost. However, because of this stringent vector size requirement, FastLanes cannot compress the inherently dynamic and variable-length intermediate results produced by certain LWC schemes like RLE, effectively trading compression ratios for decompression speed. Such lack of support for variable-length-output LWC schemes also limits the generality of FastLanes.

In summary, the above problems in the two prior designs motivate us to find a solution that can compress variable-length data while at the same time retaining data alignment and sufficiently long vector lengths.

4.2 Our Solution: Aligned Batching

The key insight of our solution, *aligned batching*, is that since there are multiple (e.g., 64) vectors in a chunk, multiple short variable-length data from these vectors can actually be *batched* to form a longer sequence whose length is close to the hardware-friendly granularity. It is then further padded to an exactly *aligned* vector to ease memory accesses, at the expense of an irregular traversal order of the cascade tree.

For instance, in Figure 4 which shows the same cascade tree as in Figure 1, the vector length is 8. After RLE compression, the first vector's running values are 100, 102, 101, and the second vector has 103, 104, 100, 101. They are naturally generated and stored contiguously. By combining the two and padding[1] with the last value 101 (grayed out, ignored in decompression), we form a new vector with the same length and alignment as the original data. This batched vector is then further compressed with FOR and BP. Note that with our configuration of 64 vectors per chunk, even with deeply nested cascaded compression, the bottom levels still have enough vectors to batch, as 64 is much larger than usual compression ratios around 5×. The size overhead of padding is minimal as we will show in Section 8.5. Even if the padding overhead can be large occasionally, since the compressor optimizes for size, it will select other simpler cascades as fallback, rather than leading to < 1 compression ratios.

During decompression which happens bottom-up, we take each batched vector at the leaf level and decompress. When we reach the level at which the data length extends (e.g., from the 3rd row to the 2nd row in Figure 4), the single batched vector would be decompressed into multiple vectors. In such tree nodes, we store extra metadata named *breaks*, which are pointers to the beginning offsets of individual vectors in the batch. For example, the first vector for RLE in Figure 4 has break 0 and sources the first 3 elements for decompression, and the second vector has break 3 and sources the next 4 elements still in the same batch. The third vector will find its break to be 8 and require decompressing another batched vector from the lower levels of BP and FOR.

We now describe aligned batching formally. During compression, the variable-length intermediate data are batched until adding another sequence will exceed the vector size. Then the last element in the current sequence is repeated as the padding. This technique works when the variable-length sequence is shorter than the original vector, which holds for many LWC schemes such as RLE and Frequency. The breaks are collected during this batching process and stored as extra metadata to the current node. During decompression, with the breaks, G-CasDec can easily compute which vector it needs (`break / vec_size`) and the offset within the vector (`break % vec_size`), as in Lines 5 and 9 in Algorithm 1. Padding at the unused tail of the aligned vector is ignored and later overwritten. Each vector in the batch is decompressed and materialized in a scratch buffer only once. Later decompression from these vectors can directly source the data from this scratch buffer multiple times, without redundant decompression.

Aligned batching brings the following benefits. First, compression ratios are improved because we now can do cascaded compression on the batched variable-length intermediate data. Second, GMEM accesses are always aligned and coalesced in the fixed vector size, maximally utilizing access bandwidth. Third, fixed-size vectors lead to static-extent loops during decompression, which can be easily unrolled by compilers, eliminating control flows and increasing instruction-level parallelism. We will show these effects with profiling results in Section 8.5. In the current implementation of G-CasDec, we implement aligned batching for running values of RLE and exception values of Frequency. We plan to add support for other schemes such as Patch and more emerging schemes.

[1]According to the LastNonNull strategy to maximize compression ratio [38].

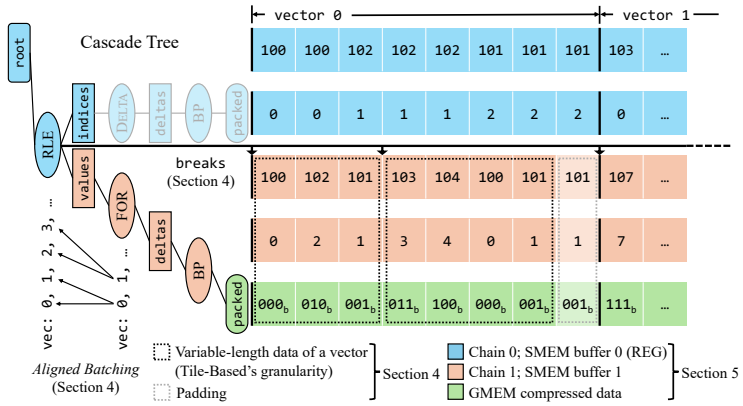


Fig. 4. **Aligned batching (Section 4) and chain-based depth-first decompression (Section 5) in G-CasDec, applied to the same cascade tree in Figure 1.** – In the 3rd sequence, the RLE running values are batched across vectors and padded to the vector alignment. The cascade tree is decomposed into several *chains* to facilitate scratch buffer allocation.

Challenges. Aligned batching introduces its own challenges. The execution flow on the cascade tree during decompression becomes irregular. One vector may produce multiple vectors, each of which needs to continue its own decompression, as described before. For example, the traversal order between RLE and FOR in Figure 4 now looks like FOR.`vecs[0]`, RLE.`vecs[0]`, RLE.`vecs[1]`, FOR.`vecs[1]`, RLE.`vecs[2]`, RLE.`vecs[3]`, ..., which is a data-dependent irregular flow. In addition, we also need to carefully manage the scratch buffers in which the decompressed vectors in the batch are materialized. This is critical as the decompressed vectors need to stay in the buffer and wait to be further decompressed one after another following the above flow. The next two sections will present how to alleviate the performance issues and enable flexible execution flow without programming burdens.

5 Chain Decomposition for On-Chip Resource Management

In cascaded decompression, one of the most performance-critical tasks is managing the scratch buffers used to materialize the intermediate results in cascade trees. Tile-Based [28] and Fast-Lanes [5] each supports only a limited set of cascades, so they can manually optimize the use of GPU shared memory and registers to realize scratch buffers. In contrast, G-CasDec aims to decompress arbitrary cascade trees, and thus it becomes non-trivial to manage these critical on-chip resources (Section 5.1). We propose a systematic approach, which first decomposes the tree into multiple *chains*, and then applies *chain-based shared memory reuse* (Section 5.2) and *register-level scheme fusion* (Section 5.3) to manage the shared memory and register resources for intermediate result materialization.

5.1 Difficulty of On-Chip Resource Management

The intermediate decompressed data between nodes in a cascade tree require materialization in some scratch buffers. On CPUs, these scratch buffers are in DRAM and can be transparently cached on-chip by hardware, as is done in BtrBlocks [17]. But on GPUs, the on-chip SRAM resources need more careful treatment. The SMEM is explicitly managed by software, and there are many more registers than on CPUs. Hence their management becomes a critical performance consideration.

The SMEM is versatile and byte-addressable, so it supports flexible and dynamic accesses, in particular to the variable-length intermediate data. For example in Figure 4, when performing RLE decompression, the running values under aligned batching and their decompressed output vectors can be kept in the SMEM. In contrast, registers, while being faster, can serve as scratch buffers only if all the accesses are static and regular. For example, the LWC scheme FOR adds a reference to every element in the vector, so the input and output have the same length and one-to-one correspondence, making it feasible to allocate registers for each.

Unfortunately, both SMEM and registers are limited, and excessive use can harm SM occupancy and degrade performance (Section 2.1). Even the most advanced server-grade GPU (e.g., NVIDIA H200 or B200) can only have 228 kB of SMEM and 256 kB of registers per SM. To reach the maximum occupancy (32 thread blocks per SM), a thread block can use at most 7296 B SMEM and 8192 B registers, which translate to only about 2 vectors of 32-bit integers.

Prior systems including Tile-Based [28] and FastLanes [5] manually control the materialization of data to SMEM and registers. Hence each cascade requires a different optimized strategy that hardens the corresponding resource allocation, lacking a general approach for arbitrary cascade trees and limiting them to only a handful of cascaded schemes. For example, `GPU-RFOR` in Tile-Based uses RLE followed by FOR and BP of the running lengths and running values. During decompression, the unpacked data from BP, after the frame of reference is added in FOR, are written to hard-coded locations in the SMEM scratch buffer. Therefore, a systematic buffer allocation approach is needed to support arbitrary cascade trees while minimizing resource usage of SMEM and registers.

5.2 Chain-Based Shared Memory Reuse

We aim to minimize the SMEM usage during decompression, by sharing the same SMEM space between multiple scratch buffers for different tree nodes' intermediate data. We notice that, when the LWC scheme of a node ensures the same length for the input and output data sequences, we can conduct *in-place operations* in the same SMEM space[2]. On the other hand, it is difficult to reuse the SMEM buffer that stores the variable-length data, such as the RLE running values in Figure 4. Because the variable-length sequences in the batch are fully dynamic, we cannot recycle the space of one of them even if it has been consumed, as the space may not be enough for another vector. For example, after we consume the running values of 100, 102, 101 to decompress into `vector 0` in the top sequence in Figure 4, we may free up the 3 slots, but it is not guaranteed that the first variable-length sequence from the next batched vector can fit in these free slots. So, we reclaim the space of the entire batched vector only after it has been fully consumed.

Following this observation, we decompose the cascade tree into multiple *chains* at the places where variable-length intermediate data are produced. Thus within each chain, the intermediate vectors have the exact same length, which enables *reusing* the same SMEM scratch buffer when decompressing the LWC schemes within each chain, while different chains are allocated with different SMEM buffers. Therefore, if the cascade tree can be decomposed into n chains, we need n scratch buffers. The buffer of the i-th chain C_i is called buffer B_i. As each iteration decompresses one vector, the buffer is of the same size as a vector, i.e., 1024 elements. In real-world datasets like Public BI [31], 14% of the cascade trees have more than 1 chain, and the decomposition can be more important when more schemes have variable-length data.

Figure 4 illustrates chain decomposition in colors. In this example there are 2 chains. C_0 colored in blue contains the nodes BP, Delta, and RLE; C_1 colored in orange contains the nodes BP and FOR. The green nodes refer to the input compressed data from GMEM.

[2]We currently assume all intermediate data are of the same type as the raw data, e.g., `int32`. Alternatively, we can allocate buffers using the wider type between input and output.

Within each chain, intermediate results of the same length use in-place operations. Consider the FOR node in Figure 4 as an example. Assume its child BP has written a vector to B_1, the SMEM scratch buffer of this chain C_1. Each value (FOR.`deltas`) only needs a reference value to be added, so this can be done in-place in B_1.

The intersection of chains is a scheme that works on variable-length data; with aligned batching (Section 4), one vector produced by the child chain can be used by the parent chain multiple times to decompress into multiple vectors. Take the RLE node in Figure 4 as an example. Two sequences of running values (100, 102, 101, and 103, 104, 100, 101) are materialized as a batched vector into B_1. After serving the first 3 running values to decompress `vector 0` with another buffer B_0, the beginning pointer of B_1 is bumped by the offset, i.e., 3, computed with the break metadata, to point to the next 4 running values, as in Line 10 in Algorithm 1.

5.3 Register-Level Scheme Fusion

Although realizing scratch buffers in SMEM is versatile and faster than GMEM, it can still become the bottleneck when compression ratios are high and intermediate results are written and read many times. In fact, on modern GPUs, the SMEM bandwidth is only 10× to 20× higher than that of GMEM [16]. With a compression ratio around 5×, and the number of nodes in a chain being 2 to 3 where each of them has intermediate data both read from and written to SMEM, the SMEM accesses could be > 20× more than the GMEM accesses, rendering decompression SMEM-bound. Therefore, we need to further consider using registers as scratch buffers whenever possible.

We notice that in most cases, we can fuse the schemes within a chain, and pass their intermediate results via registers. We call this technique *register-level scheme fusion*. For example, in Figure 4, the BP and FOR nodes in the orange chain C_1 can be fused, writing only the final 1024 RLE.`values` to SMEM buffer B_1 per vector. Without fusion, the intermediate FOR.`deltas` have to go through a round-trip between the registers and SMEM, inducing extra SMEM traffic of 2048 elements. Scheme fusion is enabled by two conditions. First, the schemes within a chain have the same input and output sequence length, decided by the chain decomposition rule. Therefore the data accesses are regular. Second, in many cases the elements are processed independently by the LWC schemes within a chain, so they do not need inter-thread communication via SMEM. For example, FOR adds a reference to every value in parallel; the Delta and RLE variants in FastLanes [3] require no inter-thread communication, either. An exception is Patch, where the warp collectively patches the vector via inter-thread communication that relies on SMEM. When such schemes are present in a chain, the chain is split into several parts that are separately fused.

Back to the overall flow. Combining the buffer allocation strategies in Sections 5.2 and 5.3, we hereby revisit the chain-based depth-first decompression in Algorithm 1. The scratch buffers are assigned using `chain_ids`. While the cascade tree can be decompressed in an arbitrary bottom-up topological order, scheme fusion requires the schemes within a chain to be executed back-to-back. Therefore, we perform a post-order traversal of the tree visiting the variable-length child first, and in the resulting depth-first order fusible schemes are executed back-to-back. To support aligned batching, we only enter the variable-length child if that child buffer has been completely consumed.

6 Bytecode Instructions for Extensibility

To retain extensibility while avoiding the need to manually implement and optimize a CUDA kernel for every new scheme and the combinations, G-CasDec defines a set of *bytecode instructions* that specify the execution flow to decompress a cascade tree. This allows us to extensively optimize a single generic kernel, which then interprets the instructions to perform decompression on arbitrary cascades. Various cascades of LWC schemes correspond to different bytecode lists, making our

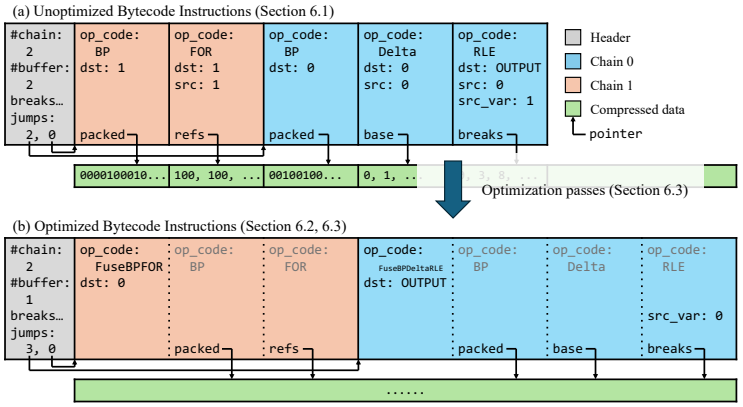


Fig. 5. **Bytecode instructions in G-CasDec to specify the decompression flow of a cascade tree.** – This list corresponds to the tree in Figure 4, and the instruction segments are colored the same as their corresponding chains. Optimization passes like scheme fusion and buffer elision can be performed on the bytecode, transforming (a) to (b).

approach highly *composable*. We describe the bytecode specification in Section 6.1 and how to use it to realize the aforementioned chain-based depth-first decompression flow. Then we implement register-level scheme fusion and buffer elision in Sections 6.2 and 6.3, which are two representatives of the general optimization passes on our bytecode.

6.1 Implementing Chain-Based Decompression

Bytecode is a common approach to program optimization. By feeding different bytecode lists to a common generic kernel, we can flexibly support diverse cascaded schemes without implementing specialized kernels, avoiding excessive programming burdens. Furthermore, compared to the typical approach of recursively traversing LWC decompression along the cascade tree (as in Algorithm 1), using bytecode lists can simplify the execution into a loop around a big `switch` statement, greatly reducing control flow overheads associated with recursion. Finally, G-CasDec, like a compiler, can also use the bytecode as an intermediate representation (IR) to perform various optimization passes [32], which are implemented as composable transformations on the bytecode (Section 6.3).

Figure 5 shows the G-CasDec bytecode for the cascade tree in Figure 4. Each LWC scheme corresponds to one instruction, encoded in a variable-length format of 8 or 16 bytes. Each instruction contains several fields including `op_code` for the LWC type, `src` and `dst` for the source and destination buffer IDs (if any, otherwise to/from GMEM), pointers to embedded metadata in the compressed data (e.g., *breaks* for RLE, *refs* for FOR), and other scheme-specific metadata. At runtime, the `op_code` is parsed and dispatched by a `switch` statement to a corresponding CUDA device function, which is automatically generated from an atom (Section 6.2) or a fusion of atoms at compile time. The device function then parses the remaining metadata and executes the decompression logic. The instructions of the LWC tree nodes are in the depth-first order in Algorithm 1. Following the chain decomposition in Section 5, the list of instructions is divided into *segments*, where one segment corresponds to one chain (with the same coloring between Figures 4 and 5). The bytecode instructions are *lowered* from the cascade tree in the epilogue phase of compression, and multiple optimization passes (see later) can run on them. The final bytecode is attached to the output compressed data chunk, and its space overhead is amortized over the large chunk of many vectors.

```
template <int bw>
void BP::dec(u32 *in, u32 *out)
{
  u32 some-states;
  for (int i = 0; i < 32; i++) {
    u32 result = compute-i-th;
    out[tid+i*32] = result;
  }
}
// Simple composition is not fused
void unfused(u32 *in, u32 *out) {
  BP::dec<bw>(in, out);
  // Need to re-read whole vector
  FOR::dec(out, ref, out);
}
// Fusion requires dedicated code
void fused(u32 *in, u32 *out) {
  // Implemented elsewhere
  BP_FOR::dec<bw>(in, ref, out);
}
```

(a) Traditional LWC schemes implementations require manual handling in fusion.

```
template <int bw>
struct BP::Atom {
  u32 some-states;
  // Atoms are scalar operations
  u32 operator()(int i) {
    u32 result = compute-i-th;
    return result;
  }
};
// One line of macro per fusion
#define SCHEME_FUSION_LIST \
    SCHEME_FUSION_ITEM(BP, FOR) ...
// Expands to...
void fused(u32 *in, u32 *out) {
  auto atom0 = BP::Atom<bw>(in);
  auto atom1 = FOR::Atom(ref);
  for (int i = 0; i < 32; i++)
    // Fusion is atom composition
    out[tid+i*32] = atom1(atom0(i));
}
```

(b) Atoms enable fusion via simple composition, which can be automated with macros.

Fig. 6. **Comparison of traditional scheme fusion implementation vs. G-CasDec.** – Each LWC scheme is abstracted into an *atom* in G-CasDec, allowing compositions for scheme fusion.

During decompression, the bytecode instructions are pre-loaded to the SMEM for fast access. Then for each vector, the instructions are sequentially traversed, interpreted and dispatched as stated above. Under our aligned batching, non-top chains (other than C_0) may produce intermediate results in their scratch buffers that are sourced by their parent chains multiple times. This means that, after the child chain's instructions run once, the parent chain's instructions need to execute multiple times to consume the intermediate data. Therefore, in each iteration, the runtime computes based on the *breaks* whether the intermediate results need to be updated, and skips a chain if not. The *jumps* pointers encoded in the header tell us the beginning of each chain's segment. For example, following Figure 4, the first iteration executes all the instructions in Figure 5 to generate `vector 0`; while in the second iteration, we skip the C_1's instructions (in orange), and only execute C_0 on the running values of 103, 104, 100, 101 in the batch.

Our bytecode-based design essentially decouples the high-level cascaded scheme and tree traversal from the low-level individual LWC nodes. The former is implemented as a generic CUDA kernel and extensively optimized to achieve high performance. This ensures high extensibility of G-CasDec, reducing the efforts needed to support new LWC schemes.

6.2 Implementing Scheme Fusion

We implement the register-level scheme fusion in Section 5.3 using a compiler optimization pass on our bytecode, which replaces multiple instructions with a fused instruction. The corresponding decompression function for the fused instruction is implemented using a novel abstraction of composable *atoms* that allow arbitrary composition with little programmer intervention.

Figure 6 (a) first shows the standard approach to implementing each LWC as a GPU device function (e.g., `BP::dec`) which reads from and writes to scratch buffers. Directly putting two such functions back-to-back (as in `unfused()`) does *not* actually fuse them since it writes the entire intermediate data and then reads them in. So programmers have to write dedicated fused functions (e.g., `BP_FOR::dec`). There can be a combinatorially large number of different LWC fusions, let alone the potential need to add new schemes in the future.

G-CasDec instead abstracts each LWC's decompression as a scalar operation called *atom*, which enables easy *composition* of schemes into fusions, as shown in Figure 6 (b). First, since each atom acts on an element instead of the whole vector, each element is decompressed with a chained invocation

Table 3. **Lines of code (LoC) per scheme implementation.**

Scheme	BP	Const	FOR	RLE	Dict	Delta	Patch	Frequency
LoC	669	164	169	240	204	178	377	275

of atoms (`atom1(atom0(i))`), allowing the CUDA compiler to use registers for intermediate results. This also allows C++ templates to compose these atoms into composed functions in template instantiations, e.g., `atom1` ∘ `atom0`. Second, with the use of X macros [33] in C++, each fusion can be defined in a single line of macro, as simple as `SCHEME_FUSION_ITEM(BP, FOR)`. Therefore, when adding a new scheme, a programmer only needs to implement the atom and leave the fusions to be generated by G-CasDec. Thanks to such ease of extension, there are currently ~30 fusions defined in G-CasDec, each in one line of macro, in contrast to FastLanes-CPU that only has 9 cascades, the kernels of which would require excessive hand-tuning (though no GPU implementations are available yet).

During the epilogue of compression, the fusions are encoded into the bytecode by the *fuser*, which automates the fusion procedure. First, the macros are expanded into a table of instruction list patterns that can be looked up by the fuser to be automatically replaced with a fused instruction (e.g., `FuseBPFOR` in Figure 5). Second, the fuser works in the fashion of compiler optimization, that is, *best-effort*, to ensure functionality. When a scheme requires inter-thread communication, for example Patch, it will be left intact by the fuser: unfused execution will always serve as a fallback, guaranteeing that any cascade can be correctly decompressed.

6.3 Optimization Passes and Buffer Elision

Our design also supports compiler-based optimization passes that transform the bytecode instructions. Section 6.2 describes one such optimization, i.e., register-level scheme fusion. We next introduce another important optimization as the *elision* of unused buffers to reduce SMEM usage and increase SM occupancy. We emphasize that this optimization framework is extensible, and other general or LWC-specific optimizations can be added as well.

For buffer elision, one can observe that if the top chain C_0 can be fused entirely, the buffer B_0 will become unnecessary. This is because the input is from the child chains' buffers, and the output decompressed data can be directly utilized by the query engine for computation (because decompression can be *fused* with queries) or written back to GMEM for future use. An optimization pass is thus devised to elide the buffer to save precious SMEM space. As in Figure 5, after buffer elision, the number of scratch SMEM buffers is reduced to 1, meaning that we only have to materialize the running values of RLE to SMEM during the entire execution of the tree.

7 Extending G-CasDec

Extending to new integer schemes. LWC schemes are decoupled in G-CasDec, so the effort to add a new scheme does not grow with the number of schemes. Adding a new scheme primarily requires manual implementation of its own bytecode, the compressor, and the decompressor atom, which are independent of other schemes. The number of lines of code (LoC) to implement each scheme is shown in Table 3.

Each fusion can be added with a *single* line of macro, which expands into both the bytecode definition and the decompressor that is composed of atoms. The listed fusions, however, are not exhaustive and can be adjusted to cater to the user's data distribution. Currently, there are ~30 fusions defined for Public BI with the current 8 LWC schemes. Notably, there are more types of cascade trees than the chain-based fusions.

Extending to new data types. While G-CasDec focuses mainly on integer (de)compression, other data types can be encoded into integers for further cascading. Aligned with prior work [5, 27, 28], we dictionary-encode strings into integers in query execution evaluations in Section 8.3. For native support for other data types, we can leverage existing algorithms such as ALP [6, 15] and FSST [10], which encode floating-point numbers and strings into integers, respectively. These schemes can be easily added to G-CasDec in a way similar to other LWC schemes, after which they can be cascaded with other schemes. Specifically, we will show in Section 8.4 how to support floating-point number compression by implementing the ALP scheme. FSST, although hard to cascade in general cases, can be added as an uncascaded special case, or combined with DICT to allow further cascading.

8 Evaluation

G-CasDec is implemented in ~10 000 lines of C++ and CUDA code for the core components of the decompressor, plus ~1 000 lines of C++ code for a simple CPU compressor. In this section, we compare the performance of G-CasDec in terms of both compression ratios and decompression speed against prior SOTA designs.

8.1 Experimental Setup

We use an NVIDIA RTX 5090 GPU for all of our experiments. The GPU features the NVIDIA Blackwell architecture with 170 SMs, a 96 MB L2 cache, 32 GB GDDR7 DRAM of 1.79 TB/s bandwidth, and a PCIe 5.0 x16 interface with bandwidth up to 64 GB/s. The host CPU is an Intel Xeon Gold 6530 processor with 14 vCPUs. The CUDA version is 12.9.

We consider two scenarios. In *warm* execution, which is the default setting, the compressed input data are already in the GPU DRAM, while in *cold* cases, we additionally consider the time to load this compressed input from the host memory over PCIe. In both cases, the CPU compressor time is excluded. Each measurement is warmed up for 3 runs and the running times of the next 17 runs are averaged and reported. We time with CUDA events and flush the L2 cache between runs.

Datasets and benchmarks. We use two benchmarks that are commonly used in prior GPU compression work, for fair and direct comparison. **Public BI** [31] is an open dataset that contains real-world data from Tableau. We specifically use it to run the microbenchmarks. We extract only the *numeric columns*, using the same extraction criteria as those of BtrBlocks [17]. **Star Schema Benchmark (SSB)** [24] contains a single fact table `lineorder` along with several other dimension tables. To saturate the GPU compute resources and remove the biases caused by tail effects, we set the scale factor to 100, so there are about 6×10^8 tuples in the fact table `lineorder`. In line with prior work [5], we test Q1.1, Q2.1, Q3.1 and Q4.1 from SSB using manually written kernels.

Baselines. We compare against the following GPU baselines. Crystal [27] and Crystal-opt [11] support executing SSB queries on uncompressed columnar formats. We only benchmark the more optimized **Crystal-opt** as our uncompressed baseline. **nvCOMP** [22] is a cascaded (de)compression library that specializes in direct decompression of data into GPU DRAM, but cannot execute queries. **Tile-Based** [28] and **FastLanes-GPU** [5] are the SOTA baselines, supporting both direct data decompression and SSB query execution. We do not include other GPU designs [12, 21] because they are significantly slower than Tile-Based and FastLanes-GPU.

We also include two CPU cascaded compression baselines, **BtrBlocks** [17] and **FastLanes-CPU** [4], for comparison with their compression ratios. Their decompression speeds are far behind those of the GPU baselines, so we omit the performance benchmarks.

Table 4. **Comparison of overall compression ratios.** – On all columns of Public BI. Higher is better.

Platform	Design	Compression ratio
GPU	G-CasDec	6.47
	FastLanes-GPU	2.32
	Tile-Based	5.15
	nvCOMP	5.27
CPU	FastLanes-CPU	6.58
	BtrBlocks	6.68

8.2 Microbenchmark Results

We first evaluate the performance of G-CasDec on the Public BI dataset in terms of the columnar data compression ratios and the speed of two workloads: (1) decompressing and writing back data to GMEM, and (2) decompressing and aggregating data into a single value.

8.2.1 Compression ratio. To comprehensively evaluate the compression ratio, we run the compressor (cf. Section 3) of G-CasDec and the baselines on all the numeric columns in the Public BI dataset on CPUs and record the compression ratio of each column.

Table 4 shows the overall compression ratio (the raw size of the entire dataset divided by its compressed size). We can see that G-CasDec matches the state-of-the-art CPU cascaded compressors BtrBlocks and FastLanes-CPU, and outperforms all GPU baselines. Note that CPU designs have less need to trade off compressibility for decompression speed, yet G-CasDec is able to match these high bars. FastLanes-GPU has the lowest compression ratio because it only supports BP and Delta. Tile-Based and nvCOMP have low compression ratios due to their inability to extend to more LWC schemes such as Patch that is very effective on real-world data. The advantage of G-CasDec is mainly due to two features: variable-length data compression, and general support for flexible cascaded schemes. Given the good extensibility of G-CasDec, it can be expected to achieve even better compression ratios with only simple extensions.

Figure 7 shows the cumulative distribution (CDF) of per-column compression ratios. G-CasDec closely follows the two CPU compressors until the highly compressible region, where G-CasDec chooses to avoid complex compression schemes or add extra metadata to speed up decompression. The small step around the compression ratio 2^5 comes from the FastLanes-Delta base values stored to parallelize prefix sums. Other GPU designs all exhibit noticeable gaps to the CPU algorithms.

Compression speed. While we do not focus on compression speed (on CPUs), it is currently acceptable. G-CasDec takes 2071 s to compress the whole Public BI dataset, while FastLanes-CPU takes 323 s and BtrBlocks takes 106 s. No previous GPU design has reported compression time. FastLanes-CPU implements a sampling-based method that makes trials on 3/64 vectors per chunk, so applying the same method on G-CasDec is expected to bring a 20× speedup.

8.2.2 Decompression speed. The first microbenchmark reads compressed values from GMEM, decompresses them all, and writes results back to GMEM. We record the time to perform such decompression for each column in Public BI. Since the tables are too small to saturate the GPU, we replicate the compressed data within GMEM by ≈ 100 times (up to 4096, depending on table sizes) and perform a batched decompression to obtain stable measurements.

The decompression rows of Table 5 present the total decompression time over all columns. While G-CasDec achieves SOTA performance, the performance lead is limited by the fact that this workload is heavily DRAM-bound. Since the size of uncompressed data written back to GMEM

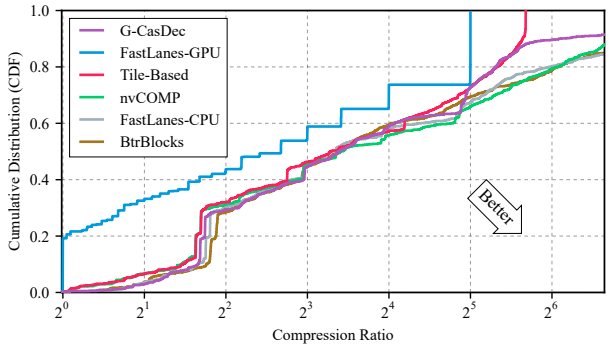


Fig. 7. **Comparison of per-column compression ratio distributions.** – On Public BI.

Table 5. **Comparison of total decompression and aggregation time.** – On all columns of Public BI.

Workload	**Metric**	G-CasDec	FastLanes-GPU	Tile-Based	nvCOMP
Decompression	Time (ms)	15.273	17.637	15.673	17.185
	Perf diff.	/	1.15×	1.03×	1.13×
Aggregation	Time (ms)	2.581	5.222	11.210	N/A
	Perf diff.	/	2.02×	4.34×	N/A

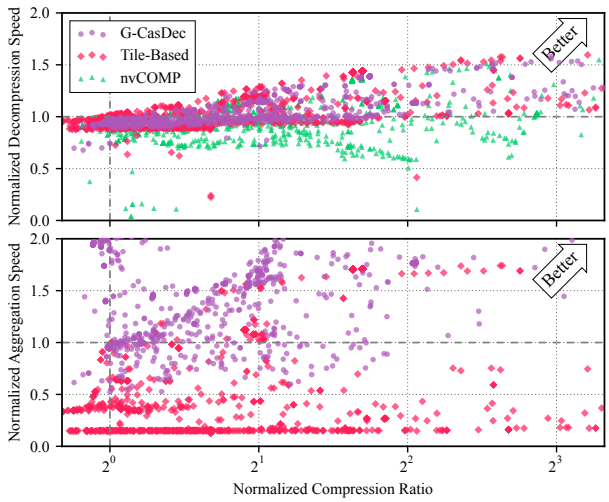


Fig. 8. **Decompression and aggregation speedups vs. compression ratio improvements over FastLanes-GPU.** – On Public BI. The top panel is decompression; the bottom panel is aggregation.

exceeds the compressed data by more than 5×, the overall throughput is dominated by the memory writeback traffic rather than the efficiency of the decompression algorithm itself.

Table 6. **Comparison of SSB query throughput under warm execution setting.** – In 10^9 tuples per second. Higher is better.

Query	G-CasDec	FastLanes-GPU	Tile-Based	Crystal-opt
Q1.1	524.6	437.1 (1.20×)	170.3 (3.08×)	235.3 (2.23×)
Q2.1	85.6	78.3 (1.09×)	72.3 (1.18×)	76.7 (1.12×)
Q3.1	86.4	45.9 (1.88×)	38.2 (2.26×)	42.4 (2.04×)
Q4.1	76.8	57.1 (1.35×)	52.6 (1.46×)	56.4 (1.36×)

The top panel of Figure 8 shows a scatter plot of per-column decompression speed versus compression ratio, both normalized to FastLanes-GPU. Each point corresponds to a column. In almost all cases, G-CasDec is faster than both nvCOMP and FastLanes-GPU, and has much better compression ratios than FastLanes-GPU.

8.2.3 Aggregation speed. To better evaluate the performance of the decompression kernels, we further measure the aggregation speed. This benchmark reads compressed data from GMEM, decompresses *without* writing back, and instead sums up all values into a single result. This benchmark challenges the decompressor kernel to the extreme, as its performance relies on both the compression ratio (determining how many input data are accessed) and the decompression speed. It also better represents real-world query patterns than direct decompression, as analytical queries typically aggregate data in-situ rather than materializing all underlying values. FastLanes-GPU also adopted this microbenchmark in their evaluation. Since nvCOMP does not support query execution, we exclude it here.

The aggregation rows of Table 5 show the total aggregation time taken to decompress and aggregate the entire dataset. G-CasDec achieves 2.02× and 4.34× speedups over FastLanes-GPU and Tile-Based, respectively, thanks to its optimizations such as chain decomposition and bytecode instructions. We conduct more detailed ablation studies in Section 8.5. The bottom panel of Figure 8 shows the per-column scatter plot, again normalized to FastLanes-GPU. In most cases, G-CasDec is both faster than FastLanes-GPU and provides significantly better compression ratios. In some rare cases, G-CasDec is slower than FastLanes-GPU. This is due to a suboptimal implementation of Patch in our system, which can be optimized similarly to G-ALP [15] in future work.

8.3 End-to-End Queries

Now we evaluate end-to-end query execution with G-CasDec and the baselines, on SSB with queries Q{1-4}.1. Aligned with the baselines [5, 28], each query is implemented as a single GPU kernel, inlining the query operators with the decompression without GMEM materialization. Q2.1–Q4.1 contain up to 4 joins, and hash join is implemented by hashing the dimension tables and probing them with the decompressed fact table tuples. G-CasDec currently does not support cases where query fusion is not possible, e.g., joins between two large fact tables. Table 6 summarizes the throughput in tuples per second, under the warm execution setting (input compressed data are in GMEM). G-CasDec outperforms the best baseline, FastLanes-GPU, by 1.20×, 1.09×, 1.88×, and 1.35× for the four queries. The speedups are more significant when compared to Tile-Based and the uncompressed Crystal-opt.

We take Q1.1 as a representative, which was previously shown to be a DRAM-bound workload [11]. A key column that G-CasDec compresses better than FastLanes is `lo_orderdate`, which contains date-type integers between 19920101 and 19980802 with each number recurring multiple times. This column is needed in all four queries. G-CasDec applies an RLE and compresses the running values with FOR, exactly the same way as in Figure 4, achieving a compression ratio of

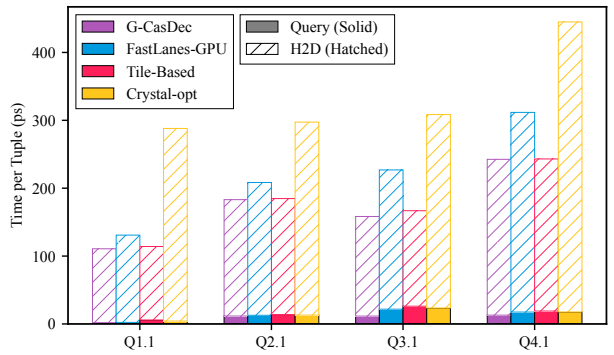


Fig. 9. **SSB query time breakdown under cold execution setting.** – The top hatched part and bottom solid part are the PCIe transfer time from host to device (H2D) and query execution time, respectively.

4.63. On the other hand, FastLanes-GPU performs a manual FOR on it and applies BP, resulting in a compression ratio of only 2.00. The smaller input data size in G-CasDec saves significant GMEM bandwidth. FastLanes-CPU, which also implements RLE, achieves a compression ratio of 3.14 on this column due to the lack of variable-length data compression support.

We further consider the cold execution setting, which additionally includes the time to copy data from the host memory to GMEM (host to device, H2D). Figure 9 shows both PCIe transfer time and query execution time. The overall performance is now affected more by the compression ratio than in the warm setting, as the dominant PCIe transfer time is proportional to the compressed data size. Consequently, Tile-Based, with higher compression ratios, outperforms FastLanes-GPU and becomes the fastest baseline. This shows that the two previous designs exhibit the compressibility-performance tradeoff, as in Table 2. Nevertheless, although the synthetic SSB dataset provides few opportunities for extra compression to reduce PCIe transfer, G-CasDec is able to achieve 1.18× to 1.43× speedups over FastLanes-GPU, and 1.00× to 1.05× over Tile-Based, and remains the best under this cold execution setting. Therefore, we can conclude that G-CasDec successfully advances this Pareto frontier.

8.4 Extensibility to Floating-Point Data Types

To demonstrate the extensibility of G-CasDec to data types other than integers, we implement the state-of-the-art ALP algorithm [6] for floating-point compression as a regular LWC scheme in G-CasDec. ALP is a lossless compression algorithm that encodes floating-point numbers into integers and scale factors; values that cannot be represented exactly are stored as exceptions. In the current implementation, ALP can only serve as the root node in the cascade tree. We also disable a few complicated schemes for performance considerations. We compare against the official GPU implementation of ALP, G-ALP [15], on the same 37 Public BI columns converted to `float32` in their paper, on the same microbenchmarks as in Section 8.2.

Table 7 shows that G-CasDec improves the total compression ratio by 1.05× over G-ALP (from 1.39 to 1.46), enabled by the more flexible cascade selection in G-CasDec versus hard-coded ones in G-ALP. Decompression and aggregation are 1.11× and 1.42× slower, respectively, because G-CasDec prioritizes generic bytecode execution over G-ALP's specialized hand-tuned kernels.

Support for ALP in G-CasDec comprises ~1000 LoC, of which ~800 implement the algorithm itself due to its complicated compressor, and only ~200 are modifications to the framework, which are

Table 7. **Comparison of compression ratios, decompression time, and aggregation time, with floating-point data.** – On Public BI. Lower time is better; higher compression ratio is better.

Design	Compression ratio	Decompression (ms)	Aggregation (ms)
G-ALP	1.39	0.889	0.330
G-CasDec	1.46	0.983	0.468

Table 8. **Impact of aligned batching on compression ratio.** – On Public BI columns which have variable-length intermediate results. Higher is better.

Design	Compression ratio
G-CasDec	6.48
w/o variable-length compression	5.51(0.85×)
w/o padding	7.20(1.11×)

Table 9. **Impact of chain-based resource management on total decompression and aggregation time.** – On Public BI. Lower is better.

Design	Decompression (ms)	Aggregation (ms)
G-CasDec	15.273	2.581
w/o scheme fusion	15.359	3.548
w/o scheme fusion or SMEM reuse	15.321	4.447

primarily front-end support for additional data types. The result demonstrates that G-CasDec can be easily extended to new data types by adding a new LWC scheme within the generic framework.

8.5 Ablation Studies

Aligned batching. Aligned batching (Section 4) enables cascaded compression on variable-length data. We find that in Public BI [31], 16% of the columns have at least one chunk of variable-length intermediate results and can benefit from aligned batching. Table 8 lists the compression ratios on these columns. Without aligned batching and variable-length compression, the overall compression ratio drops by 15% from 6.48 to 5.51. In particular, in column 45 of `CommonGovernment_1`, the running values of RLE can be further compressed by Dict and FOR, achieving a 1.65× higher compression ratio than FastLanes-CPU, where variable-length data are left uncompressed.

On the other hand, aligned batching incurs padding overheads which hurt compression ratios. The last row in Table 8 shows the compression ratio when we remove padding in variable-length compression. The 11% difference is the padding overhead, which is minor. Nevertheless, the minor loss in compressibility enables regular parallel execution and memory accesses, demonstrating a well-justified tradeoff in G-CasDec.

Chain-based resource management. To see the performance impact of chain-based SMEM buffer reuse (Section 5.2) and register-level scheme fusion (Section 5.3), we compare G-CasDec with its variants that disable scheme fusion alone or both scheme fusion and SMEM reuse.

As shown in Table 9, while decompression to GMEM shows a minor performance difference of <1% due to the dominant result writeback stage, our two optimizations have a significant impact on the aggregation workload. Chain-based SMEM reuse provides a 1.25× speedup, while scheme fusion further improves by another 1.37×. The overall benefits can be up to 1.72×.

Table 10. **GPU performance indicators when running aggregation on SSB `lo_orderdate`.** – For kernel duration, executed instructions, average stall cycles per instruction, and SMEM usage per block, lower is better; for DRAM utilization, and achieved occupancy (active warps divided by maximum warps per warp scheduler), higher is better.

Design	Kernel duration (ms)	Executed instructions (B)	Average stall cycles	DRAM utilization (%)	SMEM per block (kB)	Occupancy (%)
G-CasDec	0.47	0.34	7.10	65.0	4.22	36.4
w/o aligned decompression	2.13	1.08	10.45	13.7	4.00	37.9
w/o scheme fusion	1.09	0.49	5.86	27.9	8.22	19.2
w/o scheme fusion or SMEM reuse	1.38	0.49	5.41	21.4	12.22	14.0

GPU performance indicators. Table 10 shows the GPU kernel profiling results of aggregation on SSB column `lo_orderdate`. We leverage these in-depth hardware statistics to further explain the performance improvements brought by each technique in G-CasDec.

For aligned batching, we implemented an *unaligned* variant of G-CasDec that decompresses variable-length intermediate results directly, similar to Tile-Based. Compared with this unaligned variant, G-CasDec is 4.5× faster. It executes 3.2× fewer instructions because when data execution is aligned to vector sizes, the compiler can fully unroll the loops and reduce the control flow instructions. The average stall cycles per instruction decrease from 10.45 to 7.10, because (1) unrolled loops also allow load instructions to be hoisted by compilers and executed in parallel, maximizing instruction-level parallelism, and (2) GMEM accesses are maximally coalesced, improving load instruction throughput, as stated in Section 4.2. As a result, the DRAM utilization improves from 13.7% to 65.0%, greatly improving the performance in this memory-bound workload.

For the two techniques of chain-based resource management, SMEM reuse can reduce the per-block SMEM from 12.22 kB to 8.22 kB, and buffer elision enabled by scheme fusion further reduces it to 4.22 kB. Together they increase the occupancy from 14.0% to 36.4%.

9 Conclusions

We present G-CasDec, an optimized framework that realizes general and efficient cascaded decompression on GPUs. G-CasDec breaks the tradeoff, improves the Pareto frontier between compression ratios and decompression speed, and introduces an extensible approach to supporting arbitrary cascades. The key techniques in G-CasDec include an aligned batching scheme for variable-length data, efficient management of shared memory and registers with a chain-based decompression flow, and a set of bytecode instructions for flexible control. Our evaluation shows that G-CasDec outperforms existing designs in both compressibility and performance.

Acknowledgments

The authors thank the anonymous reviewers for their valuable suggestions. This work was supported by the National Natural Science Foundation of China (Grant No. 62532001), Tsinghua University Initiative Scientific Research Program, and Tsinghua University - Keystone Electrical (Zhejiang) Co., Ltd Joint Research Center for Embodied Multimodal Artificial Intelligence (JCEMAI). Mingyu Gao and Huanchen Zhang are the co-corresponding authors.

References

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280. doi:10.1561/1900000024

[2] Daniel J. Abadi, Samuel R. Madden, and Miguel C. Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. Association for Computing Machinery, 671–682. doi:10.1145/1142473.1142548

[3] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proceedings of the VLDB Endowment (VLDB '23)* 16, 9 (2023), 2132–2144. doi:10.14778/3598581.3598587

[4] Azim Afroozeh and Peter Boncz. 2025. The FastLanes File Format. *Proceedings of the VLDB Endowment (VLDB '25)* 18, 11 (2025), 4629–4643. doi:10.14778/3749646.3749718

[5] 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 (DaMoN '24)*. Association for Computing Machinery, Article 8, 11 pages. doi:10.1145/3662010.3663450

[6] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless Floating-Point Compression. *Proceedings of the ACM on Management of Data (SIGMOD '24)* 1, 4, Article 230 (2023), 26 pages. doi:10.1145/3626717

[7] Apache Parquet. 2013. Apache Parquet. https://parquet.apache.org/

[8] Felipe Aramburú, William Malpica, Kaouther Abrougui, Amin Aramoon, Romulo Auccapuclla, Claude Brisson, Matthijs Brobbel, Colby Farrell, Pradeep Garigipati, Joost Hoozemans, Supun Kamburugamuve, Akhil Nair, Alexander Ocsa, Johan Peltenburg, Rubén Quesada López, Deepak Sihag, Ahmet Uyar, Dhruv Vats, Michael Wendt, Jignesh M. Patel, and Rodrigo Aramburú. 2025. Theseus: A Distributed and Scalable GPU-Accelerated Query Processing Platform Optimized for Efficient Data Movement. *arXiv preprint arXiv:2508.05029* (Aug. 2025). https://arxiv.org/abs/2508.05029

[9] BlazingDB. 2018. BlazingSQL. https://github.com/BlazingDB/blazingsql

[10] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proceedings of the VLDB Endowment (VLDB '20)* 13, 11 (2020), 2649–2661. doi:10.14778/3407790.3407851

[11] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proceedings of the VLDB Endowment (VLDB '24)* 17, 3 (2023), 441–454. doi:10.14778/3632093.3632107

[12] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proceedings of the VLDB Endowment (VLDB '10)* 3, 1 (2010), 670–680. doi:10.14778/1920841.1920927

[13] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2021. NestGPU: Nested Query Processing on GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE '21)*. IEEE, 1008–1019. doi:10.1109/ICDE51399.2021.00092

[14] HEAVY.AI. 2017. HeavyDB. https://github.com/heavyai/heavydb

[15] Sven Hepkema, Azim Afroozeh, Charlotte Felius, Peter Boncz, and Stefan Manegold. 2025. G-ALP: Rethinking Light-Weight Encodings for GPUs. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, Article 11, 10 pages. doi:10.1145/3736227.3736242

[16] Aaron Jarmusch, Nathan Graddon, and Sunita Chandrasekaran. 2025. Dissecting the NVIDIA Blackwell Architecture with Microbenchmarks. *arXiv preprint arXiv:2507.10789* (July 2025). https://arxiv.org/abs/2507.10789

[17] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proceedings of the ACM on Management of Data (SIGMOD '23)* 1, 2, Article 118 (2023), 26 pages. doi:10.1145/3589263

[18] Daniel Lemire and Leonid Boytsov. 2015. Decoding Billions of Integers per Second Through Vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. doi:10.1002/spe.2203

[19] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proceedings of the VLDB Endowment (VLDB '23)* 16, 11 (2023), 3044–3056. doi:10.14778/3611479.3611507

[20] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and Dissecting the NVIDIA Hopper GPU Architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS '24)*. IEEE, 656–667. doi:10.1109/IPDPS57955.2024.00064

[21] Antonio Mallia, Michał Siedlaczek, Torsten Suel, and Mohamed Zahran. 2019. GPU-Accelerated Decoding of Integer Lists. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM '19)*. Association for Computing Machinery, 2193–2196. doi:10.1145/3357384.3358067

[22] NVIDIA Corporation. 2020. nvCOMP. https://developer.nvidia.com/nvcomp

[23] NVIDIA Corporation. 2026. CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html#warps-and-simt

[24] Patrick O'Neil, Elizabeth O'Neil, and Xuedong Chen. 2007. The Star Schema Benchmark. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF

[25] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proceedings of the VLDB Endowment (VLDB '13)* 6, 11 (2013), 1080–1091. doi:10.14778/2536222.2536233

[26] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-Precision. In *Advances in Neural Information Processing Systems (NeurIPS '24)*, Vol. 37. Curran Associates, Inc., 68658–68685. doi:10.52202/079017-2193

[27] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. 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 (SIGMOD '20)*. Association for Computing Machinery, 1617–1632. doi:10.1145/3318464.3380595

[28] 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 (SIGMOD '22)*. Association for Computing Machinery, 1390–1403. doi:10.1145/3514221.3526132

[29] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. Association for Computing Machinery, 553–564.

[30] Uber Technologies, Inc. 2018. AresDB. https://github.com/uber/aresdb

[31] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems (DBTest '18)*. Association for Computing Machinery, Article 1, 6 pages. doi:10.1145/3209950.3209952

[32] Wikipedia contributors. 2026. Multi-Pass Compiler. https://en.wikipedia.org/wiki/Multi-pass_compiler

[33] Wikipedia contributors. 2026. X Macro. https://en.wikipedia.org/wiki/X_macro

[34] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proceedings of the ACM on Management of Data (SIGMOD '25)* 3, 1, Article 39 (2025), 27 pages. doi:10.1145/3709689

[35] Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei Gong, and Xiangyao Yu. 2026. Rethinking Analytical Processing in the GPU Era. In *Proceedings of the 16th Annual Conference on Innovative Data Systems Research (CIDR '26)*. VLDB Endowment, 7 pages. https://www.vldb.org/cidrdb/2026/rethinking-analytical-processing-in-the-gpu-era.html

[36] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proceedings of the VLDB Endowment (VLDB '13)* 6, 10 (2013), 817–828. doi:10.14778/2536206.2536210

[37] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment (VLDB '24)* 17, 2 (2023), 148–161. doi:10.14778/3626292.3626298

[38] Xinyu Zeng, Ruijun Meng, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2024. NULLS!: Revisiting Null Representation in Modern Columnar Formats. In *Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN '24)*. Association for Computing Machinery, Article 10, 10 pages. doi:10.1145/3662010.3663452

[39] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. 2025. F3: The Open-Source Data File Format for the Future. *Proceedings of the ACM on Management of Data (SIGMOD '26)* 3, 4, Article 245 (2025), 27 pages. doi:10.1145/3749163

[40] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE '06)*. IEEE, 59. doi:10.1109/ICDE.2006.150

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026