ZipFlow: a Compiler-based Framework to Unleash Compressed Data Movement for Modern GPUs

Gwangoo Yeo
KAIST, South Korea
gwangoo525@kaist.ac.kr

Zhiyang Shen
Tsinghua University, China
shenzhiy21@mails.tsinghua.edu.cn

Wei Cui*
Microsoft, CA
weicu@microsoft.com

Matteo Interlandi
Microsoft, US
matteo.interlandi@microsoft.com

Rathijit Sen
Microsoft, US
rathijit.sen@microsoft.com

Bailu Ding
Microsoft, US
badin@microsoft.com

Qi Chen
Microsoft, CA
cheqi@microsoft.com

Minsoo Rhu
KAIST, South Korea
minsoo.rhu@gmail.com

ABSTRACT

Modern Lakehouse architectures store data in open, columnar formats to enable cross-engine interoperability, and GPU-accelerated data analytics is increasingly becoming one of these engines. In GPU-accelerated analytics, data transfer from CPU to GPU becomes a performance bottleneck due to limited PCIe bandwidth. Data compression is widely used to reduce transfer volume while leveraging GPUs for decompression. However, to optimize end-to-end query performance, the workflow of compression, transfer, and decompression must be holistically designed based on the compression strategy and hardware characteristics to balance I/O latency and computational overhead.

In this work, we present *ZipFlow*, a compiler-based framework for optimizing compressed data movement in GPU-accelerated data analytics. ZipFlow classifies compression algorithms into three distinct patterns based on their inherent parallelism. For each pattern, ZipFlow applies generalized scheduling strategies to effectively exploit GPU computational resources across diverse architectures. Building on these patterns, ZipFlow provides flexible optimizations allowing to improve end-to-end query latency by 2.08× over the state-of-the-art GPU compression library (nvCOMP), and by up to 3.14× over a state-of-the-art CPU engines.

PVLDB Reference Format:
Gwangoo Yeo, Zhiyang Shen, Wei Cui, Matteo Interlandi, Rathijit Sen, Bailu Ding, Qi Chen, and Minsoo Rhu. ZipFlow: a Compiler-based Framework to Unleash Compressed Data Movement for Modern GPUs. PVLDB, 19(12): 3888 - 3901, 2026.
doi:10.14778/3827998.3828003

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zipflow-artifact/vldb2026_zipflow.

*Corresponding author: weicu@microsoft.com

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828003

1 INTRODUCTION

GPUs have become widely available with the rapid development of artificial intelligence, such as deep neural networks and large language models. Interestingly, given the powerful computation capability and high memory bandwidth of modern GPUs, they also exhibit great potential for analytical database workloads. Recent work has demonstrated the promise of leveraging GPUs to speed up analytical database systems, including efforts from both academia and industry [2, 33, 37–39, 41, 48, 67].

Despite such promising results, the constrained GPU memory capacity has been a critical limitation for GPU-accelerated database systems. For example, high-end GPUs such as NVIDIA H100 [6] only have up to 80 GB of memory capacity, while analytical databases often scale to hundreds of GB or even several TBs. Although using multiple GPUs connected with high-performance interconnect (i.e., NVLink [14]) can increase the memory capacity at high memory bandwidth, such hardware can be prohibitively expensive. As the database cannot be fully cached in the GPU memory, the data often needs to be transferred from CPU to GPU via the PCIe bus on demand during query execution [25, 27, 32, 68, 76]. Unfortunately, the data transfer often becomes a performance bottleneck due to the slow PCIe bus. For example, Figure 1 shows the query latency breakdown of the 22 TPC-H queries on CPU-based and GPU-based database systems. Although the query execution on A100 [1] is more than 15× faster than that on CPU, due to the slow data transfer from CPU to GPU, the speedup of the end-to-end query latency is much more moderate.

This matters even more in today's Lakehouse-style stacks where data is stored in open, columnar formats for cross-engine interoperability. As GPU-accelerated analytics becomes one of the consumers of these shared datasets [43], efficiently moving (and decoding) columnar data into GPUs becomes a system-level requirement. A prime example of this ecosystem is Microsoft Fabric [54] which stores tabular data in Parquet [4] format in OneLake [55] so that multiple compute engines can operate on a single copy of data. Fortunately, to reduce storage footprint, data in the lake is typically stored in compressed format. Hence, the system can transfer the compressed data fetched from the data lake directly to the GPU, where it is decompressed prior to query processing. This design

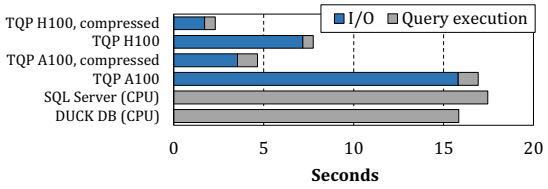

Figure 1: Latency breakdown of 22 TPC-H queries (SF=100) on CPU (AMD EPYC 7V12) [8, 12], A100 (PCIe-4), and H100 (PCIe-5), TQP [37] is used for query processing on GPU.

choice reduces the amount of data transferred and simultaneously leverages the computational power of the GPU. While data compression has become the status quo for analytical database systems, we observe that existing data compression techniques are designed for CPU-based database systems and do not fully exploit the unique performance characteristics of GPU-accelerated database systems.

First, GPUs offer massive computation that can enable more aggressive compression strategies, improving end-to-end query performance. Existing techniques often limit algorithm choices and their nesting to strike a balance between the compression ratio and decompression overhead. For example, Parquet only considers compute-efficient compression algorithms, such as run-length-encoding (RLE), delta encoding, dictionary encoding, and bit-packing, while more expensive compression algorithms, such as LZ77-based and entropy-based algorithms, can lead to much higher compression ratios on some workloads. Even state-of-the-art GPU libraries like nvCOMP [13] limit nesting to specific combinations of RLE, delta encoding, and bit-packing.

Second, compression schemes must be tailored to the target GPU hardware to achieve optimal performance. While specialized algorithms that adapt to data characteristics can be highly effective [20, 24], mapping them respectively onto a wide variety of GPU architectures entails a substantial overhead for adapting utilization on each target platform. Existing GPU compression frameworks [13, 19, 68] provide libraries of pre-built kernels only for conventional algorithms, while manually extending these frameworks to incorporate new or custom compression techniques is both cumbersome and inefficient. In addition, these frameworks often lack the support for optimizing nested compression schemes, i.e., no kernel fusion for cross-stage optimization with pre-built kernels. Furthermore, achieving seamless kernel compatibility across heterogeneous GPU architectures, such as support for NVIDIA GPUs via CUDA [5] and AMD GPUs via ROCm [10], continues to present a formidable challenge for existing frameworks.

Finally, optimizing end-to-end query performance requires holistic design of the data compression, transfer, and decompression workflow. This includes exploring the tradeoffs between compression ratio, decompression speed, and data transfer schedules to minimize overall latency. While prior work has studied the performance of individual components [19, 45, 51, 68] or examined designs with a limited set of algorithms [49, 58], such localized decision-making can lead to suboptimal end-to-end performance.

To address these gaps, we propose ZipFlow, a flexible compiler-based compression framework for modern GPUs in Lakehouse settings [43]. ZipFlow abstracts commodity compression algorithms into three parallel compute patterns. These patterns serve as building blocks, enabling flexible and generalized device scheduling of both existing and new custom algorithms. Each pattern is easily optimized to exploit its parallelism across diverse GPU architectures with ZipFlow's matching compiler technique. By holistically exploring the tradeoffs between I/O and computation overhead of the compressed data transfer workflow, ZipFlow delivers optimized end-to-end query performance. Our evaluation on the industry-standard TPC-H benchmark demonstrates the benefits of ZipFlow. Our custom compression methods provide superior compression ratios, with 1.85× savings in I/O latency compared to a state-of-the-art compression GPU library [13]. Meanwhile, ZipFlow also provides a 3.26× speedup in decompression with automated compiler optimizations. This leads to an average speedup of 2.08× in end-to-end query latency compared to the baseline.

In summary, this work makes the following contributions:

- We abstract three patterns to encapsulate the core parallelism structures commonly found in (de)compression algorithms (Section 3). These patterns enable native optimization of existing and custom (de)compression algorithms on GPUs, which can also facilitate development of a broader set of GPU-friendly (de)compression algorithms in the future.
- For each pattern, we formulate a universal optimization space attuned to GPU execution geometries and the SIMT paradigm (Section 4). Combined with fusion and pipelining, it holistically minimizes the end-to-end compressed data transfer overhead.
- We conduct a head-to-head comparison between ZipFlow and the state-of-the-art nvCOMP on identical algorithms across a wide range of data distributions (Section 5.2), highlighting that our hardware-aware approach effectively outperforms the state-of-the-art GPU compression library (i.e., nvCOMP).
- We comprehensively evaluate ZipFlow on the industry-standard TPC-H benchmark (Section 5.4), demonstrating the impact of joint optimization of the compressed data transfer workflow on the end-to-end query performance. Finally, we showcase how ZipFlow enables tailored hardware-aware execution by evaluating its performance on heterogeneous GPUs.

This paper takes a first step toward GPU-centric compression in unified Lakehouse platforms such as Microsoft Fabric, improving end-to-end performance as GPU-accelerated analytics becomes mainstream [38, 43, 48, 60].

2 BACKGROUND

2.1 Existing Compression Algorithms

In this section, we present a summary of widely used lossless compression algorithms, covering both lightweight schemes surveyed by Damme et al. [28] and heavyweight algorithms relevant to modern systems. We also illustrate an example diagram for each algorithm in Figure 2, and discuss their key design ideas.

Dictionary encoding replaces data with a *dictionary* and an *index*. The *dictionary* stores unique values from data, while the *index* maps original data values to their corresponding entries in the dictionary. Dictionary encoding tables can be sorted or unsorted, with different trade-offs between build cost and querying capabilities.

Delta encoding replaces each value with the difference (delta) between the current value and previous value, while storing the initial value as a base. Delta encoding alone does not provide compression, but it enables further compression with RLE or bit-packing.

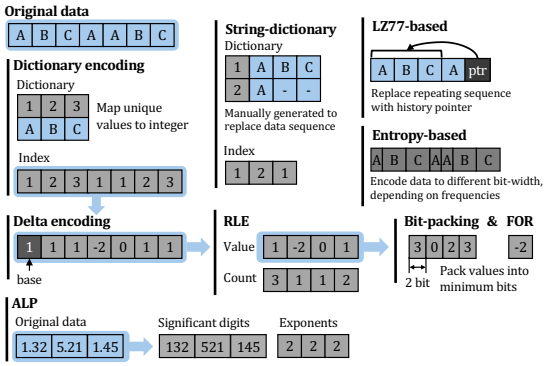

Figure 2: Compression algorithms of different families.

Run-Length Encoding (RLE) minimizes consecutive repeating integer values by replacing them with a *value* and a *count*. The *value* stores the repeated value, while the *count* records the number of repetitions.

Bit-packing and Frame of Reference (FOR) pack integers into the minimum bit-width required for the given data. The bit-width is calculated using minimum and maximum values, while the minimum value is stored separately as a FOR.

ALP [20] separates significant digits and exponents from floating-point values, converting them into integers, enabling further compression. This approach is effective for compressing columns with limited decimal precision. While we use ALP as the representative algorithm in this paper, the core idea has been explored in prior works [20, 32, 40, 45].

String-dictionary substitutes repeated byte sequences with dictionary indices. The compression ratio depends on how well it de-duplicates the patterns. Recent algorithms [24] support automated dictionary construction from high-frequency byte sequences in the dataset. Compressing the dictionary table itself has also been explored in prior work [57].

Entropy-based algorithms such as ANS [9, 31] and Huffman encoding [42] compress the data by exploiting the skewness of data occurrences. Fewer bits are assigned to frequently occurring data to reduce overall data size.

LZ77-based algorithms such as LZ4 [11] and Snappy [15] de-duplicate repeated sequences by encoding them as pointers to previous data. These algorithms are referred to as "general-purpose" due to their broad compatibility. State-of-the-art algorithms such as Deflate [30] and zstd [16] combine LZ77 with entropy-based methods for superior compression ratios.

2.2 Existing Approaches in Database Systems

In modern database systems, achieving efficient data compression is often accomplished through the *composition of several lightweight compression techniques*, such as run-length encoding (RLE), delta encoding, dictionary encoding, and bit-packing. This paradigm, frequently described as *cascaded compression*, leverages the strengths of individual algorithms by orchestrating them in a sequence, tailored to the characteristics of the data at hand. Well-designed nested algorithms for individual data files show an effective compression ratio, while offering simple compute patterns [28]. BtrBlocks [45] introduces an automated search mechanism to determine optimized nested algorithms, significantly improving compression ratios.

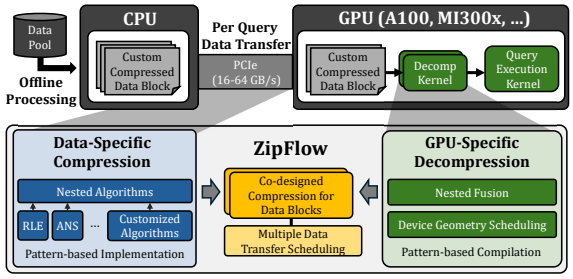

Figure 3: End-to-end pipeline from stored data to ZipFlow.

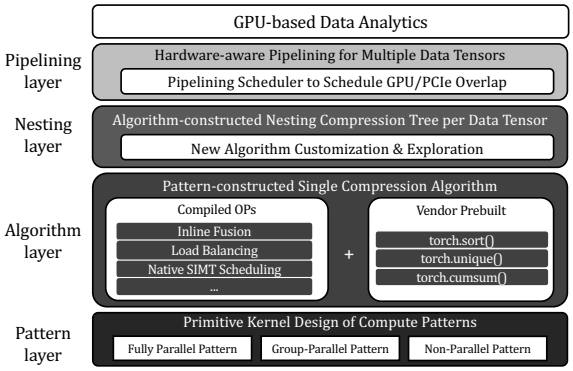

Figure 4: ZipFlow Architecture Overview.

Prominent columnar storage formats such as ORC [61] and Parquet [4] have embraced this cascaded approach by implementing a variety of well-engineered nested compression schemes, such as combining RLE with dictionary encoding or integrating delta encoding with bit-packing. They support further compression by enabling general-purpose algorithms to be applied on top of predefined nesting algorithms. The FastLanes project [18] was introduced to enhance compressed columnar data formats by enabling increased opportunities for parallel decoding, providing several handcrafted GPU kernels, such as bit-packing, delta encoding [19], and ALP [40], aiming to leverage the SIMD and SIMT processing capabilities of CPUs and GPUs. Beyond these lightweight encodings, hardware acceleration has also been applied to stronger string compression schemes on both CPUs [47] and GPUs [21, 72].

3 DESIGN

ZipFlow is a framework to accelerate compression-based data movement to GPUs. Figure 3 presents a high-level overview of the end-to-end data movement pipeline as optimized by ZipFlow. Raw data in various formats (e.g., numpy, parquet, text, binary) are first compressed utilizing ZipFlow's flexible nesting and customizable algorithms offline into static base data with deterministic distributions, recording necessary latency-related metadata (original size, PCIe/GPU latency). For online data transfer, ZipFlow facilitates the efficient movement of compressed data from the host across the PCIe interconnect, followed by a high-throughput, device-specific decompression phase. This end-to-end process is co-designed by ZipFlow to fully exploit the heterogeneity of modern GPUs, incorporating not only hardware-native resource management but also optimized overlap between PCIe data transfers and GPU decompression to maximize throughput and minimize latency. Given that compression occurs offline and remains orthogonal to the online

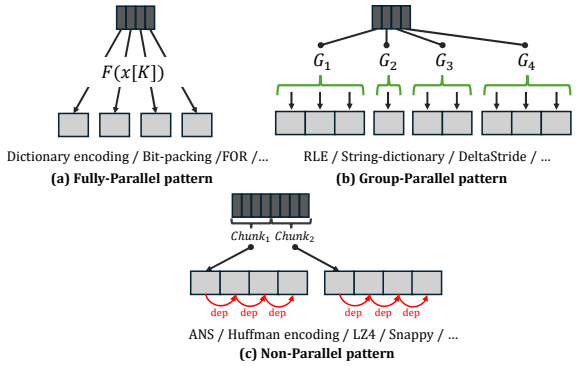

Figure 5: Logical dependencies for ZipFlow parallel patterns.

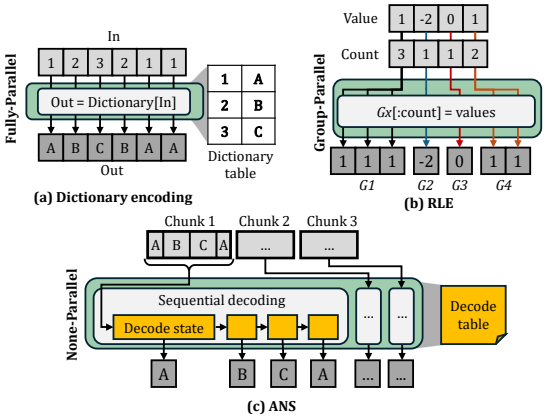

Figure 6: Data mapping example for each pattern.

data transfer process, we focus on decompression, the critical online read path that determines end-to-end data movement latency.

Figure 4 introduces a four-layer design for ZipFlow. Starting from the bottom, the **Pattern Layer** enables the design of primitive GPU operators based on three predefined compute patterns. Each designed operator corresponds to a GPU kernel and serves as a building block for (de)compression algorithms. These GPU operators enable simple implementation of the target algorithm while exploiting the prevalent parallel patterns in (de)compression algorithms. In the **Algorithm Layer**, GPU patterns are organized to form primitive compression algorithms such as RLE, dictionary encoding, bit-packing, or ANS. Based on primitive algorithms from the Algorithm Layer, the **Nesting Layer** explores and customizes the nesting of algorithms applied to data columns (e.g. Dictionary encoding + ANS or RLE + Bit-packing), to achieve optimal end-to-end data movement latency. Finally, at the top is the **Pipelining Layer**, where our framework schedules the pipelining of I/O operations and GPU decompression latency across multiple data chunks to minimize overall latency.

3.1 Pattern Layer

We identify three major parallel schemas based on our observations on commodity (de)compression algorithms. To ensure comprehensive coverage of possible parallelism options, we classify the parallelizable schema into three distinct patterns: *Fully-Parallel*, *Group-Parallel*, and *Non-Parallel*. Given an input and output array, each pattern covers **N to 1**, **1 to N**, and **N to M** independent compute blocks in (de)compression algorithms, respectively.

Fully-Parallel Pattern is tailored for (de)compression operators where each input independently maps to output values with no cross-element dependencies, as shown in Figure 5(a). Although the figure only shows a simple 1:1 scenario, a fixed scalar number of input arrays can exist to support the N to 1 computation block. Any mapping function can be used to define the computation between input and output values, and arbitrary mappings between input and output indices are also supported. Decompression algorithms such as dictionary encoding, bit-packing, ALP [20, 40] and FastLanes [19] algorithms, can be implemented using the Fully-Parallel kernel. In Figure 6(a), we present an example of dictionary-encoded data being decompressed using the Fully-Parallel kernel. The kernel uses a dictionary look-up table as a parallel mapping function, where the `Dictionary` is provided as metadata. This Fully-Parallel kernel performs parallel dictionary look-ups for each element selected by `In` and writes the corresponding value to `Out`.

Group-Parallel Pattern focuses on customizing compression operators for spatially non-uniform distributions, where the entire task is segmented into multiple variable-sized groups $G_1, G_2, .., G_n$ containing independent subtasks within each group. Different sizes of G_x result in an imbalance of execution. For example, Figure 5(b) shows four groups, each containing a different number of elements internally, while the arrows within each group indicate a parallel relationship of 1 to N. Typically, this pattern applies to all RLE-based encoding families including delta encoding, String-dictionary and GSST [72]. We demonstrate how RLE decoding can be mapped to the Group-Parallel pattern in Figure 6(b). Compressed RLE data consists of a `count` array and a `value` array, where the *count* array specifies the number of occurrences for each group, and each element in the `value` array corresponds to an input symbol. In this use case, a direct copy function is used as the mapping function when writing to the output, so that the Group-Parallel kernel replicates each element in the `value` array in parallel, repeating it `count` times per group, and writes the results to the appropriate group indices.

Non-Parallel Pattern is reserved for all other compression methods where parallelization is not feasible with Fully-Parallel and Group-Parallel, such as inherently serial algorithms like Snappy and Huffman coding. We leverage the file chunking feature of the compression algorithm, dividing the target file into smaller chunks that are compressed and decompressed independently for parallel processing. This approach enables different processors to handle independent file chunks concurrently, as illustrated in Figure 5(c), allowing a single GPU processor to process serialized code as if it were implemented for a CPU. Similar to many other serial algorithms, we take ANS decompression as an example. Each intermediate `decode state` in ANS depends on its predecessor to construct the corresponding element and recursively contributes its state to its successor. As shown in Figure 6(c), this workflow enforces a strict, sequential progression of the `decode state` throughout the process within each chunk. However, opportunities for parallelism arise by grouping the intermediate `decode states` from different chunks and dispatching them in a SIMT (Single Instruction, Multiple Threads) manner, which is further explained in Section 4.

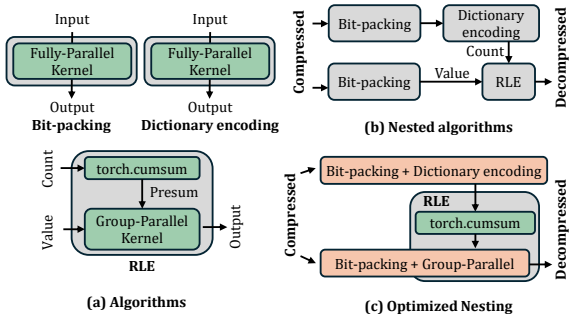

Figure 7: Execution of Algorithm & Nesting layer in PyTorch.

3.2 Algorithm Layer & Nesting Layer

The Algorithm Layer incorporates kernels compiled from the Pattern Layer to construct primitive (de)compression algorithms. The pool of primitive (de)compression algorithms in ZipFlow includes delta encoding, dictionary encoding, String-dictionary, ANS, bit-packing, and ALP, while additional algorithms can be added when necessary, as demonstrated in Section 5. To further simplify the implementation, the framework also supports direct invocation of prebuilt operators from PyTorch's out-of-the-box operations, such as sort, unique, and cumulative sum, as auxiliary operations.

Figure 7(a) illustrates the decompression function design of bit-packing, dictionary encoding, and RLE. Decompression of bit-packing and dictionary encoding requires a single Fully-Parallel kernel, whereas RLE requires two-step kernels for decompression. The variable `count` is first pre-processed with PyTorch's cumulative sum to calculate the base output index which is stored in a variable named `presum`. Next, the Group-Parallel kernel is applied for parallel expansion of the compressed data. In the Nesting Layer, users can implement nested compression algorithms by combining a pool of primitive algorithms from the Algorithm Layer. These nested algorithms are regarded as the eventual (de)compression for target data columns. In Figure 7(b), we show an example of nested decompression using sub-algorithms designed from Figure 7(a), while Figure 7(c) further shows optimization by fusing Fully-Parallel patterns to nearby patterns. Consecutive Fully-Parallel kernels from the bit-packing and the dictionary encoding are fused to calculate the `count` tensor. Similarly, the bit-packing that generates the `value` tensor is also fused with the Group-Parallel kernel inside the RLE function. Revisiting the Pattern Layer for kernel fusion enables further optimization by eliminating redundant off-chip memory movement by keeping intermediate data on-chip (e.g., in registers).

3.3 Pipelining Layer

GPU-based query processing necessitates transferring multiple blocks of data from CPU→GPU. To address the latency introduced by these data transfers and subsequent GPU-side decompression, ZipFlow introduces a dedicated Pipelining Layer. This layer aims to minimize end-to-end query latency by orchestrating the pipelined execution of CPU→GPU data transfers alongside on-GPU decompression tasks, seamlessly overlapping these operations across several data blocks. Our empirical investigations reveal that the sequence in which data blocks are pipelined exerts a considerable impact on the achieved performance improvements. As illustrated in Figure 8, consider two representative data blocks, A and B: data

Algorithm 1 Pipeline Scheduling via Johnson's Algorithm

Require: Blocks b_i required for the query with profiled latencies t_i^{io}, t_i^{dec}
1: $P \leftarrow \{(t_i^{io}, b_i, \texttt{IO})\} \cup \{(t_i^{dec}, b_i, \texttt{DEC})\}$, sort P ascending
2: $L_{IO}, L_{decompression} \leftarrow [\,], [\,]$
3: **while** $P \neq \emptyset$ **do**
4: $\quad (t, b, \text{op}) \leftarrow P$.pop_front()
5: $\quad$ **if** op = IO **then** L_{IO}.append(b) **else** $L_{decompression}$.append(b)
6: $\quad P$.remove((_, b, _))
7: **return** Concat(L_{IO}, reverse($L_{decompression}$))

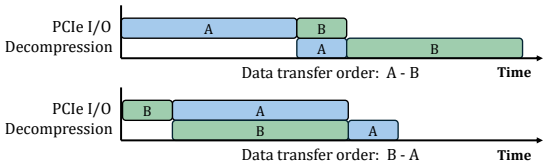

Figure 8: Transfer latency under different pipelining orders.

A is characterized by high data transfer overhead but rapid decompression, whereas data B exhibits the converse pattern, with low transfer cost but prolonged decompression time. Comparing alternative data block transfer orders, initiating the pipeline with data B followed by data A (i.e., B→A) achieves lower overall latency. This performance gain is due to the maximum concurrency between the data movement and decompression phases.

To generalize this optimization for arbitrary queries, it is critical to identify the optimal scheduling order of data transfers for minimum total latency. To this end, we design a specialized pipelining scheduler leveraging Johnson's algorithm [44]. Algorithm 1 details this greedy scheduling based on Johnson's rule. At runtime, the scheduler retrieves offline-profiled PCIe I/O and decompression latencies for all required base data, sorting them into a unified ascending priority queue (line 1). It iteratively extracts the shortest latency entry, appending the associated block to either an *I/O list* or a *decompression list* based on the entry type (lines 3–5); once scheduled, a block's remaining entry is invalidated (line 6). The final execution order is assembled by concatenating the I/O list with the reversed decompression list, prioritizing blocks with high-speed I/O at the front and those with high-speed decompression at the back. This approach efficiently determines the optimal data transfer order with only $O(n \log n)$ complexity.

4 DEVICE GEOMETRY SCHEDULING

The core of ZipFlow's hardware-aware optimization for data movement lies in its cross-GPU optimization spaces, which connect GPU-specific capabilities with custom compression algorithms. Given the diversity of GPU architectures in terms of on-chip resources such as compute units, memory transaction sizes, wavefront sizes, and cache capacities, compilation plays a key role in enabling fine-grained mapping of computational tasks to these heterogeneous native resources. We introduce a three-dimensional configuration vector, denoted as <*L*, *S*, *C*>. Each tuple specifies a kernel instantiation that performs identical computation while leveraging varying granularities of native hardware resources. This configurable format standardizes kernel scheduling by enabling precise adaptation of vector values to the architectural features of a specific GPU, thereby maximizing both on-chip and off-chip resource utilization.

Scheduling Fully-Parallel with Fusion. The Fully-Parallel mode is highly GPU-friendly. Compared to other computation patterns, its lack of cross-element dependencies offers unique flexibility for kernel inlining and fusion. This characteristic enables seamless integration with various other operations, such as fusing type conversion or dictionary lookup into RLE decoding, thereby improving

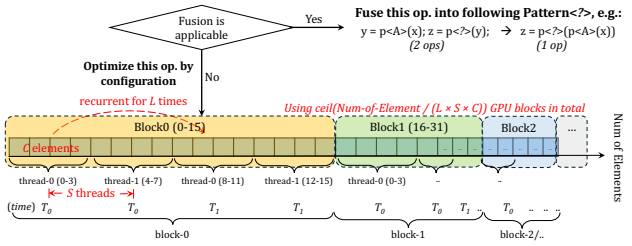

Figure 9: Fully-Parallel scheduling with <*L*,*S*,*C*>=<2,2,4>.

overall computation efficiency and reducing kernel launch overhead. In scenarios where such fusion is infeasible, decompression tasks require a separate GPU kernel. To exemplify the kernel scheduling strategy for this pattern, Figure 9 presents a representative configuration. In this instance, the variable C denotes the number of contiguous data elements jointly processed by a single GPU thread in one memory transaction, which is typically determined by the size of the data type under consideration. The parameter S specifies the number of GPU threads allocated per block, dictating the degree of intra-block parallelism. L indicates the iteration count of the main processing loop, such that each thread sequentially handles successive regions of data, offset by $S \times C$ elements, for L iterations. Collectively, the product $L \times S \times C$ effectively defines the "tile size" that represents the total workload assigned to each GPU block. This flexible configuration not only matches the hardware execution model but also allows efficient scaling and adaptation across diverse GPU architectures.

Scheduling Group-Parallel for Load Balance. GPUs are not well-suited for processing non-uniform data, which makes optimization challenging. Taking RLE decompression as an example, a straightforward strategy is to evenly distribute different RLE entries among all compute units for parallel execution. However, this approach often suffers from substantial load imbalance, since the workload of each subtask (i.e., the number of elements per RLE entry) can vary dramatically. As a result, some processors are assigned disproportionately heavy subtasks, while others complete early and remain idle, under-utilizing available resources. ZipFlow tackles this divergence by combining *multiple GPU blocks to co-process within a single group* and *a single GPU block to process across multiple groups*. As illustrated in Figure 10, the computation is organized for data arranged in four groups from left to right, each represented by a vertical bar whose height denotes the independent items within the group that can be processed in parallel. In this configuration, the entire data plane is processed by scheduling 8 GPU blocks (denoted as L), each containing 4 threads (denoted as S). For each group, 16 threads in total (denoted as C), evenly contributed by 4 GPU blocks (i.e., C/S), collaborate to process all the subtasks along the vertical axis. Along the horizontal dimension, each GPU block iteratively advances to process subsequent groups, skipping over a stride of 2 (i.e., $L \times S/C$) until all groups are exhausted. Notably, S can exceed C, implying that surplus threads from a single GPU block may concurrently process tasks spanning across adjacent S/C groups in a "lockstep" fashion, as opposed to strictly sequential execution from one group to the next. This schedule enables imbalanced data to scale along both vertical and horizontal dimensions at the cost of one-time data scan.

Scheduling Non-Parallel towards SIMT. Given that the maximum achievable parallelism in this pattern is inherently bounded by the total number of data chunks, ZipFlow issues L GPU blocks and $S \times C$ threads per block to deal with up to $L \times S \times C$ chunks of data, with each chunk managed by one thread for sequential execution. L is calculated based on the number of chunks, while S is fixed to the

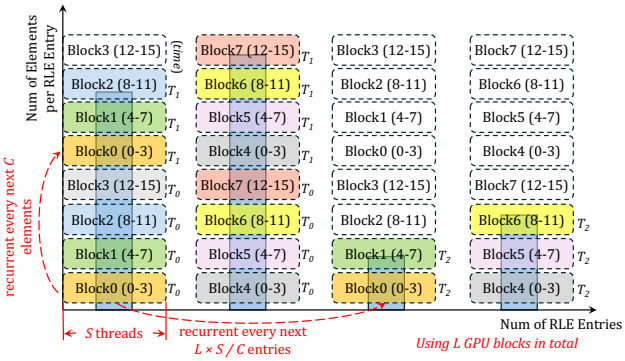

Figure 10: Group-Parallel scheduling with <*L*,*S*,*C*>=<8,4,16>.

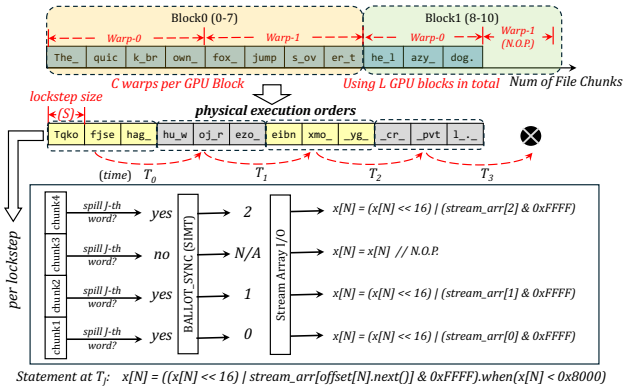

Figure 11: Non-Parallel scheduling with <*L*,*S*,*C*>=<2,4,2>.

hardware warp size (e.g., 32 for all NVIDIA GPUs and 64 for most AMD GPUs). The parameter C is architecture-dependent and enables each GPU block to handle more than S chunks, depending on available register file size per physical compute unit. As illustrated in Figure 11, which presents the physical execution order, a stack of sequential statements is repeatedly executed at different time steps without parallelism. Within each iteration, however, the processing of items from independent chunks is grouped into a SIMT unit, enabling lockstep execution. This approach enhances the consistency of I/O and cache accesses across chunks, thereby mitigating divergence and improving overall memory access efficiency.

5 EXPERIMENT

5.1 Experiment Setup

We assess the compression ratio, decompression throughput, and end-to-end query execution latency of ZipFlow using the TPC-H benchmark. By default, all experiments use a **TPC-H scale factor of 100** and are conducted on an Azure NCv4 VM with an NVIDIA A100 GPU [1] with 80GB HBM2, connected to an AMD EPYC 7V12 64-core processor via PCIe Gen4 (32 GB/s of theoretical bandwidth). To evaluate the device utilization of our approach across heterogeneous GPU architectures, we further conduct experiments on an Azure NCv5 VM with an NVIDIA H100 GPU [6] with 80GB HBM2e, an AMD MI300X GPU [7] with 192GB HBM3, and an AMD MI50 GPU [3] with 32GB HBM2. All measurements are performed with data buffers pinned to CPU memory, and we restrict execution to a single CPU node to minimize NUMA effects. We report the compression ratio as the quotient of compressed size over original size, and decompression throughput as the uncompressed data size divided by the decompression time.

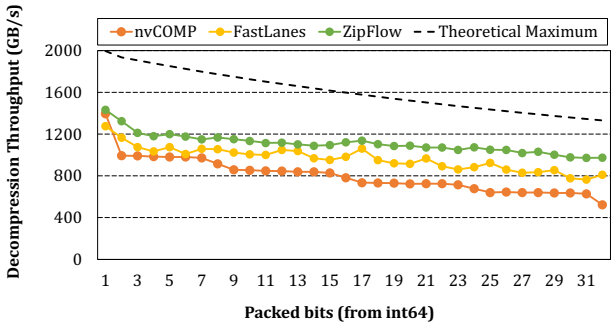

Figure 12: Throughput of bit-packing (Fully-Parallel) under varying distributions of bit-width to unpack.

For GPU (de)compression baseline, we employ nvCOMP v4.0.1.0 [13], which is widely recognized as the state-of-the-art compression library for NVIDIA GPUs. It provides a comprehensive suite of GPU-optimized algorithms, including standard methods such as LZ4, Snappy, Deflate, Zstandard (ZSTD), Asymmetric Numeral Systems (ANS), and Huffman encoding, as well as specialized techniques tailored for structured data (Cascaded), scientific data (Bitcomp), and GPU-optimized Deflate (GDeflate). Among these, the Cascaded framework in nvCOMP supports configurable nesting of predefined run-length encoding (RLE), delta encoding, and bit-packing.

For the end-to-end TPC-H query execution experiment, we combine "ZipFlow + TQP [37]" to form a complete CPU-GPU hybrid query engine fully executed in the Pytorch environment (Python 3.12 + Pytorch 2.6.0). We further compare it with state-of-the-art CPU-based DBMS, specifically, DuckDB v1.4.1 [8] and Microsoft SQL Server 2022 RTM [12], both of which were run with 64 physical threads on the same AMD processor as the GPU baseline. Note that the GPU VM price (NCv4, $3.67/hr) is comparable to a CPU-only VM (HB120rs_v2, $3.60/hr) on Azure pay-as-you-go pricing [56].

5.2 Microbenchmark

We begin with an apples-to-apples comparison against nvCOMP on the same compression patterns. For each primitive pattern, we select bit-packing, RLE, and ANS as representative instances and compare them to the corresponding algorithms in the nvCOMP library across a range of data distributions. Given the closed-source nature of nvCOMP's implementations, we calibrate algorithm settings of ZipFlow and nvCOMP in terms of same compression ratio, thereby enabling a fair evaluation of performance metrics.

5.2.1 ***Fully-Parallel Instance: Bit-packing.*** Figure 12 compares the decompression performance of the bit-packing implementations in ZipFlow, nvCOMP and FastLanes [19]. FastLanes library provides handcrafted GPU kernels for Fully-Parallel algorithms such as bit-packing and delta encoding and serves as a strong baseline for implementations optimized for previous GPU generations (T4/V100). Dashed line in Figure 12 is a theoretical maximum throughput calculated by Equation 1:

$$GpuMemBandwidth \times \frac{\text{(plain size)}}{\text{(compressed size)} + \text{(plain size)}} \quad (1)$$

To ensure a fair evaluation, we generate 4 GB synthetic input with different granularities of bit-width by uniformly sampling int64 values, with calibrated compression ratio. Taking advantage of ZipFlow's native kernel scheduling for A100, it consistently outperforms both nvCOMP and FastLanes as the compressed bit-width increases. This indicates that even for identical compression algorithms, a commodity library may not always deliver optimal

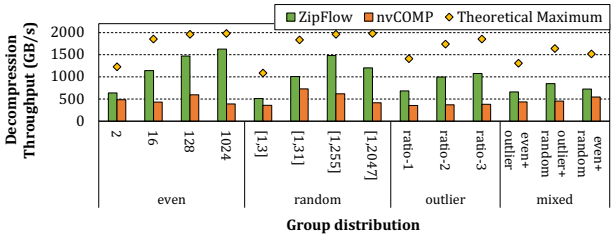

Figure 13: Throughput of RLE (Group-Parallel) under varying group size distributions: *even*, *random*, *outlier*, *mixed*.

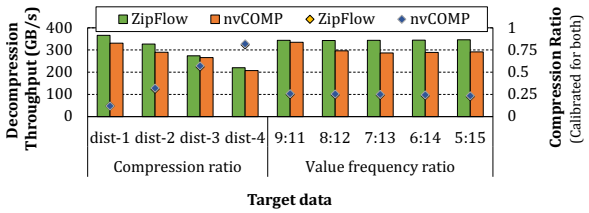

Figure 14: Throughput of ANS (Non-Parallel) under varying compression ratio (left) & frequency distributions (right).

performance depending on the input data pattern. For instance, on the L_PARTKEY column in TPC-H (compressed to a 25-bit width), our method achieves a 1.63× improvement compared to nvCOMP in decompression performance.

5.2.2 ***Group-Parallel Instance: RLE.*** We evaluate RLE decompression throughput across various group size distributions (i.e., the distribution of the *count* array in RLE), considering four representative scenarios: *even*, *random*, *outlier*, and *mixed*. In the *even-X* distribution, all groups contains identical number of elements, specifically of size X. For example, *even-2* yields an uncompressed sequence such as A-A-B-B-C-C. The *random*[L, R] distribution assigns each group a size selected uniformly at random from the range L to R. The *outlier* distribution consists predominantly of groups of size 1, interleaved with a small number of significantly larger groups (e.g., of size 1024), thereby simulating highly skewed group sizes. The *mixed* distribution is generated by concatenating any two of the aforementioned scenarios, enabling the evaluation of performance under hybrid group size patterns.

In Figure 13, we provide a comprehensive performance comparison of RLE decompression using ZipFlow and nvCOMP across 14 representative data distribution cases. The bars represent the measured throughput, while the points denote the theoretical peak achievable under memory bandwidth constraints. The results show that nvCOMP offers limited adaptability to data distribution variability, maintaining consistently suboptimal performance that remains detached from the theoretical upper bound. This limitation primarily arises from nvCOMP's fixed parallelization approach, which assigns one thread per output element irrespective of group size, resulting in memory read contention and inefficient utilization of available bandwidth [59]. In contrast, ZipFlow maintains performance much closer to the theoretical maximum across all tested distributions. This close alignment demonstrates that ZipFlow effectively leverages both data locality and GPU parallelism, leading to improved occupancy and superior bandwidth utilization.

5.2.3 ***Non-Parallel Instance: ANS.*** For the Non-Parallel pattern, we conduct the performance evaluation along two key dimensions: 1) varying the value frequency ratios to manipulate the entropy, as these significantly influence the compression efficacy of ANS

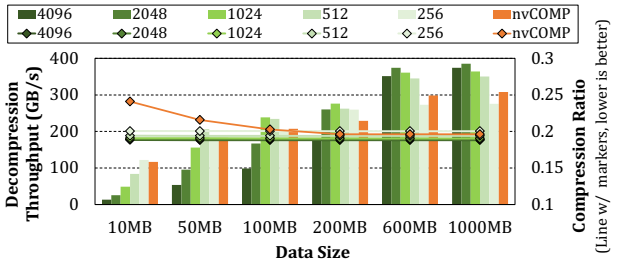

Figure 15: Throughput of ANS using varying chunk sizes (e.g., 4096 = 4KB per chunk, with nvCOMP under opaque setting).

(Figure 14); 2) analyzing the optimal chunk size across different total data volumes (Figure 15), as this parameter impacts both the compression ratio and decompression throughput. Our evaluation datasets are constructed to capture the distributional properties of the L_RETURNFLAG column from the TPC-H benchmark, where ANS compression is well-suited for encoding values containing distinct frequency distribution.

Figure 14 presents the decompression throughput from two dimensions: four bars on the left depict decompression throughput across increasing compression ratios, while the remaining four bars on the right illustrate decompression throughput as the data frequency distribution shifts from balanced to imbalanced. When the data distribution is made more skewed by increasing the frequency ratio of certain values, nvCOMP throughput drops noticeably. In contrast, ZipFlow consistently maintains stable throughput, which has been observed in Fully-Parallel and Group-Parallel, with its performance mostly governed by the compression ratio and shows minimal sensitivity to frequency variation or data skew.

Figure 15 depicts a trade-off between employing larger chunk sizes, which yield greater compression ratios, and utilizing smaller chunk sizes that facilitate more parallelism and higher decompression throughput. Given that the chunk size used by nvCOMP is not explicitly documented, we systematically sweep over a range of chunk sizes in the ZipFlow implementation to facilitate a calibrated comparison. Our findings reveal that, for smaller inputs, decreasing the chunk size leads to higher throughput by leveraging finer-grained parallelism. Conversely, for larger inputs, employing larger chunk sizes yields better scalability and improved throughput. Across all evaluated input sizes, our approach consistently identifies an optimal chunk size configuration that surpasses nvCOMP in terms of decompression throughput, while simultaneously achieving equal or better compression ratios.

In summary, the performance comparison among the three pattens (Fully-Parallel, Group-Parallel, and Non-Parallel) demonstrates the superior performance of ZipFlow. This improvement primarily arises from its fine-grained GPU scheduling adaptability. Such adaptability better aligns with both the characteristics of diverse data distributions and the architectural strengths of modern GPUs.

5.3 Compression Evaluation on TPC-H Dataset

For this evaluation, we pursue best possible performance for each data column, and design optimized compression algorithms by leveraging our framework's nesting mechanisms and flexible kernel pool customization. We focus on the three largest tables (L, O, and PS), which constitute the primary bottleneck for end-to-end query latency in TPC-H. Our custom compression algorithms integrate nesting multiple techniques from different pattern families, such as bit-packing, RLE, delta encoding, dictionary encoding, ANS, and ALP. We also employ two specialized compression variants to handle particular data patterns. First, DeltaStride (a RLE variant)

Table 1: Comparison between ZipFlow and public methods (nvCOMP [13], Parquet [4], BTRBLOCKS [45], FastLanes [18, 40]), with F.P., G.P., and N.P representing different pattern families: Fully-Parallel, Group-Parallel and Non-Parallel.

Support		nvCOMP	Parquet	BTRBLOCKS	FastLanes	ZipFlow
Nesting		Predefined	Predefined	Extensible	Extensible	Extensible
GPU support		✓ [64]	✓	✗	F.P. Only	✓
F.P.	Bit-pack & FOR	✓	✓	✓	✓	✓
	Delta encoding	✓	✓	✓	✓	✓
	Dictionary encoding	✗	✓	✓	CPU-only	✓
	ALP	✗	✗	✓	✓	✓
G.P.	RLE	✓	✓	✓	CPU-only	✓
	RLE + Delta	✓	✗	✓	CPU-only	✓
	RLE + Dictionary	✗	✓	✓	CPU-only	✓
	String-dictionary	✗	✗	✓	CPU-only	✓
	DeltaStride	✗	✗	✗	✗	✓
N.P.	LZ4	✓	✓	✗	✗	✓
	ANS	✓	✗	✗	✗	✓
	Huffman	✓	✗	✗	✗	✓
Algorithm extensibility		✗	✗	✗	✗	✓

compresses monotonically increasing integer sequences by representing them as *(start, stride, count)* triples. Second, we generate a custom dictionary for the String-dictionary compressor, the details of which are presented in Section 5.3.1. The nested compression algorithms we use are given in Table 2.

5.3.1 ***Compression Ratio.*** In this section, we evaluate the compression ratios achieved by ZipFlow in comparison to one GPU-based baseline, nvCOMP [13], and three CPU-based baselines: Parquet [4], Parquet + zstd, and BTRBLOCKS [45], irrespective of their compatibility for GPU. Table 1 summarizes the set of compression algorithms supported by each framework, while Figure 16 presents their compression ratios. For both nvCOMP and Parquet, we explore their best compression ratio attained across all supported algorithms listed in Table 1. We further provide a detailed analysis of the results for representative column clusters below.

As shown in Figure 16, columns containing date values (i.e., four columns plotted in between L_COMMITDATE and O_ORDERDATE) benefit substantially from dictionary encoding. In contrast, nvCOMP does not support this data representation and therefore yields consistently lower compression ratios relative to all alternative solutions. For numerical columns (i.e., five columns plotted from L_DISCOUNT to PS_SUPPLYCOST), our framework employs a ALP compression technique from the Fully-Parallel family, which outperforms other methods in terms of compression efficiency. For columns storing primary keys (e.g., O_ORDERKEY), we adopt a hybrid approach that combines lightweight methods: the nesting of DeltaStride and the bit-packing algorithm respectively from Group-Parallel and Fully-Parallel families. This strategy is well-suited for encoding "nearly" monotonically increasing integer sequences, resulting in substantially improved compression ratios. For string-valued columns like O_COMMENT, we introduce a tailored dictionary construction from the Group-Parallel family by tokenizing on spaces and periods. This yields a compact String-dictionary encompassing 1,878 unique words, where indices can be further bit-packed into 12 bits. When decompressing, each unique word serve as a group in Group-Parallel and expands according to the lookup dictionary. The String-dictionary approach avoids the expensive decompression overheads of LZ77 (Non-Parallel) algorithms. We further cascade our custom String-dictionary encoding with bit-packing and ANS. This approach consistently achieves higher compression ratios than both nvCOMP (zstd) and BTRBLOCKS (FSST[24]), as reflected in our experimental results.

Overall, ZipFlow broadens the range of possible nesting combinations among a wider range of custom algorithm choices, which is a key factor in discovering better compression schemes across different column types.

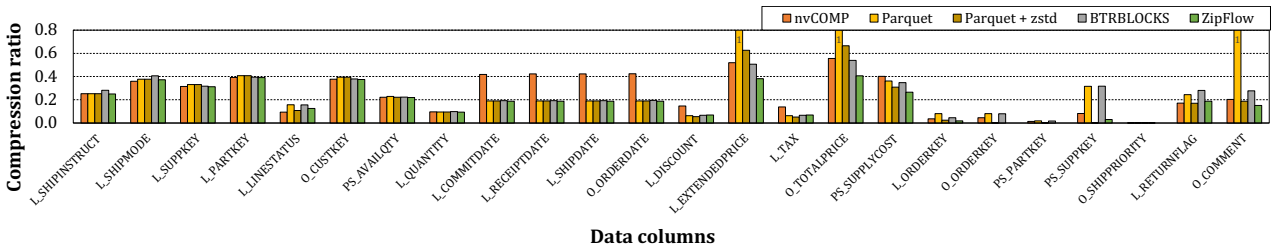

Figure 16: Compression ratio comparison between ZipFlow and four different baselines. For Parquet and nvCOMP, we iterate through all possible algorithm options and report only the best ratio. For nvCOMP's cascaded, we iterate through 72 ($6 \times 6 \times 2$) different possible nesting levels of RLE, delta encoding, and bit-packing.

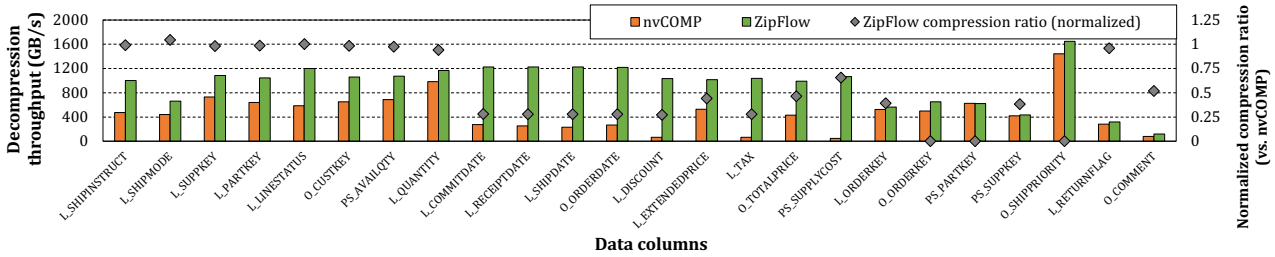

Figure 17: Decompression throughput comparison between nvCOMP and ZipFlow, where diamond markers represent ZipFlow 's compression ratios normalized relative to nvCOMP.

Table 2: Custom compression algorithm for each file. "|" divides each nested algorithm. Algorithms with two or more outputs are enclosed with "[,]".

Data Columns	Nested algorithms
L_SHIPINSTRUCT	Bit-packing
L_SHIPMODE	Bit-packing
L_SUPPKEY	Bit-packing
L_PARTKEY	Bit-packing
L_LINESTATUS	Bit-packing
O_CUSTKEY	Bit-packing
PS_AVAILQTY	Bit-packing
L_QUANTITY	Bit-packing
L_COMMITDATE	Dictionary encoding | Bit-packing
L_RECEIPTDATE	Dictionary encoding | Bit-packing
L_SHIPDATE	Dictionary encoding | Bit-packing
O_ORDERDATE	Dictionary encoding | Bit-packing
L_DISCOUNT	ALP | Bit-packing
L_EXTENDEDPRICE	ALP | Bit-packing
L_TAX	ALP | Bit-packing
O_TOTALPRICE	ALP | Bit-packing
PS_SUPPLYCOST	ALP | Bit-packing
L_ORDERKEY	RLE | [DeltaStride | [Delta encoding | RLE | [Bit-packing,Bit-packing], Bit-packing], Bit-packing]]
O_ORDERKEY	DeltaStride | [Delta encoding | RLE | [Bit-packing,Bit-packing], Bit-packing]
PS_PARTKEY	RLE | [DeltaStride , RLE]]
PS_SUPPKEY	Delta encoding | Dictionary encoding | Bit-packing | Dictionary encoding | Bit-packing
O_SHIPPRIORITY	RLE
L_RETURNFLAG	ANS
O_COMMENT	String-dictionary | Bit-packing | ANS

5.3.2 ***Decompression Throughput.*** This section evaluates GPU decompression throughput and its influence on file-level data movement. In contrast to the compression experiments, nvCOMP serves as the sole state-of-the-art baseline for comparison, since Parquet and BTRBLOCKS lack the capability to perform decompression after the compressed data are transferred to GPU. Notably, RAPIDS [64] also integrates nvCOMP as its primary decompression backend.

From the nvCOMP library, we select the algorithm that delivers the highest end-to-end performance by minimizing combined I/O and decompression latency among the functions provided. Figure 17 presents the decompression throughput of nvCOMP and ZipFlow across all data columns, and additionally illustrates the compression ratio advantage of ZipFlow over nvCOMP as indicated by the gray diamond markers. The leftmost eight data points correspond to columns based on bit-packing. For these columns, the optimal algorithm from nvCOMP is the bit-packing implementation within the Cascaded function, which yields the same compression ratio as our method. For columns ranging from `L_COMMITDATE` to `PS_SUPPLYCOST`, nvCOMP exhibits notably poor performance. Since nvCOMP lacks support for the dictionary encoding and ALP algorithms, it resorts to various alternative techniques to optimize data movement, such as Bitcomp, Gdeflate, ANS, and Cascaded. However, these complex algorithms fail to provide advantages in terms of either compression ratio or decompression throughput, as shown in Figure 17. In contrast, ZipFlow leverages a specialized yet lightweight Fully-Parallel solution, achieving on average a 21% improvement in compression ratio while delivering substantially higher decompression throughput. In summary, ZipFlow achieves an average reduction in file-level data movement overhead by a factor of 2.07× relative to nvCOMP.

5.3.3 ***Impact of Fusion on Decompression.*** We conduct an ablation study to evaluate the impact of kernel fusion on nested decompression. In Figure 18, we present the latency of fused and non-fused kernels on three nested functions: ALP + Bit-packing, Dictionary encoding + Bit-packing, and RLE + Bit-packing. Each algorithm is applied to `L_SHIPDATE`, `L_EXTENDEDPRICE`, and `L_ORDERKEY`, respectively, where all three files have the same size. The fused version of the kernel for Dictionary encoding + Bit-packing and ALP + Bit-packing executes a single kernel, while RLE + Bit-packing executes two kernels (i.e., PyTorch's cumulative sum and ZipFlow operator). The non-fused version does not fuse the bit-packing operator, resulting in two, two, and three separate kernel executions.

Empirically, kernel fusion yields a performance improvement of 2.45× for Dictionary encoding + Bit-packing and 2.08× for ALP + Bit-packing over their non-fused counterparts. This significant performance gap arises from the variances in off-chip memory access. In Equation 2, we explain the difference of required memory traffic of fused and non-fused kernels for above two cases. Memory traffic of single fused kernel consists of only the *compressed size* and *plain size* (i.e. uncompressed size), while non-fused kernel require extra

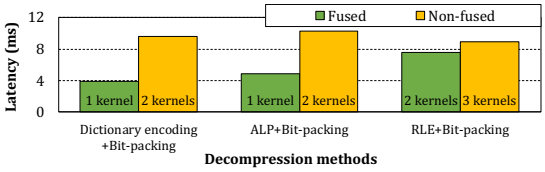

Figure 18: Kernel fusion on nested decompression kernels.

memory round trip of intermediate value. As dictionary encoding and ALP alone do not compress the data, the intermediate data cost is considered as *plain size*.

$$\frac{1 \times (\text{compressed size}) + 3 \times (\text{plain size})}{1 \times (\text{compressed size}) + 1 \times (\text{plain size})} > 2 \tag{2}$$

This result underscores the substantial penalty incurred by even a single additional memory round trip in non-fused pipelines. In contrast, we observe only a moderate speedup for RLE + Bit-packing, where kernel fusion increases performance by 1.18× relative to the non-fused baseline. This is because bit-packing is only applied to the volume `count`, which has already been compressed by RLE and only constitutes 12.5% of the original data. As a result, the fused kernel achieves a limited performance boost.

5.4 End-to-end TPC-H Query Performance

We benchmark ZipFlow against several baselines in a full-pipeline, end-to-end query execution setting using the complete TPC-H workload (all 22 queries). For each query, we transfer only the columns required by that query to the GPU. Compression is applied exclusively to the columns listed in Table 2, while other columns such as nation ID, are small enough that compression would not provide a measurable benefit. After the required data is prepared in GPU memory, we synthetically incorporate query execution latencies measured by TQP [37] on the same A100 GPU. To ensure a fair comparison of end-to-end costs and to equalize the cost of the query processing phase, we employ TQP as the query processing engine across all GPU-based baselines, while varying the compression configurations as follows: (1) an approach where no compression is applied (labeled "noCOMP") and (2) applying the nvCOMP library with the best-performing algorithms identified in Section 5.3.2. For the CPU-based baseline, we compare against state-of-the-art database systems, DuckDB and Microsoft SQL Server.

5.4.1 ***Overall Query Latency.*** Figure 19 presents the per-query execution times for all configurations. The baseline noCOMP suffers the highest latency because PCIe transfers dominate the runtime, accounting for an average 91.3% of the total execution time. Consequently, nvCOMP and ZipFlow significantly mitigate the PCIe overhead, yielding latency reductions of 1.91× and 4.08×, respectively. Across the entire benchmark, ZipFlow consistently outperforms nvCOMP, achieving an average end-to-end speedup of 2.08×.

While GPU-based solutions demonstrate significant performance advantages in most cases, CPU baselines remain competitive for several queries: DuckDB and SQL Server outperform the nvCOMP baseline respectively in 6 and 9 out of 22 queries. This occurs because the data movement overheads of nvCOMP outweigh the benefits provided by GPU acceleration. Based on a deeper data transfer optimization with ZipFlow, GPU-based query execution outperforms DuckDB in all queries except TPC-H 6, and outperforms SQL Server in all but three queries (TPC-H 6, 17, and 19). Queries such as TPC-H 6 are dominated by a 1-time scan operation and primarily constrained by memory and PCIe bandwidth. Since CPU memory bandwidth far exceeds PCIe bandwidth, these queries are more suitable for CPU execution and gain little benefit from transferring data to the GPU. Overall, ZipFlow delivers average speedups of 3.14× over DuckDB and 3.32× over SQL Server.

5.4.2 ***Latency During GPU Decompression.*** Figure 20 decomposes the query-execution latency of ZipFlow and nvCOMP into their constituent parts. To make the latency contributions of each part clearer, we plot the bar with label "C" that stands for ZipFlow without pipelining the copy and decompression phases, which clearly shows the latency saved by overlapping those operations. On average, our approach achieves a 3.26× reduction in GPU kernel latency. Among the 22 queries, we observed that those including the `L_TAX` and `L_DISCOUNT` columns (TPCH 1, 3, 5–10, 14, 15, and 19) exhibit significant differences in GPU decompression latency between our solution and nvCOMP. This gap arises because the optimal algorithm for `L_TAX` and `L_DISCOUNT` in nvCOMP is Gdeflate, which has a relatively low decompression speed of an average of 66.6GB/s. In contrast, our proposed solution employs tailored ALP decompression, achieving an average speed of 1030GB/s, effectively reducing decompression overhead.

5.4.3 ***Latency During PCIe I/O.*** Attributed to the reduced size of compressed data transferred, ZipFlow achieves a substantial reduction in PCIe I/O latency, resulting in 1.85× improvement than that achieved by nvCOMP on average. Since PCIe transfer time is highly sensitive to compression ratio, the observed improvement is primarily driven by columns compressed using dictionary encoding and the ALP algorithm, for which our framework demonstrates a significant compression ratio advantage (shown in Figure 17). To provide a closer examination, we select TPCH-7 as a representative example and present a further detailed latency breakdown of three different execution scenarios in Figure 21. Our observations reveal that, in the nvCOMP-based compression strategy for TPCH-7, `L_EXTENDEDPRICE` and `L_SHIPDATE` contribute most significantly to I/O latency. In contrast, our proposed solution employs compression methods optimized for these columns, resulting in a more balanced latency distribution across all columns. Interestingly, even though `L_ORDERKEY` and `L_SUPPKEY` have the same original size as the aforementioned columns, they exhibit a far smaller impact on end-to-end latency. This is due to their high compression ratio and decompression speed, resulting in only a small portion of the overall query latency. A similar pattern is observed in TPCH-13, where latency is dominated by the `O_COMMENT` column, which constitutes 89.9% of the total data transfer volume. From the above observations, we conclude that as compressed data movement is an effective solution, the lack of compression support for a few data patterns becomes a noticeable outlier. Specifically, when other columns of similar size are efficiently compressed and decompressed, unsupported data patterns induce disproportionate latency, making certain columns stand out as end-to-end performance bottlenecks.

5.4.4 ***Impact of PCIe/GPU Pipelining.*** We analyze the impact of the Pipelining layer by comparing ZipFlow with and without pipelining, labeled as "Z" and "C" in Figure 20, respectively. For all queries except TPCH-13, pipelining significantly reduces the exposed decompression kernel latency, resulting in an average of 10% reduction in end-to-end query latency compared to non-pipelined execution. We observe that, in our current compression method design, all columns except RLE/Delta-relevant compression are dominated by PCIe I/O latency. This allows for sufficient overlap of most GPU decompression latency, but also indicates that PCIe I/O is still the primary bottleneck in the full query pipeline. This observation highlights a key optimization opportunity for designers: by further nesting complex compression methods to achieve higher compression ratios, PCIe I/O bottlenecks can be mitigated at the

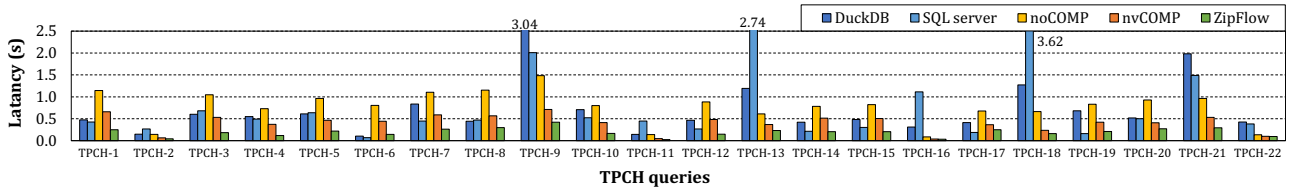

Figure 19: TPC-H query latency comparison with different CPU and GPU baselines.

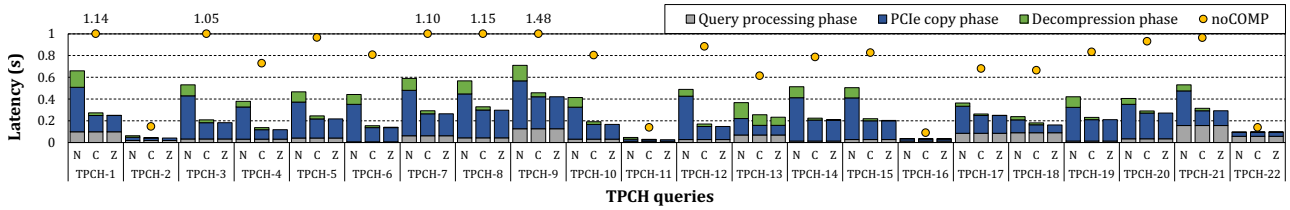

Figure 20: Breakdown of TPC-H query latency by execution phases —— "N" for nvCOMP, "C" for ZipFlow with only (de)compression applied, "Z" for full ZipFlow including PCIe pipelining, and "dots" for noComp (the baseline for ablation).

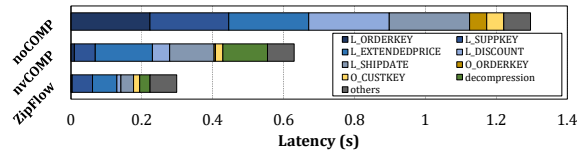

Figure 21: Breakdown of TPC-H query 7's execution latency by each input column ("others" refer to the cost spent on query processing kernels and uncompressed data transfer).

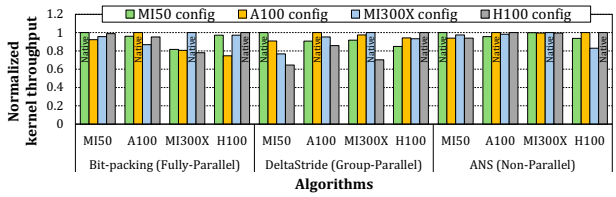

Figure 22: Kernel efficiencies under "Native Config" and "Shared Config" across 4 GPU types. All values are normalized to the respective kernel efficiency in the "Native Config".

cost of decompression throughput. It is also important to note that the tradeoff between I/O and decompression latencies is largely determined by the bandwidth disparity between PCIe and GPU memory, which represents an interesting direction for future work.

5.5 Evaluation on Heterogeneous GPUs

Although GPUs differ in terms of core counts, cache capacity, and other architectural resources, ZipFlow enables a kernel designed for one GPU to be executed on another. However, employing a non-optimal configuration typically results in sub-optimal decompression performance. Given that the optimal kernel configuration may vary across hardware platforms, we define the configuration tuned for a specific GPU as its "Native Config", and refer to configurations originally designed for other GPUs as "Shared Configs" (i.e. non-native configs). Drawing on the design principles of bounded autotune techniques ([26], [35], [71]), our system performs a one-time offline search over the design space (Section 4) to identify the "Native Config". The resulting "Native Config" ensures that online data decompression is performed at peak efficiency without the need for real-time parameter adjustment. As illustrated in Figure 22, we obtain and evaluate corresponding "Native Configs" on four representative GPUs (MI50 [3], A100 [1], H100 [6], and MI300x [7]). We also examined their cross-device portability by evaluating the performance of these "Native Configs" as "Shared Configs" on other GPU types. Across all scenarios, we observe that a GPU employing a "Shared Config" is unable to fully realize its inherent performance potential, with degradations of up to 35%.

6 PRODUCTIZATION AND LESSONS LEARNED

With the rise of GPU query engines in unified Lakehouse scenarios such as Microsoft Fabric [54], native GPU support for data lake formats like Parquet is of paramount importance. In fact, one of the key design choices in CoddSpeed [43] is to allow each host engine to directly provide data in its internal native (compressed) format, thereby avoiding costly transcoding steps on the CPU before GPU transfer. Currently, CoddSpeed's GPU engine (TQP) supports SQL Server/DW in-memory compressed format and Parquet. Nevertheless, we are actively investigating with the product team how to bring more modern formats such as Vortex [17], which allows arbitrary nesting over a diverse set of compression algorithms.

In this context, ZipFlow is the result of our early investigation into how nested compression can be integrated into TQP, and what tradeoffs and performance gains it can bring to the product. Our experience building ZipFlow and evaluating it on production-representative workloads has yielded several practical insights that we believe are valuable for practitioners working at the intersection of GPU acceleration and data compression.

Compression strategy must be co-designed with the query engine. A standalone compression library optimizes for a single metric, typically compression ratio or decompression throughput in isolation. In a query engine, however, the relevant metric is end-to-end query latency, which depends on the interplay between PCIe transfer time and GPU decompression overhead.

Hardware portability is a production requirement. Cloud environments expose a heterogeneous mix of GPUs and hardware architectures, and a production engine cannot afford to maintain hand-tuned kernels for each device. ZipFlow's compiler-based approach, in which scheduling decisions are parameterized by device geometry (e.g., warp size, shared memory capacity), allows a single algorithm specification to generate efficient kernels across vendors. This is particularly important in production environments where workload may land on different SKUs depending on availability.

Custom compression unlocks gains that general-purpose libraries miss. Our case study on TPC-H (Section 5.4) shows that data-specific algorithms can substantially improve compression ratios while maintaining high decompression throughput. Following

the optimization insight of ZipFlow, such algorithms can be efficiently compiled for rapid customization. However, these options remain absent from general-purpose libraries like nvCOMP.

Handling data updates. To support data updates without compromising decompression efficiency, ZipFlow follows the industry-standard Update-in-Delta pattern: Incoming updates are initially appended to an uncompressed Delta-Store [53] in CPU memory. During query execution, ZipFlow merges these pending updates with the original base data in GPU memory (Merge-on-Read). For long-term maintenance, an asynchronous background compaction process re-compresses the Delta-Store into compressed data during GPU idle periods. During this phase, ZipFlow can capture and store the performance metadata (*original-size*, *PCIe/GPU-latency*) to guide its pipeline scheduling with superior predictive precision.

Handling NULL values. While our primary evaluation utilizes the TPC-H dataset, which is devoid of NULL values, we acknowledge that NULL handling is a pivotal requirement in production-grade systems [78]. Columnar systems typically manage NULLs via a Validity Bitmap (or NULL-indicator bit-vector) stored alongside the data chunks. Such bitmaps are highly amenable to independent compression using schemes like RLE or Roaring Bitmaps, which leverage the sparse distribution of NULLs to achieve high efficiency.

7 LIMITATIONS AND FUTURE DIRECTIONS

Compression design on diverse data. ZipFlow relies on manually designed compression schemes tailored to TPC-H. Automating compression scheme selection is a promising direction for production deployments. A natural starting point is BTRBLOCKS [45], which constructs nested compression algorithms via iterative greedy search over sampled data chunks to maximize compression ratio. ZipFlow can extend this approach by jointly minimizing the movement time (PCIe time + GPU decompression time) during greedy selection rather than optimizing for compression ratio alone.

Mitigating GPU memory exhaustion. Memory exhaustion during decompression can be mitigated through two mechanisms: (i) preemptively checking available GPU memory before planning decompression using *original-size* embedded by *ZipFlow* in the base data, and (ii) employing pinned host memory as a "safety net" to buffer chunks during unexpected overflows. A practical approach involves integrating *ZipFlow* with a streaming query processing engine to minimize the peak GPU memory footprint, which is already employed in production systems like TQP for streamable columns. We leave the analysis of how fine-grained data ingestion further optimizes memory for future work.

Impact of NVLink interconnects. Our evaluation targets PCIe-based configurations as the most common cloud lakehouse deployment. While NVIDIA's CPU-GPU superchips offer substantially higher host-to-device bandwidth, ZipFlow remains applicable in this context, despite the need to re-calibrate the offline-profiled latencies. Therefore, evaluating behavioral differences on NVLink-enabled platforms is a compelling direction for future work.

8 RELATED WORKS

GPU accelerated query execution. Several open-source GPU-based database systems have been developed, such as Ocelot [39], BlazingSQL [2], HeavyDB [38], and RAPIDS [64]. A large body of work has extensively explored techniques for optimizing query execution on GPU hardware [29, 33, 34, 36, 46, 62, 63, 69, 70, 73]. Some studies have further investigated leveraging PyTorch implementations to accelerate GPU-based query processing [22, 37]. Building upon the significant advancements of prior research, our study addresses the next major bottleneck: data movement to GPUs.

Optimizing data movement to GPUs. Compression has been extensively employed to alleviate data movement bottlenecks to GPUs [23, 32, 49, 58, 68, 76, 79]. Prior studies [32, 49] propose planners that select optimal combinations of lightweight, column-specific algorithms based on compression ratio and decompression throughput for CPU. Several studies emphasize the importance of designing GPU-friendly decompression algorithms [19, 68]. More recent work [23] explores the use of heavyweight compression algorithms combined with on-the-fly data decompression to enable fine-grained pipelining of data transfers. Our framework further complements these efforts by providing a flexible and extensible infrastructure for developing optimized decompression kernels, thereby supporting a broader spectrum of algorithmic designs and their device scheduling. Other solutions to manage memory constraints of GPU-based data analytics include hybrid use of CPU and GPU [50, 52, 74], pipelining data movements [23, 66], minimizing redundant movement by lazy scheduling and transfer sharing [65], and utilizing all CPU-GPU PCIe links in multi-GPU scenarios [75].

Comparison of Parallelization Patterns. Parallelization patterns have been studied at various levels of the data processing stack. Spark [77] utilizes second-order functions (e.g., Map, Reduce) for distributed scalability and Heimel et al. [39] focus on hardware-oblivious patterns for portable query execution. ZipFlow introduces specialized patterns to resolve bit-level dependencies and data-dependent expansion inherent in compressed streams. By targeting the interconnect data movement layer, ZipFlow serves as a complement to these execution-focused frameworks.

9 CONCLUSION

In this paper, we propose ZipFlow: a compiler-based framework that systematically accelerates compressed data movement from CPU to GPU. This approach mitigates the significant PCIe overhead associated with transferring uncompressed data by leveraging both data-specific characteristics and GPU architectural features.

ZipFlow explicitly couples two metrics that matter in production—compression ratio and decompression throughput—and optimizes them jointly to maximize end-to-end transfer efficiency. While achieving high compression ratio largely depends on the diversity and adaptability of the available compression algorithm pool, attaining high decompression throughput relies on sophisticated kernel scheduling strategies. ZipFlow addresses these challenges by introducing three novel design patterns that facilitate flexible optimization across a wide range of compression families on the GPU. This approach enables users to efficiently compose custom algorithms for novel compression strategies and nesting, while guaranteeing high decompression throughput on heterogeneous devices. Our results demonstrate clear advantages over predefined, device-tuned solutions, and suggest that ZipFlow is a first step toward a more GPU-centric view of compression strategies in modern Lakehouse environments.

ACKNOWLEDGMENTS

This work was supported by MSRA, Microsoft Azure DB, and the MSIT(Ministry of Science, ICT), Korea under the projects supervised by the Institute of Information & Communications Technology Planning & Evaluation (IITP): DPU-Centric Datacenter Architecture for Next-Generation AI Devices (No.RS-2024-00395134), Implementation and Validation of an AI Semiconductor-Based Data Center Composable Cluster Infrastructure (No.RS-2025-02264029, 30%), (SW Starlab) High-performance Privacy-preserving Machine Learning System and System Software (No.RS-2024-00438851), Global Research Support Program in the Digital Field (No.RS-2024-00436680).

REFERENCES

[1] 2020. NVIDIA A100 TENSOR CORE GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf. Accessed: Jan 04, 2025.

[2] 2021. blazingsql. https://github.com/BlazingDB/blazingsql. Accessed: Jan 04, 2025.

[3] 2023. Instinct MI50 (32GB) specifications. https://www.amd.com/en/support/downloads/drivers.html/accelerators/instinct/instinct-mi-series/instinct-mi50-32gb.html#amd_support_product_spec. Accessed: Jan 04, 2025.

[4] 2024. Apache Parquet. https://parquet.apache.org/. Accessed: Jan 04, 2025.

[5] 2024. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: Jan 04, 2025.

[6] 2024. NVIDIA H100 Tensor Core GPU. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet?ncid=no-ncid. Accessed: Jan 04, 2025.

[7] 2025. AMD MI300x. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/data-sheets/amd-instinct-mi300x-data-sheet.pdf. Accessed: Jan 04, 2025.

[8] 2025. DuckDB - An in-process SQL OLAP data management system. https://duckdb.org/. Accessed: Jan 04, 2025.

[9] 2025. FiniteStateEntropy. https://github.com/Cyan4973/FiniteStateEntropy. Accessed: Jan 04, 2025.

[10] 2025. HIPIFY. https://github.com/ROCm/HIPIFY. Accessed: Jan 04, 2025.

[11] 2025. lz4. https://github.com/lz4/lz4. Accessed: Jan 04, 2025.

[12] 2025. Microsoft SQL server. https://www.microsoft.com/en-us/sql-server. Accessed: Jan 04, 2025.

[13] 2025. nvCOMP, High-Speed Data Compression Using NVIDIA GPUs. https://developer.nvidia.com/nvcomp. Accessed: Jan 04, 2025.

[14] 2025. NVLink NVSwitch: Fastest HPC Datacenter Platform. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed: Jan 04, 2025.

[15] 2025. snappy. https://github.com/google/snappy. Accessed: Jan 04, 2025.

[16] 2025. zstd. https://github.com/facebook/zstd. Accessed: Jan 04, 2025.

[17] 2026. Vortex: An extensible, SOTA columnar file format. https://vortex.dev/. Accessed: Jun 29, 2026.

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

[19] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing Using FastLanes Compression. In *DaMoN '24: Proceedings of the 20th International Workshop on Data Management on New Hardware*. ACM, 1–11. https://doi.org/10.1145/3662010.3663450

[20] Azim Afroozeh, Leonardo Kuffó, and Peter Boncz. 2024. ALP: Adaptive Lossless Floating-Point Compression. In *SIGMOD '24: Proceedings of the 2024 ACM SIGMOD International Conference on Management of Data*. ACM, 1–13. https://doi.org/10.1145/3626717

[21] Tim Anema, Joost Hoozemans, Zaid Al-Ars, and H. Peter Hofstee. 2025. High Throughput GPU-Accelerated FSST String Compression. In *VLDB 2025 Workshop: 16th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS'25)*. https://www.vldb.org/2025/Workshops/VLDB-Workshops-2025/ADMS/ADMS25-01.pdf.

[22] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. 2022. Share the tensor tea: how databases can leverage the machine learning ecosystem. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3598–3601. https://doi.org/10.14778/3554821.3554853

[23] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6, Article 237 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698812

[24] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661. http://www.vldb.org/pvldb/vol13/p2649-boncz.pdf

[25] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (2023), 441–454. https://doi.org/10.14778/3632093.3632107

[26] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[27] Wei Cui, Qianxi Zhang, Jesús Camacho-Rodríguez, Spyros Blanas, Brandon Haynes, Yinan Li, Peng Cheng, Ravishankar Ramamurthy, Rathijit Sen, and Matteo Interlandi. 2023. Query Processing on Gaming Consoles. In *19th International Workshop on Data Management on New Hardware (DaMoN '23), June 18–23, 2023, Seattle, WA, USA*. https://www.microsoft.com/en-us/research/publication/query-processing-on-gaming-consoles/

[28] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 72–83. https://doi.org/10.5441/002/EDBT.2017.08

[29] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. 2024. Output-sensitive Conjunctive Query Evaluation. *Proc. ACM Manag. Data* 2, 5 (2024), 220:1–220:24. https://doi.org/10.1145/3695838

[30] L. Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. https://doi.org/10.17487/RFC1951

[31] Jarek Duda, Khalid Tahboub, Neeraj J. Gadgil, and Edward J. Delp. 2015. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium (PCS)*. 65–69. https://doi.org/10.1109/PCS.2015.7170048

[32] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proc. VLDB Endow.* 3, 1 (2010), 670–680. https://doi.org/10.14778/1920841.1920927

[33] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[34] Henning Funke and Jens Teubner. 2020. Data-parallel Query Processing on Non-uniform Data. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 884–897. https://doi.org/10.14778/3380750.3380758

[35] Xiaotian Gao, Wei Cui, Lintao Zhang, and Mao Yang. 2020. OpEvo: An Evolutionary Method for Tensor Operator Optimization. *CoRR* abs/2006.05664 (2020). arXiv:2006.05664 https://arxiv.org/abs/2006.05664

[36] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[37] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proc. VLDB Endow.* 15, 11 (July 2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[38] HEAVYDB.AI. 2025. World's Fastest Analytics Database. Powered by GPUs. https://www.heavy.ai/product/heavydb. Accessed: Jan 04, 2025.

[39] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. https://doi.org/10.14778/2536360.2536370

[40] 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, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3736227.3736242

[41] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating Database with Tensor Processors. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1360–1374. https://doi.org/10.1145/3514221.3517869

[42] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[43] Matteo Interlandi, Nicolas Bruno, Brandon Haynes, Carlo Curino, Rathijit Sen, Yinan Li, Kaushik Rajan, Bailu Ding, Lukas M. Maas, Wei Cui, Kevin Gaffney, Mingsheng Hong, Brian Kroth, Sampath Rajenda, Peng Cheng, Surajit Chaudhuri, Johannes Gehrke, Raghu Ramakrishnan, Lidong Zhou, Momin Al-Ghosien, Craig Peeper, Marius Dumitru, Conor Cunningham, Kevin Bocksrocker, Prasanna Sundar, Ron Boskovic, YongChul Kwon, Marko Zivanovic, Runbin Shi, Krystian Sakowski, Denis Lamtsov, Josep Aguilar Saborit, Krish Srinivasan, Adarsh Kapil, Andrew Putnam, Anudeep Kambapu, Aparna Konduri, Aaron Landy, Aaron Moore, Aaron Pitman, Artem Oks, Ashit Gosalia, Babu Kandimalla, Blake Pelton, Bogdan Crivat, Cesar Galindo-Legaria, Chidamber Kulkarni, Jesus Camacho Rodriguez, Evgeny Babin, Hans Lehnert Marino, Igor Konev, Israel Arroyo, Luis Penaranda, Mandar Datar, Morten Borup Petersen, Nicholas Simmons, Parminder Poonian, Pravija Danda, Rob Rydberg, Esteban Calvo, Ronak Bajaj, Sai Sumanth Kammal Shetty, Sharanya Bhat, and Zach Iracheta. 2026. CoddSpeed: Hardware Accelerated Query Processing in Microsoft Fabric. In *Companion of the International Conference on Management of Data* (India) *(SIGMOD Companion '26)*. Association for Computing Machinery, New York, NY, USA, 359–372. https://doi.org/10.1145/3788853.3803077

[44] Selmer Martin Johnson. 1954. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly* 1, 1 (1954), 61–68.

[45] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26. https://doi.org/10.1145/3589263

[46] Harald Lang, Andreas Kipf, Linnea Passing, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2018. Make the Most Out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Houston, Texas) *(DaMoN '18)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/3211922.3211928

[47] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Süleyman Sirri Demirsoy, and Kai-Uwe Sattler. 2020. Faster & strong: string dictionary compression using sampling and fast vectorized decompression. *VLDB J.* 29, 6 (2020), 1263–1285. https://doi.org/10.1007/S00778-020-00620-X

[48] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *Proc. VLDB Endow.* 14, 12 (July 2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[49] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[50] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases Beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (Sept. 2025), 4518–4531. https://doi.org/10.14778/3749646.3749710

[51] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. 2024. Corra: Correlation-Aware Column Compression. *arXiv preprint arXiv:2403.17229* (2024).

[52] Haotian Liu, Bo Tang, Jiashu Zhang, Yangshen Deng, Xiao Yan, Xinying Zheng, Qiaomu Shen, Dan Zeng, Zunyao Mao, Chaozu Zhang, Zhengxin You, Zhihao Wang, Runzhe Jiang, Fang Wang, Man Lung Yiu, Huan Li, Mingji Han, Qian Li, and Zhenghai Luo. 2022. GHive: accelerating analytical query processing in apache hive via CPU-GPU heterogeneous computing. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig (Eds.). ACM, 158–172. https://doi.org/10.1145/3542929.3563503

[53] Microsoft. [n.d.]. Delta Store and Rowgroup: Microsoft SQL Server. https://learn.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver17#deltastore Accessed: Jan 04, 2026.

[54] Microsoft. 2025. Microsoft Fabric: A Unified Analytics Platform. https://learn.microsoft.com/en-us/fabric/. Accessed: 2025-10-10.

[55] Microsoft. 2026. OneLake, the OneDrive for data. https://learn.microsoft.com/en-us/fabric/onelake/onelake-overview. Accessed: 2026-02-12.

[56] Microsoft Azure. [n.d.]. Linux Virtual Machines Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. Accessed: May 01, 2026.

[57] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 283–294. https://doi.org/10.5441/002/EDBT.2014.27

[58] Hamish Nicholson, Konstantinos Chasialis, Antonio Boffa, and Anastasia Ailamaki. 2025. The Effectiveness of Compression for GPU-Accelerated Queries on Out-of-Memory Datasets. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/3736227.3736240

[59] Nikolay Sakharnykh and Rene Mueller. 2020. Software-Based Compression for Analytical Workloads. Presented at NVIDIA GTC 2020. https://developer.nvidia.com/gtc/2020/video/s21597-vid.

[60] NVIDIA. 2026. Spark RAPIDS: GPU-accelerated Apache Spark. https://nvidia.github.io/spark-rapids/. Accessed: May 01, 2026.

[61] Apache ORC. 2025. the smallest, fastest columnar storage for Hadoop workloads. https://orc.apache.org/. Accessed: Jan 04, 2025.

[62] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2019. Revisiting Hash Join on Graphics Processors: A Decade Later. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*. IEEE, 294–299. https://doi.org/10.1109/ICDEW.2019.00008

[63] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 202–214. https://doi.org/10.14778/3425879.3425890

[64] RAPIDS. 2025. GPU Accelerated Data Science. https://rapids.ai/. Accessed: Jan 04, 2025.

[65] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated Data Management Under the Test of Time. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[66] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (2020), 708–720. https://doi.org/10.14778/3436905.3436927

[67] 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 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1617–1632. https://doi.org/10.1145/3318464.3380595

[68] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1390–1403. https://doi.org/10.1145/3514221.3526132

[69] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[70] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 417–432. https://doi.org/10.1145/3035918.3064043

[71] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[72] Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. 2025. GSST: Parallel string decompression at 191 GB/s on GPU. In *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (Rotterdam, Netherlands) *(CHEOPS '25)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3719330.3721228

[73] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM Manag. Data* 3, 1, Article 39 (Feb. 2025), 27 pages. https://doi.org/10.1145/3709689

[74] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[75] Yichao Yuan, Advait Iyer, Lin Ma, and Nishil Talati. 2025. Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics. *Proc. VLDB Endow.* 18, 4 (May 2025), 1250–1263. https://doi.org/10.14778/3717755.3717780

[76] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210

[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 2.

[78] 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 2024, Santiago, Chile, 10 June 2024*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 10:1–10:10.

[79] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu