G-ALP: Rethinking Light-weight Encodings for GPUs

Sven Hepkema
CWI
Amsterdam, Netherlands
sven.hielke.hepkema@cwi.nl

Azim Afroozeh
CWI
Amsterdam, Netherlands
azim@cwi.nl

Charlotte Felius
CWI
Amsterdam, Netherlands
felius@cwi.nl

Peter Boncz
CWI
Amsterdam, Netherlands
boncz@cwi.nl

Stefan Manegold
CWI
Amsterdam, Netherlands
stefan.manegold@cwi.nl

Abstract

This paper introduces G-ALP, a GPU-optimized version of ALP, which is a recent and state-of-the-art compression scheme for floating-point. This GPU-optimization is based on two core ideas. First, all parts of the decoding process must be fully data-parallelized. In this paper, we fully data-parallelize exception patching, which typically applies to only 1% of the data. While patching has negligible performance cost on CPUs, it can become the main bottleneck on GPUs if it is not data-parallel. Second, the decoding API must minimize its register footprint, a highly scarce resource on GPUs, and hence deliver just one value-at-a-time. Our unique aim is to integrate G-ALP decoding into GPU kernels that consume data from global memory, rather than let decompression be a separate kernel. We consider these two ideas general guidelines for future GPU-optimized lightweight encodings, and a significant evolution of our new FastLanes file format, making it GPU-friendly. We extensively test G-ALP in a series of microbenchmarks and evaluate its performance on an NVIDIA V100 GPU and an NVIDIA RTX4070 Super Ti GPU, demonstrating superior performance compared to NVIDIA nvCOMP and ndzip in both decoding and filtering queries.

CCS Concepts

• **Information systems** → **Data compression**.

Keywords

Compression, Encodings, Data-Parallelism, GPU, Floating-Point, FastLanes, ALP, G-ALP

ACM Reference Format:
Sven Hepkema, Azim Afroozeh, Charlotte Felius, Peter Boncz, and Stefan Manegold. 2025. G-ALP: Rethinking Light-weight Encodings for GPUs. In *21st International Workshop on Data Management on New Hardware (DaMoN '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3736227.3736242

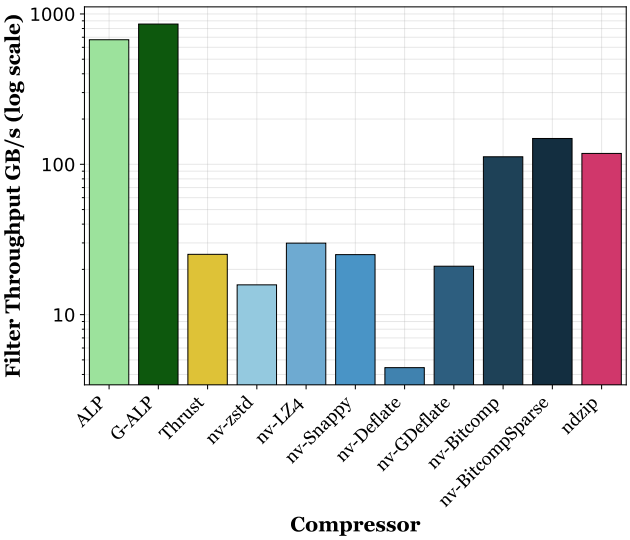


Figure 1: Throughput in GB/s of a filter-query executed with a V100 GPU on real-world data (double-precision floating-point columns from the real-world ALP datasets), stored with various compression schemes. This filter benchmark runs on compressed data, requiring decompression before the filter is executed, except for G-ALP, which can filter compressed data directly. nv-zstd achieves a similar compression ratio as G-ALP, but has 54.2x slower filter throughput. G-ALP achieves a 34x higher throughput than the baseline Thrust, which only executes the filter without any compression. This shows that data processing kernels can be accelerated by integrating decompression in them, as it reduces bandwidth needs.

1 Introduction

FastLanes is a project initiated at CWI, designed as a foundation for next-generation big data formats. With release v0.1 [1], FastLanes provides an open-source, dependency-free file format, implemented in C++. FastLanes is fully data-parallelized by utilizing the novel 1024-bit interleaved and Unified Transposed Layout [2], enabling fully data-parallel decoding even with scalar code on the CPU. It significantly outperforms the state-of-the-art, achieving an 80× speedup on the M1 processor compared to Parquet while also achieving a 40% better compression ratio.

DaMoN '25, Berlin, Germany

ACM ISBN 979-8-4007-1940-0/25/06
https://doi.org/10.1145/3736227.3736242

Furthermore, we addressed the need for a data-parallel encoding for floating-point data by incorporating our novel decoding scheme, **ALP** [4]. ALP encodes floating-point data by mapping it to the integer domain while storing a small amount of metadata to convert floating-point values to integers. This conversion consists of two multiplications and one cast operation, which can be fully data-parallelized. The resulting integers are then further compressed using the FastLanes Frame of Reference (FFOR). We observed that approximately one percent of double-precision values cannot be mapped to integers. Therefore, we separate these values, referred to as exceptions, from the main data and encode them separately using a patching mechanism [25]. This mechanism reinserts exceptions during decoding with negligible overhead in terms of both compression ratio and decoding speed, as exceptions occur very rarely. Combining ALP with FFOR and Exception patching uses the *Expression Encoding* feature of Fastlanes, in which multiple light-weight compression methods can be cascaded.

Designing a new analytical file format like FastLanes, should take into account AI workloads, and therefore **GPU** based decoding and, to a lesser extent, encoding. Processing highly compressed data on the GPU is particularly attractive, as GPUs typically have smaller RAM ("global memory") than the host CPU, meaning that storing compressed data alleviates a capacity bottleneck. Furthermore, data is transferred to the GPU over the PCIe bus, so reducing the amount of data moved through compression also helps mitigate this bottleneck. The experiences in this paper with adapting ALP to GPUs in G-ALP illustrate that GPUs can benefit strongly from small changes in compressed data-layouts. Therefore we think that there are broader lessons to be learned from this paper, and we intend to apply these in the next versions of FastLanes, also for other compression methods.

In our initial work on extending FastLanes to the GPU [3], we demonstrated that data-parallel encodings are key to utilizing the massive parallelism of the GPU. However, we observed that our API, optimized for the vectorized execution model—the most widely adopted execution model on the CPU—forces GPU kernels to materialize 32 values per thread per column, which can easily become a bottleneck, as the amount of nearby high throughput memory per thread (registers/shared memory/L1 cache) is significantly more limited on the GPU compared to the CPU. Therefore, we adjusted the FastLanes API to enable even more fine-grained decoding, allowing 16, 8, and 4 values per thread. We observed that this modification was crucial for achieving high occupancy on SSB benchmarks running on a state-of-the-art academic database, Crystal [22].

In this paper, we introduce **G-ALP**, a GPU-friendly version of ALP with two optimizations. First, we propose a novel data-parallelized layout for storing exceptions, allowing the GPU to reinsert exceptions entirely in parallel. Second, we provide a flexible API for delivering decoded values, ranging from one value at a time per thread—offering the lowest possible number of materialized values to ensure minimal local memory usage for any library using FastLanes—to 32 values at a time per thread, providing a more versatile API with different granularities.

Contributions. Our main contributions are:

- A novel data layout for storing exceptions, fully data-parallel, enabling GPU threads to reinsert exceptions in parallel.
- The design and implementation of G-ALP, a GPU-friendly version of ALP with a data-parallel layout for exceptions and a novel API, enabling one-value-per-thread processing to minimize pressure on users of G-ALP.
- Open-sourcing the implementation of G-ALP[1].
- An extensive set of microbenchmarks used to fully optimize G-ALP.
- An evaluation against nvCOMP, the state-of-the-art compression framework developed by NVIDIA, demonstrating the superior performance of G-ALP in both decoding and aggregation queries.

Outline. We begin by explaining key aspects of GPUs in Section 2. Next, we present the design of G-ALP in Section 4, followed by our evaluation results in Section 5. We then discuss related work, with an emphasis on nvCOMP, in Section 6. Finally, we conclude our work in Section 7 and outline our vision in the future work section, Section 8.

2 GPU

In this section, first the general hardware structure of NVIDIA GPUs is described. The section continues with an explanation of how instructions are issued and executed. The section concludes with how *instruction-level parallelism* (ILP) can be used to reduce the impact of instruction latencies on performance.

SIMT. An NVIDIA GPU consists of a number of *streaming multiprocessors* (SM). Each SM consists of *warps*. Warps consist of 32 *threads*. All threads in the same warp execute the same instruction; NVIDIA calls this the *single instruction, multiple threads* (SIMT) model. *CUDA*, NVIDIA's GPU programming language, enables developers to program the GPU as if they were programming a single, independent thread. However, because all threads within a warp execute the same instruction, it is more clear to think of instructions executed by the GPU as vector instructions [24].

Instruction pipelines. A SM contains a set of heterogeneous instruction pipelines. Each pipeline only executes certain classes of instructions, such as memory instructions or floating point arithmetic. Some classes of instructions can have multiple pipelines [24]. Warps themselves do not execute instruction, they issue these to the pipelines, which are shared by multiple warps. This sharing of pipelines is somewhat comparable to hyperthreading on CPUs. The SM's warps are distributed among multiple *instruction issuers*, each of which control access to a set of pipelines. Each clock cycle, the instruction issuer picks a warp to issue an instruction [7]. When the warp is not able to issue an instruction, the warp is considered *stalled*. A warp might not be able to issue an instruction due to *data hazards* or *structural hazards*, then the warp needs to wait for the result of a previously issued instruction to complete. When a warp stalls, the instruction issuer will pick another, non-stalled warp to issue an instruction [6, 7, 11].

Occupancy. SMs contain a fixed amount of resources that are shared among all warps. A kernel might be programmed in such a way that the SM needs to allocate a large amount of resources per warp. In that case the SM can disable some warps, lowering the

[1] https://github.com/cwida/FastLanesGpu-Damon2025

occupancy, the ratio of *active* warps. If the number of active warps is relatively low, there is a smaller chance that the instruction issuer can find a non-stalled warp to issue an instruction and saturate the pipelines. This can slow down the execution of kernels.

Hiding latency. Reading memory from the GPU's RAM has high latency, in the order of hundreds of cycles [7]. GPUs can bypass this latency by switching warps, executing instructions from other warps while some of the warps are waiting for the results of their memory access, this is called *latency-hiding* [14]. In some situations, the latency of a memory access can be completely hidden, if there are enough other warps that are able to issue instructions.
Instruction-level parallelism (ILP). Another way of hiding latency is by enabling warps to issue instructions more often. By increasing ILP, a kernel's instructions can contain less data hazards and control hazards. Then, warps do not have to stall as often due to these hazards. Because warps do not stall as often, the instruction issuer is more likely to be able to pick an active, non-stalled warp, and saturate the instruction pipelines. ILP can be increased by replacing branches with branchless code, by using different algorithms, or by processing multiple values in parallel. ILP only helps when latency is a bottleneck, as when arithmetic throughput or memory bandwidth is the bottleneck, the instruction pipelines are already saturated [23].

3 FastLanes File Format

In this section, we briefly explain Light-Weight Compression (LWC) methods and the internals of the FastLanes file format [2], as well as the the design of the ALP compression algorithm [4]

3.1 Light-Weight Compression Methods

Light-weight compression methods, or *encodings*, are used to reduce memory footprint. LWCs are usually more efficient in terms of decoding speed compared to most general purpose compression methods such as LZ4. LWCs exploit specific characteristics of the data stored, such as the type and the domain, as well as patterns in the data. In a data processing pipeline, the time spent decoding compressed data should ideally be negligible compared to the processing time of the actual data. It therefore is important that we decrease time spent in decoding as much as possible. For example, in **Bitpacking**, the leading zero bits are omitted to represent the data in a more compact form. This is widely used as a last encoding and thus first decoding step. Other frequently used encoding schemes are, e.g., *Dictionary Encoding*, *Run Length Encoding* (RLE), *DELTA encoding* and *Frame-Of-Reference* (FOR). **Dictionary encoding** groups identical values together into a single bucket, which is mainly effective when the cardinality within a column is low, but values are uniformly distributed (i.e., when there are many identical values). **RLE** is preferred when there are many runs of subsequent values that are identical, while **DELTA** encoding is more useful if subsequent values have small differences (e.g., an increasing ID). **FOR** encoding applies a Frame-of-Reference to each value. For example, a sequence of 1000, 1003, 1005, 1002 can have a base of 1000, which transforms the resulting values into 0, 3, 5 and 2 respectively. These FOR-encoded values are now a good candidate for bit-packing as a final encoding step, because we can represent the encoded values in 3 bits, if we separate the base.

3.2 FastLanes Internals

FastLanes is a novel file format [2] that efficiently encodes/decodes values on the CPU targeting (virtual) 1024-bit SIMD registers with 8-, 16-, 32- and 64-bits *lanes*, by providing *data parallel* encodings. The source code of FastLanes is scalar C++, which allows compilers to auto-vectorize CPU code and exploit the SIMD parallelism.

Interleaving. To be able to efficiently decode multiple values per CPU cycle, FastLanes makes use of *interleaving*, i.e., it distributes data round-robin into separate lanes. Each SIMD lane decodes one value each cycle, thanks to interleaving having stored adjacent values in distinct lanes, making bit-unpacking fully data-parallel. This allows the original value sequence to be decoded on a CPU at staggering speed (e.g., 60 values per core per cycle).

Unified Transposed Layout. Second, FastLanes proposes a *Unified Transposed Layout* to avoid data dependencies that occur in, e.g., `DELTA` decoding, where each subsequent element is dependent on its predecessor. FastLanes further maps `Run-Length Encoding` (RLE) to dictionary encoding and applies `DELTA` encoding to the indexes of the repeated values, herewith enabling to use the *Unified Transposed Layout* for faster RLE decoding. Since FastLanes is able to remove the data-dependencies, this results in an ultra-high decoding speed.

Expression Encoding. The ultra-high decoding speed leads to the third main addition: FastLanes is making use of *cascaded encodings*, also referred to as *Expression Encoding*. Cascaded encoding refers to multiple lightweight encodings applied on top of each other, that encode or decode a single value. An example is fusing `FOR` and `DELTA` encoding, which in FastLanes leads to a higher compression ratio while achieving a fast decoding speed. It has been shown that cascaded encodings achieve compression ratios comparable to Parquet and LZ4 [10], which our findings also support.

Operators. FastLanes introduces novel *operators* which encode data based on existing light-weight encodings, in such a way that it leverages vectorized processing, accelerating the decompression speed. For example, FastLanes introduces a **Fused-FOR (FFOR)** operator, which fuses bit-unpacking with FOR-decoding in a single kernel. The main advantage from fusing is that this avoids an additional `LOAD` and `STORE` instruction. This operator stores data in a FOR vector, where all values except the base are bit-packed. Thus, using `FFOR` eliminates the need for a separate bit-packing operator.

FastLanes on the GPU. In GPU data processing, Single Instruction Multiple Threads (SIMT) parallelism is used, allowing us to map each FastLanes (SIMD) lane onto a single thread on the GPU. FastLanes encodes 1024 values together in a data segment (=a vector). Within a GPU thread, FastLanes can therefore decode one or multiple (up to 32) values [3].

3.3 Adaptive Lossless Floating-point Compression (ALP)

ALP is a novel, *vectorized*, *adaptive*, *lossless* encoding for floats- and doubles [4]. The main motivation behind ALP is the observation that most floating-point data in databases have limited precision, which means that they often can be cast to integers. By casting floating points to integers, the usage of light-weight compression methods is eased – rounding errors in floating-point arithmetic make this problematic otherwise. For the values that cannot be cast to an integer, ALP designed a separate *exception patching* mechanism. ALP is written in scalar C++ code, which triggers auto-vectorization by the compiler, and operates on arrays (vectors) of 1024 values. The encoding (ALP_{enc}) and decoding (ALP_{dec}) procedures of ALP are based on the formulas (1) and (2), respectively, where e is an exponent and f refers to an inverse factor which is introduced to remove trailing 0-digits. Within a 1024 value vector, ALP aims to use the same exponent and factor for the whole vector. In addition, ALP always verifies whether the original value can be recovered. If it detects that the original double cannot be retrieved by the given formulas with the corresponding exponent and factor, it will encode this value as an exception.

$$ALP_{enc} = round(n * 10^{e} * 10^{-f}) \quad (1)$$

$$ALP_{dec} = d * 10^{f} * 10^{-e} \quad (2)$$

ALP Design. There are four key design points that ALP leverages. The first (i) is *vectorized compression*. This is achieved by decoding 1024 values at the same time within one vector, while using the same exponent and factor. As shown in Formula (1), ALP first multiplies a value by exponent e to get a large (int32 or int64) integer, after which it is reduced with a factor f to get a small number again. For each vector, the same metadata (exponent, factor and bitwidth) are stored, followed by the bit-packed values and the exceptions. Since this incurs only little control flow and values are en/decoded per vector, ALP easily auto-vectorizes. The second key design point (ii) is *fast rounding*, for which ALP designed its own procedure that is SIMD-friendly. The third point (iii) is how ALP handles *exceptions*. Since ALP uses vectorized processing, values that are exceptions are replaced by an encoded value in their original position in the vector, while exceptions itself are stored separately in a different segment. Note that it is important to keep track of the original positions of these exceptions, which therefore are stored in another different segment by ALP. The last main design point (iv) is that *ALP uses* `FFOR`, which fuses FOR- and bit-packing, to encode doubles within a 1024-value vector. Note that ALP itself does not actually encode data, but it rather *transforms* data such that it can subsequently exploit existing light-weight encoding schemes.

Why exception patching for ALP is hard on the GPU. In ALP on the CPU, a separate loop is used to reinsert all separately stored exceptions in their original positions in the vector. This loop is usually cheap because exceptions are rare, and it avoids branching, as it does not require to check for every value whether it is an exception or not. On the GPU however we aim to decode a single value-at-a-time. This significantly complicates decompression, especially on GPUs. For example, if each thread in a warp would have to check for each value whether it is an exception or not, each thread would need to scan the exception array until it reaches a position that is equal or higher to its current position. Such a branchy loop is not GPU-friendly at all, and will cause branch divergence. In practice, this will take longer than the actual decoding of the value and should thus be avoided.

4 G-ALP

G-ALP is a floating-point data compression scheme designed to optimize FastLanes ALP (referred to as CPU-ALP) for GPUs, built upon two core ideas:

All parts of decoding on the GPU must be fully data-parallelized, even negligible ones. For example, exception patching, which accounts for only one percent of the data, must also be data-parallelized because GPUs perform poorly on any sequential workload, even at such a small scale. In contrast, CPUs are designed to handle sequential workloads efficiently, which is precisely what CPU-ALP is designed for.

The decoding API should decode one value per thread. By decoding only a single value per thread for each kernel call, we minimize additional pressure on the registers of libraries using the FastLanes reader to an absolute minimum. In the ALP implementation each thread first decodes all 32 values in a lane, and after that patches all exceptions by iterating over each position and checking whether the position exists in the exception list. With single value decoding, a single value is decoded, and then immediately the decoder checks whether the position of that value occurs in the exception list and patched if needed. The resulting value is then returned to the caller of the decoding function. When the decoding function is called again, the next value in the lane is decoded.

Overview. The encoding process of G-ALP is similar to CPU-ALP, with one key difference: a data-parallel exception layout, which is explained below. In G-ALP, each set of 1024 floating-point values is considered a single encoding/decoding unit. During encoding, these 1024 floating-point values are mapped to integers, which are further compressed using `FFOR`, along with metadata specifying how to cast these integers back to doubles, which is later used during decoding. The decoding process follows FastLanes' 1024-bit ISA [2], where each lane corresponds to a separate thread. Conceptually, the decoding process on the GPU can be visualized as 32 threads, each decoding one value at a time for 32 iterations, resulting in a total of $32 \times 32 = 1024$ decoded values.

Data-Parallel Exception Layout. G-ALP stores exceptions in a fully data-parallel layout. This is implemented as an additional step at the end of ALP encoding, where exceptions are reordered into a data-parallel format. In this section we will study the layout for 32-bit floating-point data, but the specification can be trivially changed to accommodate 64-bit floating-point data as well.

The key idea behind this new layout is to provide each GPU thread, responsible for decoding a specific lane in the CPU-ALP data-parallel layout (e.g., thread 0 handling values at positions 0,

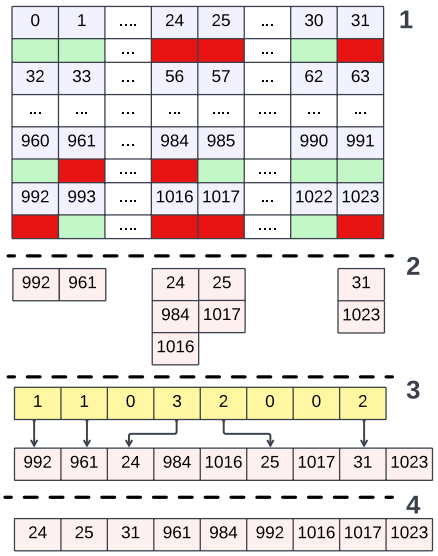


Figure 2: Example of Data-Parallel Layout for Patching. 1) A vector of 1024 values consists of two components for each value: the top box represents the position of the value within a vector, ranging from 0 to 1023, while the bottom box is color-coded (red and green) to indicate whether the value is an exception, with red denoting an exception. 2) The second part illustrates exceptions per lane, where each lane contains its own subset of exceptions. 3) The third part shows the actual storage format for exceptions. Exceptions are stored based on lane number, starting with lane 0, followed by lane 1, and so on. Yellow boxes represent metadata consisting of offsets (shown by arrows) that indicate the starting position of exceptions for each lane, as well as the number of exceptions denoted in the box. This structure allows each thread to efficiently access its corresponding exception list. 4) Finally, for comparison, we present the CPU layout of the same exception list, highlighting the structural differences between the two layouts.

32, ..., 960, 992), with direct access to all exceptions occurring in its lane in a single location. This eliminates the need for each thread to traverse the entire exception list to locate its exceptions, enabling fully parallel exception handling.

For each thread, after decoding a value, the next exception position is checked. If this position corresponds to the current value being decoded, the thread returns the exception; otherwise, it returns the decoded value.

For this data-parallel layout, we first store all exceptions for thread 0, corresponding to lane 0, which consists of positions 0, 32, 64, ..., 992 sequentially. Additionally, we store 16-bit metadata consisting of two parameters:

- **Offset** – indicating where the exceptions for lane 0 start.
- **Count** – specifying how many exceptions this lane has.

The offset and count are essential to provide each thread with direct access to its own exceptions without further computation. The offset can be as large as 1024, requiring 10 bits in the worst-case scenario when all values are exceptions. Each lane can have a maximum of 32 exceptions, requiring only 5 bits to store the count. We use 16 bits to efficiently pack these two parameters together.

This process is repeated 31 more times for threads 1 to 31. Figure 2 illustrates this layout by comparing the exception layouts of CPU-ALP and GPU-ALP, highlighting how this design enables more efficient decoding on GPUs.

Compression Overhead. G-ALP introduces a compression overhead compared to ALP, arising from the additional metadata required to make exception patching fully data-parallel. This overhead is fixed at $16 \times 32 = 512$ bits per vector (1024 values), which translates to 0.5 bits per value. Considering that ALP encodes double-precision data using 21 bits, the additional 0.5-bit overhead is negligible.

Decoding. Each thread is responsible for decoding values stored in its corresponding lane of the FastLanes layout, ranging from 0 to 31. The decoding process for a value at position X in the lane consists of the following steps:

(1) Apply the frame-of-reference method to decode the integer at position X. This involves generating the appropriate bitmask to extract the relevant bits, followed by a right shift. If the bit-packed value spans two words, an `if` condition fetches the next word and combines the bits.
(2) Cast the decoded value to floating-point using a cast instruction.
(3) Check whether the current value is an exception. If not, deliver the decoded floating-point; otherwise, return the exception. To determine if a value is an exception, compare the next exception position in the list with the position of the current value in the lane. If they match, the value is an exception, and we move to the next exception.
(4) Prefetch the next exception if the current value is an exception to ensure the data is available for the next iteration.

5 Evaluation

We conducted two sets of experiments: the first set consists of microbenchmarks to evaluate the effects of possible design choices for optimizing ALP on the GPU, and the second set benchmarks G-ALP against other compression schemes used for GPUs across three important categories: compression ratio, filter throughput, and decompression throughput.

Setup. All experiments were conducted on an AWS EC2 instance, `p3.2xlarge`, equipped with an NVIDIA V100 16GB GPU, featuring compute capability 7.0 and based on the Volta architecture, a data center-grade GPU. The code was compiled using NVCC 12.8 and `g++-12` as the host compiler. The version of nvCOMP used was 4.2.11. An older version of `g++` was chosen due to nvCOMP's lack of support for newer `g++` versions. Additionally, some experiments were repeated on a NVIDIA RTX4070 Super Ti GPU, which is a consumer-grade GPU with compute capability 8.9.

Data. To ensure a fair comparison, we selected double-precision columns from the Public BI dataset [5], as there was no single-precision data type available, and cast them to single-precision floats. We find `PUBLIC_BI` highly relevant, as it was also used in the original design of ALP. Additionally, floating-point data columns

that would be better suited for compression with run-length encoding or ALP_{rd} were excluded from the evaluation. For the double-precision benchmarks we reuse the dataset from ALP [4], excluding the few high-entropy columns that were compressed using ALPrd.

Measurements. To measure throughput, we use two different approaches: Nsight Compute Command Line Profiler 2025.1.1.0 for microbenchmarks and CUDA events for end-to-end queries (Figure 1 and 6, and Table 1). CUDA events are the recommended method for measuring multi-kernel end-to-end execution time [16] and are also used by NVIDIA to benchmark nvCOMP [17].

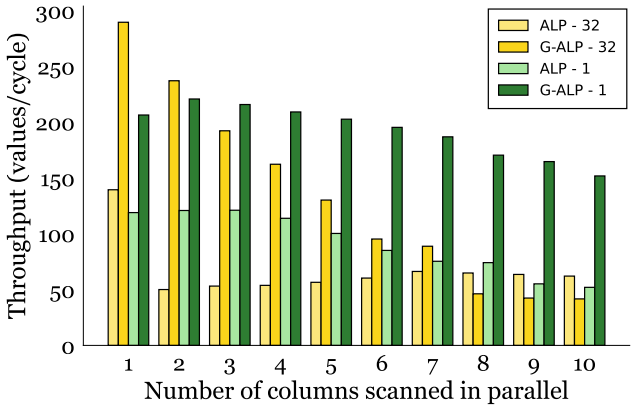
Implementation. The experiments and implementation of G-ALP are open-sourced in our repository[2]. The source code of ndzip is open source as well[3]. The nvCOMP code is not open source; however, we used its header files and binaries, which are freely available from NVIDIA's website[4].

5.1 Micro Benchmarks

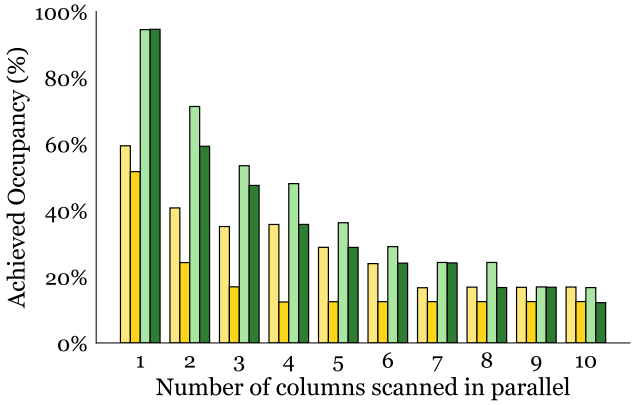
Single value decoding. We benchmarked GPU decoding of both ALP and the new G-ALP, each tested with two different APIs: 1 value-at-a-time and 32 values-at-a-time. The benchmark consists of a simple filter query on generated data while increasing the number of columns from 1 to 10, thereby increasing register pressure, as more results needs to be materialized. As can be seen in Figure 3a, our new API maintains throughput close to that of a single-column scan, even as the number of columns increases.

Furthermore, we examined achieved occupancy, as shown in Figure 3b. When scanning multiple columns, occupancy starts to decline, indicating that there are fewer warps concurrently active. This limits the ability of the SM to hide the latencies of instructions, as the SM is less likely to be able to execute an instruction from another warp. For a high number of columns, the kernel's performance depends on how efficiently the algorithm utilizes the remaining active warps, relying on instruction-level parallelism rather than occupancy to hide instruction latencies. With the new one-value-at-a-time API implementation, there is enough instruction-level parallelism to hide instruction latencies even at very low occupancy levels while retaining high decoding throughput.

G-ALP Optimization Performance. We conducted another microbenchmark to analyze the difference between the ALP implementation that more closely resembles the CPU implementation, and the GPU optimized implementation of G-ALP. In this benchmark, a filter query is repeated on generated data, where the number of exceptions per vector is varied, as shown in Figure 4. From the figure it is clear that G-ALP has much higher decoding throughput than ALP. Additionally, we study the effect of a micro-optimization where the next exception that a thread will decode is buffered in advance. This optimization increases instruction-level parallelism as reading from the buffer in the registers has lower instruction latency than reading from global memory. With this micro-optimization, G-ALP can reach up to 25% higher decoding throughput for some exception counts per stored vector of 1024 values. All implementations suffer



(a) Throughput per decoding approach



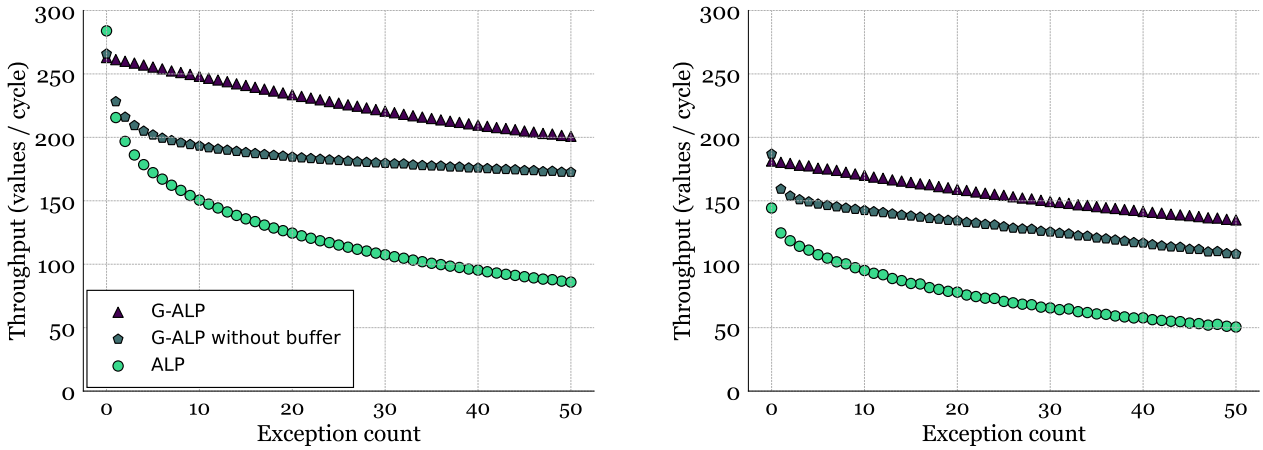
(b) Occupancy per decoding approach

Figure 3: Throughput of G-ALP executing a filter query on a V100 GPU on single-precision floating-point data with 1% exceptions, using different APIs with varying numbers of columns (1 to 10). With more columns to decode, register pressure increases, and occupancy drops for all variants. G-ALP 1 value-at-a-time is the only to maintain good instruction throughput under low occupancy, by achieving high instruction parallelism thanks to the simplicity of its kernel. G-ALP decoding 32 values at-a-time initially profits from an efficient templated bit-unpacking routine, but with more columns to decode, this routine suffers under low occupancy. Both ALP variants (1 and 32 at-a-time) need an inefficient branchy loop for exception patching, that degrades their efficiency.

under higher exception counts, but a normal exception count is 10, corresponding to a 1% exception rate.

Hardware Comparison. In Figure 5, the filter benchmark is repeated with a 1% exception rate on the RTX4070 GPU. This figure shows that consumer grade GPUs perform relatively better on decoding `float` columns than `double` columns. The reverse is true for datacenter GPUs. This difference is due to the different balance in arithmetic throughput for single-precision floating-point instructions and double-precision floating-point instructions for

[2]https://github.com/cwida/FastLanesGpu-Damon2025
[3]https://github.com/celerity/ndzip
[4]https://developer.nvidia.com/nvcomp



(a) Single-precision floating point

(b) Double-precision floating point

Figure 4: Microbenchmarks with generated data showing the impact of an increasing amount of exceptions per vector on the decoding throughput on a V100 GPU. ALP is more heavily impacted by exceptions, degrading fast in decoding throughput when a vector contains more exceptions. Double-precision data has lower decoding throughput in terms of values/cycle, but each decoded value has twice as many bytes, resulting in a higher decoding throughput when calculated as GB/s.

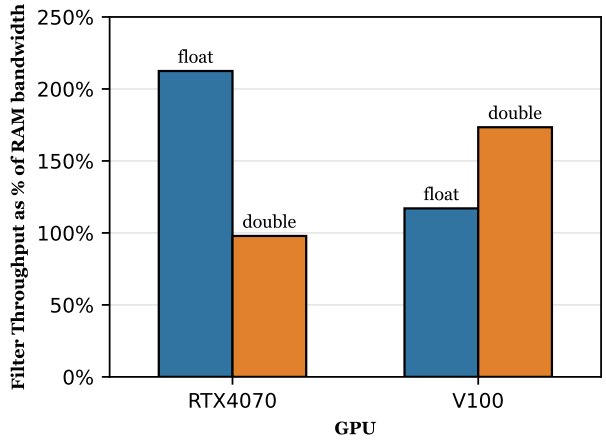


Figure 5: Filter benchmark of G-ALP on generated data with 1% exceptions per vector, for RTX 4070 and V100. The RTX 4070 is a consumer GPU has very little double-precision floating-point instruction arithmetic throughput in comparison to the RAM bandwidth, and relatively much single-precision floating-point instruction arithmetic throughput. The V100 is a datacenter GPU and has relatively more double-precision floating-point instruction throughput than the RTX4070. The V100 has higher RAM bandwidth than the RTX 4070, but not proportionally more single-precision floating-point instruction throughput to match the higher RAM bandwidth, resulting in relatively low `float` decoding throughput.

consumer-grade and datacenter-grade GPUs [12]. The figure shows that for both GPUs it can be beneficial for memory bound kernels to load data in compressed form, as the decoding overhead is less than the speedup attained by being able to load less data from RAM, as indicated by the higher than 100% filter throughput.

5.2 End-to-End Benchmarks

We include the original ALP, where we simply map the FastLanes 1024 ISA to CUDA, to measure the extent to which our two optimizations: 1) data-parallel exception handling, and 2) one-value-at-a-time decoding API, accelerate the implementation ALP. Additionally, we compare against the encodings supported by the nvCOMP framework as the current state-of-the-art in practice and use Thrust as a baseline to evaluate the performance of G-ALP against Thrust when there is no compression, highlighting the benefits of loading compressed data on GPUs. The last compressor we compare is ndzip [9], a compressor focused on HPC applications where high compression throughput and high decompression throughput are important for fast data transfers between compute nodes.

To understand how different compressors perform in comparison to each other, we measure their throughput under two queries:

(1) **Full decompression** – measuring the absolute decoding time, where data is fully written to global memory, simulating cases where G-ALP is forced to decode data completely.
(2) **Filter** – measuring the end-to-end time between decompressing data, and evaluating a query on the decompressed data. For Thrust, ALP and G-ALP no separate decompression kernel is required, as Thrust does not use compression, whereas ALP and G-ALP can load compressed data directly. This simulates the performance of G-ALP in a *tile-based execution model* [22], where our API can be used to process data immediately after decoding instead of writing it to global memory. The filter evaluates whether a certain value occurs in a column. A variation of the traditional filter is

COMPRESSOR	FLOAT					DOUBLE				
	CR	RTX4070		V100		CR	RTX4070		V100	
		Filter (GB/s)	Decompress (GB/s)	Filter (GB/s)	Decompress (GB/s)		Filter (GB/s)	Decompress (GB/s)	Filter (GB/s)	Decompress (GB/s)
ALP	1.44	241.4	235.9	138.2	124.0	**3.29**	459.0	276.9	673.7	280.2
G-ALP	1.41	**568.5**	**463.5**	**293.5**	**210.6**	3.25	**504.3**	**321.3**	**856.8**	**324.7**
Thrust	1.00	30.3	-	14.2	-	1.00	49.0	-	25.2	-
nv-zstd	**1.74**	6.3	6.4	2.3	2.4	3.11	25.8	28.8	15.8	16.5
nv-LZ4	1.27	8.3	8.3	4.3	4.4	2.27	44.2	52.3	29.9	32.4
nv-Snappy	1.37	11.8	12.0	5.7	5.8	2.25	41.8	48.6	25.1	26.6
nv-Deflate	1.54	3.3	3.4	2.0	2.0	1.53	5.5	5.6	4.4	4.5
nv-GDeflate	1.53	12.2	12.3	6.6	6.7	1.52	30.5	33.1	21.0	21.9
nv-Bitcomp	1.11	84.6	91.3	42.3	45.3	1.27	123.4	184.8	112.2	144.4
nv-BitcompSparse	1.12	108.2	119.1	53.9	59.9	1.18	136.5	207.3	148.5	209.9
ndzip	1.10	148.0	168.7	92.6	112.2	1.36	131.9	202.5	118.1	155.6

Table 1: Throughput per GPU and datatype for the filter benchmark and the decompression into RAM benchmark. Compression ratio (CR) is constant for both benchmarks and GPUs. The `float` datasets are from the real-world `PUBLIC_BI` dataset, the `double` datasets are from the real world ALP dataset. Because the `float` dataset was cast from double-precision columns, the exception rate and bits per value are higher than normal, resulting in a lower compression ratio. Thrust does not use compression and is used as a baseline, therefore Thrust is not measured in the decompression into RAM benchmark. The filter kernel for all other compressors is implemented similarly to the G-ALP filter kernel, which is faster than a normal Thrust filter kernel. G-ALP achieves the highest throughput in all benchmarks, and has only a slightly lower compression ratio than ALP.

performed, where not the row numbers are returned but simply a boolean answer on whether the value occurs in the column. We choose this variant as it requires no synchronization between threads, and also requires little write bandwidth. This in turn allows us to isolate the performance of how fast kernels can load and decode data.

The results are shown in Table 1. As can be seen, G-ALP outperforms all competitors by a significant margin in both filter and decompression throughput, while achieving a reasonable compression ratio for the `float` dataset and the highest compression ratio for the `double` dataset. G-ALP especially has a large advantage in the filter benchmarks, as with G-ALP there is no need to launch a separate decompression kernel, saving a round-trip of non-compressed data to and from RAM.

Figure 6 shows that G-ALP achieves the highest data decompression throughput, while achieving a reasonable compression ratio for the `float` dataset and the highest compression ratio for the `double` dataset. Nv-zstd achieves high compression ratios as well, while nv-Bitcomp and nv-BitcompSparse achieve a high decompression throughput, but achieve very little compression. Ndzip achieves high decompression, but relatively little compression on these datasets. G-ALP is designed for the data in the datasets, where the floating-point data can be interpreted as a decimal number with limited precision and can be casted to an int. For this kind of data Figure 6 shows that G-ALP is a very strong option as a compressor.

6 Related Work

We consider the NVIDIA nvCOMP compression framework [13, 19–21] to be the most relevant work related to G-ALP, as it is widely used in practice, with a total of 608,509 downloads as of the time of writing this paper [18]. nvCOMP supports three data types: integers, strings, and floats (16-bit). Its encoding pool for floating-point data consists of several heavyweight compression schemes such as LZ4, Snappy, ZSTD, and Deflate, as well as GPU-optimized formats like Bitcomp (proprietary and closed-source) and GDeflate.

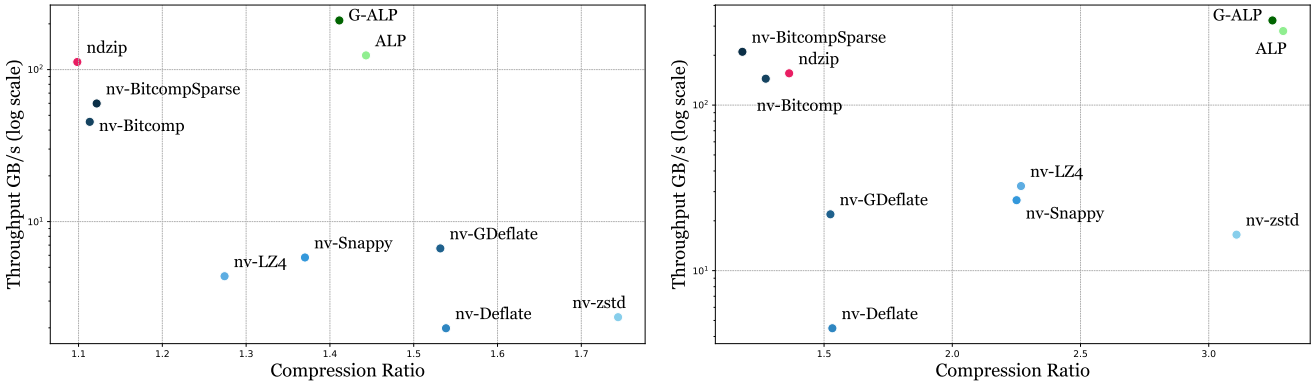
GDeflate is a GPU-friendly variant of Deflate that introduces interleaved Huffman coding, where codes are permuted into 32 partitions, though its details are vaguely explained [15]. This enables intra-threadblock parallelism, allowing GPU threads within a block to decode different partitions simultaneously, similar to interleaved bit-packing in FastLanes, where data is distributed across 32 lanes.

For LZ4, nvCOMP enhances its GPU-friendliness by breaking datasets into blocks and compressing/decompressing each block using a thread block [20]. Within each thread block, only a single warp is used to ensure efficient coordination among threads via warp-level primitives.

Below, we compare nvCOMP to G-ALP:

Schema Selection. While providing a set of compression schemes, nvCOMP does not automatically select the best scheme, leaving this decision to the user [13]. In contrast, the FastLanes file format automatically chooses the most suitable compression scheme, with G-ALP being selected only if the data is decimal-like [4].

API. nvCOMP provides two different APIs: a Low-Level API and a High-Level API [13]. The Low-Level API allows users to define the chunk size, enabling data to be compressed in smaller chunks, where each compressed chunk can later be decompressed in parallel using different thread blocks, with one thread block assigned to each compressed chunk. This approach sacrifices compression ratio in favor of increased parallelism. The High-Level API abstracts the



(a) Single-precision floating point datasets

(b) Double-precision floating point datasets

Figure 6: Full data decompression into RAM throughput per encoding scheme, for both the `float` and `double` datasets. The compression ratio and decompression throughput for all columns in the dataset are averaged per encoding scheme. The G-ALP encoding schemes achieve the highest single-precision floating point data decompression throughput, but a slightly lower compression ratio due to relatively many exceptions in the `float` dataset columns. The G-ALP encoding schemes achieve both the highest compression ratio as well as the highest decompression throughput of all encoding schemes for the `double` dataset. This shows the potential of G-ALP as a compression scheme for floating-point data on GPUs.

chunk size selection from the user by automatically determining the optimal chunk size. It then compresses the data as a whole, adding a header at the start of the compressed data that contains information about the chosen nvCOMP compression scheme.

In contrast, G-ALP and FastLanes do not require additional decoding configurations to be set by the user, making them easier to use. We provide similar support to the High-Level API, allowing the entire dataset to be decoded and materialized in global memory. Additionally, we introduce an even more fine-grained Low-Level API capable of delivering decoded data that fits into registers, the fastest form of memory on the GPU.

ndzip. ndzip is a block-based compression scheme that is designed and tested for scientific data for high performance computing workloads, to bypass data transfer bottlenecks by focusing on high compression throughput and high decompression throughput, while retaining a reasonable compression ratio [8, 9]. ndzip is also implemented for CPUs, and performs well in comparison to compressors with similar design goals in the HPC space. ndzip can compress multidimensional data, and makes no assumptions about the kind of data. Instead, it interprets floating-point data as integers and decorrelates the data using a data transformation that results in data that is easier to compress using bitpacking.

7 Conclusion

When comparing general-purpose compression schemes like zstd and LZ4 to light-weight encodings in data processing workloads on CPUs, the former already suffer from their block-based nature—often larger than the CPU cache for decompression—and their lack of data parallelism, preventing SIMDized decoding. On GPUs, which have less on-chip memory and a more rigid execution model, these problems become even bigger. Therefore, we posit that GPU-friendly data compression should be based on data-parallel light-weight encodings. By introducing G-ALP, the GPU-optimized version of the recently proposed ALP floating-point encoding ALP, we therefore also make a step towards more GPU-friendly compressed data file formats. We plan on not only integrating G-ALP itself into our FastLanes format, but also applying the lessons learned (e.g., data-parallel exception layouts) to its other encoding schemes.

Additionally, we have shown that by creating a GPU decoding API that is maximally fine-grained: a single value at-a-time, we can *integrate* decompressing data delivery into GPU kernels with minimal pressure on register and shared memory usage. This should be contrasted with the currently dominant approach in GPUs to see decompression as a separate processing step or kernel. We think that the ability to directly read compressed data can be highly beneficial for GPU-based ML and data workloads, which can easily be bottlenecked by memory bandwidth.

8 Future work

FastLanes GPU Reader. G-ALP is a step forward toward developing a GPU reader for the FastLanes file format. Through optimizing G-ALP, we learned that even optimizing a single scheme requires significant effort, indicating that optimizing the entire file format for the GPU would be highly demanding. Therefore, we leave the remaining work, such as supporting additional data types (e.g., strings and nested data types) and integrating different schemes, for future work.

GPU Diversity. We conducted all experiments on two NVIDIA GPUs, as detailed in Section 5. While those GPUs have similar capabilities to the most commonly used GPUs in the cloud, we plan to extend our benchmarking to different types of GPUs to gain a broader perspective on heterogeneous GPU architectures. This will enable us to design a more robust file format that performs

efficiently across all GPUs without significant performance cliffs on any specific hardware.

New Benchmarks. To compare G-ALP with other compression schemes, we benchmarked only scan throughput when the data is fully materialized in GPU RAM and aggregation over a single column to evaluate the effect of the one-value-at-a-time API. While these benchmarks provide insight into the performance of G-ALP compared to other schemes, they are far from comprehensive. However, the lack of a standard benchmark or workload for GPUs—similar to TPC-H for CPUs—makes this particularly challenging. We envision the creation of a new set of queries, explicitly collected from real-world GPU use cases, to better design and understand GPU file formats.

References

[1] Azim Afroozeh. 2025. FastLanes v0.1. https://github.com/cwida/FastLanes/tree/release_v0.1 Accessed: 2025-03-07.

[2] 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. doi:10.14778/3598581.3598587

[3] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. doi:10.1145/3662010.3663450

[4] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4, Article 230 (Dec. 2023), 26 pages. doi:10.1145/3626717

[5] CWI DA. 2025. Public BI benchmark. https://github.com/cwida/public_bi_benchmark

[6] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. arXiv:1903.07486 [cs.DC] https://arxiv.org/abs/1903.07486

[7] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. arXiv:1804.06826 [cs.DC] https://arxiv.org/abs/1804.06826

[8] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data. 103–112. doi:10.1109/DCC50243.2021.00018

[9] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip-gpu: efficient lossless compression of scientific floating-point data on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 93, 14 pages. doi:10.1145/3458817.3476224

[10] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. Btrblocks: Efficient columnar compression for data lakes. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[11] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture. arXiv:2402.13499 [cs.AR] https://arxiv.org/abs/2402.13499

[12] NVIDIA. 2025. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions NVIDIA Documentation, arithmetic instruction throughput per compute capability..

[13] NVIDIA. 2025. Accelerating Lossless GPU Compression with New Flexible Interfaces in NVIDIA nvcomp. https://developer.nvidia.com/blog/accelerating-lossless-gpu-compression-with-new-flexible-interfaces-in-nvidia-nvcomp/ NVIDIA Developer Blog.

[14] NVIDIA. 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#multiprocessor-level NVIDIA Documentation.

[15] NVIDIA. 2025. GDeflate: GPU-Optimized Lossless Compression. https://docs.nvidia.com/cuda/nvcomp/gdeflate.html NVIDIA Documentation.

[16] NVIDIA. 2025. How to Implement Performance Metrics in CUDA C/C++. https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/ NVIDIA Documentation.

[17] NVIDIA. 2025. nvCOMP Benchmarks. https://github.com/NVIDIA/nvcomp/tree/main/benchmarks

[18] NVIDIA. 2025. nvcomp: GPU-Accelerated Compression Library. https://anaconda.org/conda-forge/nvcomp Available on conda-forge.

[19] NVIDIA. 2025. nvcomp v2.0.0 Now Available with New Compressors. https://developer.nvidia.com/blog/nvcomp-v2-0-0-now-available-with-new-compressors/ NVIDIA Developer Blog.

[20] NVIDIA. 2025. Optimizing Data Transfer Using Lossless Compression with nvcomp. https://developer.nvidia.com/blog/optimizing-data-transfer-using-lossless-compression-with-nvcomp/ NVIDIA Developer Blog.

[21] NVIDIA. 2025. Using Fully Redesigned Batch API and Performance Optimizations in nvcomp v2.1.0. https://developer.nvidia.com/blog/using-fully-redesigned-batch-api-and-performance-optimizations-in-nvcomp-v2-1-0/ NVIDIA Developer Blog.

[22] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. 1617–1632. doi:10.1145/3318464.3380595

[23] Vasily Volkov. 2016. *Understanding Latency Hiding on GPUs*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[24] Vasily Volkov and James W Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–11.

[25] M. Zukowski, S. Heman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*. 59–59. doi:10.1109/ICDE.2006.150