Universiteit Leiden

# Master Computer Science

G-ALP: Rethinking GPU Decompression of Light-Weight Encodings


Name: Sven Hielke Hepkema
Student ID: 2454556


Date: 12/05/2025

Specialisation: Advanced Computing and Systems


1st supervisor: Prof. Dr. Stefan Manegold
2nd supervisor: Prof. Dr. Peter Boncz
Daily supervisor: Azim Afroozeh


Master's Thesis in Computer Science


Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

# Contents

**Abstract**

This thesis implements G-ALP, a data-parallel compression scheme for floating-point data that is suitable for decoding on GPUs. G-ALP expands upon ALP, which is part of the FastLanes file format encodings. The work in this thesis develops G-ALP by revisiting previous work on decoding FastLanes on GPUs, and by creating a new data-parallel exception layout. G-ALP is then compared with nvCOMP, a library of common compression schemes that were implemented for the GPU by NVIDIA. These comparisons are done on real-world datasets, for single-precision as well as double-precision floating-point data. G-ALP has one order of magnitude higher decompression throughput than other compressors, while achieving a compression ratio equal to that of the best competing compressor, zstd. Decoding throughput differences were even higher when data was not compressed into RAM but decoded directly within a data processing kernel. These results show that G-ALP is a viable encoding for compressing floating-point data on the GPU. Using G-ALP, floating-point data can be compressed, resulting in higher data transfer throughput from the CPU to the GPU, and can store more data in the GPU's RAM. With the fine-grained single value decoding API, the compressed data does not have to be decompressed into RAM, but can be read directly by a data processing kernel.

# Chapter 1

# Introduction

This thesis creates a decoder for decompressing floating-point data on *Graphics Processing Units* (GPUs). Compression has two benefits. The first benefit is that it is possible to store the same amount of information using less space. The second benefit is that it is then faster to transfer this information, as less physical data needs to be copied to transfer the same amount of information. Compression is interesting for GPU workloads, as information needs to be transferred from the host's RAM to the GPU using a PCIe link with relatively low throughput, and can also store more information in the GPU's RAM.

There are multiple possible scenarios for decompressing data before processing it on the GPU. In Figure 1.1 the most simple scenario is shown. The compressed data is decompressed on the CPU, before it is transferred to the GPU and processed there. The benefit of this approach is that it is simple, and can use common CPU decompressors. However, the GPU part of the workload does not benefit from compression, as the data is transferred in compressed form to the GPU, and stored in decompressed form on the GPU.

The second scenario is depicted in Figure 1.2, showing an alternative approach where compressed data is transferred to the GPU, and then decompressed into the GPU's RAM. This approach partially realizes the possible benefits of compression for GPU workloads. The data can be transferred in compressed form, however, the data still needs to be decompressed into the GPU's RAM, requiring a large allocation of memory.

The third scenario in Figure 1.3 bypasses the decompression step completely. By using an API that gives GPU kernels fine-grained, high-throughput access to the data in compressed form, the data can be left compressed in RAM. This approach has the advantage of both benefits of compression. Data is never transferred in decompressed form, saving memory read throughput, and no memory is allocated for storing the decompressed data, saving RAM.

There are many GPU workloads that could benefit from compression. Any form of data processing where memory throughput is a bottleneck or RAM storage is a limit

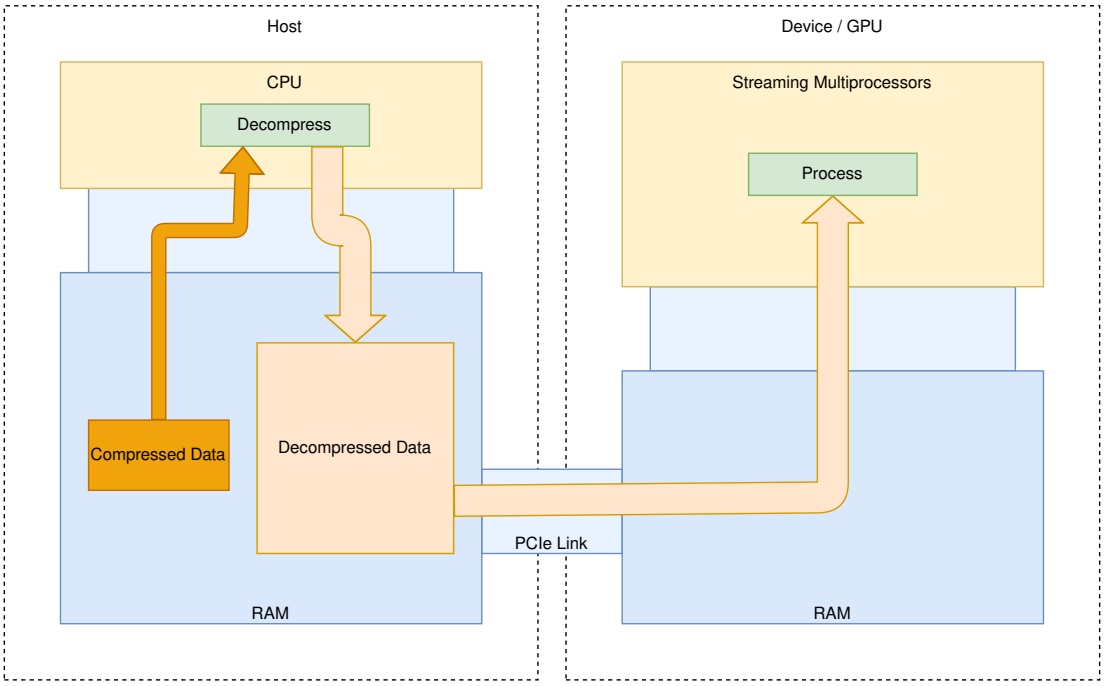


Figure 1.1: First scenario of GPU processing of compressed data. The compressed data is first decompressed on the CPU, before being transferred in decompressed form over the relatively slow PCIe link. Decompressing the data before transferring the data over the PCIe link is slow, as the throughput of the PCIe link is limited.

could benefit. Such workloads could be AI training or inference workloads, data processing kernels or GPU databases. The second scenario in Figure 1.2 requires RAM to be allocated to store the fully decompressed data, but can be convenient in some cases, as kernels that process the data do not have to be changed and can simply load the data in non-compressed form. The third scenario in Figure 1.3 saves both memory transfer throughput and RAM, but requires kernels to be changed to use the fine-grained decompression API.

FastLanes[2] is a file format designed for big data *online analytical processing* (OLAP) workloads. In OLAP workloads, data is recorded and written once, and then read many times after that to analyze the data. With the FastLanes file format data can be compressed at ratios equal to those obtained by zstd[2, 11], and decompressed at high throughput using SIMD instructions. The central FastLanes bit-interleaved encoding can be efficiently decoded on GPUs, even speeding up certain memory-throughput-bound queries[4]. FastLanes has multiple encodings for different types of data. In previous work[4] one of the FastLanes encodings for integers was implemented for GPUs. FastLanes also has encodings for floating point data, one of which is ALP[5]. No FastLanes encodings for floating-point data were implemented for GPUs yet. Therefore, this thesis implements a GPU decoder that efficiently decodes ALP encoded data with an unobtrusive API that offers fine-grained access to read compressed data.

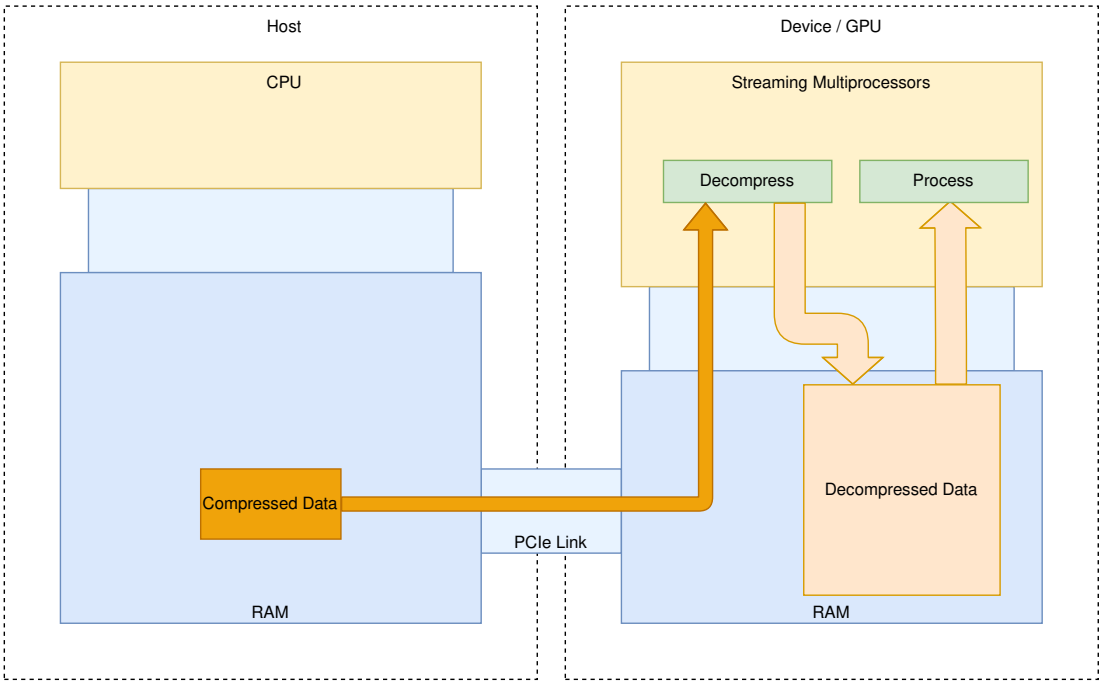


Figure 1.2: Second scenario of GPU processing of compressed data. The data is transferred in compressed form over the PCIe link, then the data is decompressed on the GPU. The decompressed data is processed. This scenario is faster than the previous scenario, as the data is transferred over the PCIe link in compressed form. However, this scenario still requires a decompression kernel to be launched, and a large allocation of memory to decompress the data into.

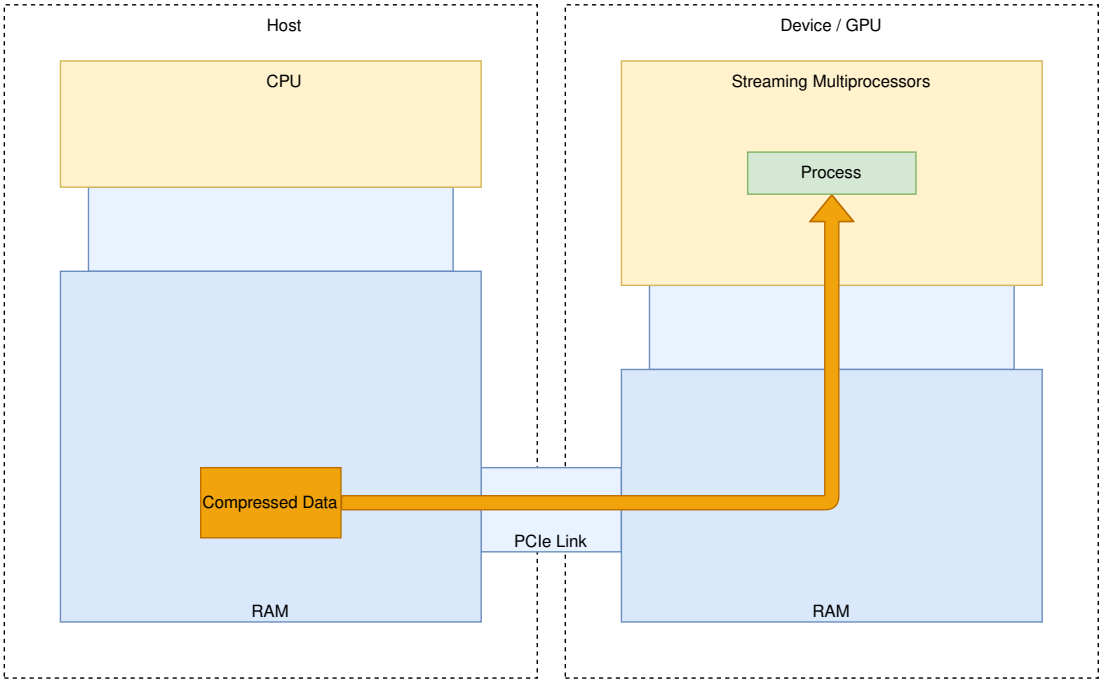


Figure 1.3: Third scenario of GPU processing of compressed data. The data is transferred in compressed form over the PCIe link, and never decompressed into RAM. The processing kernel can use an unobtrusive API to access the data in compressed form, completely bypassing the decompression step. This approach does not have to allocate memory for the decompressed data, and can also load the data into the streaming multiprocessors in compressed form, saving memory read throughput.

Creating a decoder for decompressing floating-point data on GPUs consists of three parts. In previous work, one of the FastLanes encodings was successfully decoded on GPUs[4]. However, the decoding throughput of this decoder degraded when multiple columns were decoded concurrently. (1) Therefore the first part to implement is a new decoder that uses a new decoding approach with an API that should offer consistent high decoding throughput with a single value granularity, no matter how many columns are decoded simultaneously. To decode floating-point data on GPUs, an ALP GPU decoder should be implemented. (2) The second part of this thesis implements a decoding technique for high throughput *exception patching* for GPUs. Exception patching is an encoding technique to handle values that are outliers which are hard to compress separately from the majority of the values. ALP uses it to handle values which cannot be losslessly compressed using ALP's core compression method. (3) Finally, the third part of this thesis measures how performant ALP decoding on GPUs is. ALP decoding should be compared to the state of the art on GPU decompression, and NVIDIA implements a variety of common compression schemes in the nvCOMP library[42]. The GPU implementation of ALP's decoding throughput is compared to these nvCOMP compression schemes.

## 1.1 Research Questions

The implementation and benchmarking of an ALP GPU decoder can be decomposed into three research questions:

> *RQ 1: How should a general purpose API for single value decoding of bit-packed data be implemented?*
>
> *RQ 2: How can exceptions be efficiently patched on GPUs?*
>
> *RQ 3: Can floating-point data be decoded faster with ALP than with common GPU compression schemes?*

## 1.2 Outline

This thesis first discusses the background of the research in Chapter 2. This background starts with a discussion of different kinds of compression approaches and their benefits. After that, the FastLanes file format[2] and ALP[5] are discussed. ALP is part of FastLanes and is a compression encoding for floating-point data. Finally, the nvCOMP library is discussed, nvCOMP is a compression library from NVIDIA that implements common compression schemes. Then in Chapter 3 NVIDIA GPU hardware internals are discussed. This chapter also provides an in-detail description of how programs are executed on GPUs.

Then the thesis continues in Chapter 4 with a discussion of how FastLanes encodings can be decoded efficiently on GPU. First multiple decoding approaches for the core FastLanes bitpacking encoding are discussed, and then multiple decoding approaches for patching exceptions in ALP are discussed. This chapter also designs G-ALP, which uses an alternative data-parallel exception layout to efficiently decode ALP encoded floating-point data on GPUs. In Chapter 5 the throughput of each decoder and exception patcher is extensively measured using microbenchmarks, and concludes with multiple benchmarks which compare the throughput and compression ratio of G-ALP with the compressors in the nvCOMP library on real-world data. Chapter 6, the final chapter, draws conclusions from the benchmarks, provides answers to the research questions and discusses future work.

# Chapter 2

# Background

This chapter explains why compression is useful on GPUs. First the benefits of compression are discussed, and what the advantages and disadvantages are of common compression schemes that rely on heavy-weight compression. Then the benefits of light-weight compression over heavy-weight compression are discussed. FastLanes is then introduced, FastLanes is a new data format that combines these light-weight compression benefits with data-parallel encodings that are ideal for processing with SIMD instructions. The chapter continues with a section that explains why compression is useful on GPUs, how it can speed up memory-throughput-bound kernels and why the FastLanes format is a good fit for data decompression on GPUs. Lastly, a brief explanation of NVIDIA's nvCOMP and Thrust libraries is given.

## 2.1 Compression

Encoding data into a format that uses fewer bits to store the same data is called compression. The original data can be retrieved by decoding the encoded data. Compression is useful for two reasons. The first reason is that less space is needed to store the same amount of data. Saving space can allow users of compression to store more data, or to use smaller storage devices to store that data. The second reason is that transporting encoded data is faster, as fewer bits need to be transferred to transfer the same amount of information.

Commonly *heavy-weight compression* (HWC) schemes are used to compress data. HWC can encode any kind of data, and are widely used in the industry. Examples of HWC are: zstd[11], Deflate[13], LZ4[10], and Snappy[17]. These schemes commonly rely on complex encodings such as Huffman coding[16] or LZ77[61]. Decoding these complex encodings requires many instructions, that are hard to parallelize. Heavy-weight compression schemes also operate on large blocks of data, requiring a relatively large amount of memory during decompression[1]. This high memory usage causes the decompression function to continuously spill data out of caches, introducing a memory

throughput bottleneck. Both the complex decoding and the high memory usage slows down the decoding of HWC significantly.

An alternative to HWC are *light-weight compression* (LWC) schemes. LWC operate on columnar data and rely on simple encodings. These simple encodings are faster to decompress, as decoding can be vectorized, requiring fewer and more parallel instructions, and little memory[62]. Using LWC can speed up database queries by leveraging the high decompression speed. The read throughput of data in kernels can then be increased by loading compressed data instead of normal data, and decompressing that data at the last moment before processing it[59, 15]. Some light-weight encodings are also directly interpretable in encoded form. This property enables databases to operate on those encodings without decoding the data at all.

An example of a LWC is *bit-packing*, a corner stone encoding upon which many other light-weight encodings expand. In binary representation, any integer in the range $[0, 2^b)$ can be represented using $b$ bits. However, on computers integers are represented as 8-bit, 16-bit, 32-bit, or 64-bit integers. If a set of integers is represented as 32-bit integers, but the set only contains values in the range $[0, 2^b)$, with $b < 32$, there are $32 - b$ bits that are always zero. Bit-packing compresses data by reusing these bits to store the bits of the next integer, instead of leaving those bits unused. A 32-bit integer can then contain two 16 bit values, or 16 two bit values. Smart bit-packing arrangements can even store bits of a single value across multiple integers, enabling for example three 32-bit integers to store four 24 bit values.

The *FOR* (Frame Of Reference) encoding expands upon bit-packing by preprocessing values before bitpacking those values[14]. Bitpacking can encode values that fit into a range $[0, 2^b)$ with $b$ bits. However, the smallest value within a range of numbers is not necessarily zero, so the lower bound of this range could be improved. A series of values might be encoded with fewer bits, by scanning a series of values and saving the smallest scanned value as the offset. The range of values $[\mathit{offset}, 2^b)$ is then smaller if $\mathit{offset} \neq 0$. This smaller range can be mapped back to a range that starts with a 0 for bitpacking by subtracting the offset from each value. Then you get a new range $[0, 2^{'b})$ that is smaller if $'b < b$. The values can be decoded by unpacking the values, and adding the offset back to each value to retrieve the original values.

## 2.2 FastLanes

FastLanes is a new next-generation big data format[2]. FastLanes is data-parallelized by utilizing the novel 1024-bit interleaved and Unified Transposed Layout[3], enabling fully parallel decoding even with scalar code on the CPU. Parallel encodings are faster to decompress as there are no dependencies between instructions, and parallel encodings can leverage SIMD instructions for high-throughput decoding of data. FastLanes significantly outperforms the state-of-the-art, achieving an 80× speedup on the M1 processor

compared to Parquet, an industry standard data format, while also achieving a 40% better compression ratio.

FastLanes uses the concept of *vectors* to group values within columns, based on the work of Zukowski *et al.*[7]. Vectors are groups of 1024 values that are compressed with the same parameters. By grouping values together this way, there are many costs that can be amortized across 1024 values instead of incurring them for decoding every value. Vectorization of data decoding also exposes more parallel instructions to the compiler, which can be executed faster in pipeline based processors. When using bitpacking as a compression method, the range of all values of a vector of 1024 values decides with how many bits each value in the vector will be packed using the 1024-bit interleaved layout. The *FFOR* (FastLanes Frame Of Reference) encoding is based on the bitpacking mechanism and contains an offset per vector of values. During decoding, the offset addition is fused with the SIMD instructions for unpacking the values, to be able to keep the data in the SIMD registers.

Exception patching is a technique to even further increase the possible compression ratio, while retaining high throughput decoding performance[62]. If for example an FFOR vector contains a single outlier, the range of values in that vector is significantly increased, requiring all values to be packed using more bits. Instead, by using exception patching, that outlier can be marked as an exception, and separately encoded. Then the FFOR range can be calculated using the range of all non-outlier values. During decoding, packed values that are actually exceptions can be replaced with the value of the exception.

In the FastLanes file format, these different light-weight encodings such as FFOR and exception patching can be combined and cascaded using the Expression Encoding feature of FastLanes. FastLanes defines a set of *encoding operators*[2] and, by using Expression Encoding, each column can be encoded by combining a series of these operators into an *encoding expression*. By combining these LWC expressions, compression ratios equal to HWC schemes such as zstd can be achieved[6].

FastLanes also has an encoding to encode floating-point data, called Adaptive Lossless floating-Point Compression (ALP)[5]. As the name implies, the encoding can losslessly compress both single-precision as well as double-precision floating-point data. The encoding expands upon PseudoDecimals[20]. PseudoDecimals observed that floating-point data in real datasets are commonly storing data that each could also be represented by storing an integer in combination with an exponent $e$. An example of this kind of data is monetary data, which has a limited, and fixed number of decimals. This is important, as the binary representation of floating-point data is not suitable for LWC such as FOR. The data can be compressed better by mapping floating-point data to these integers and exponents, as these integers can then be compressed using run-length encoding and FOR.

ALP improves upon PseudoDecimals in compression speed, decompression speed and

compression ratio[5]. (De-)Compression speed is improved by vectorizing the encoding and decoding, using primitives from the FastLanes compression library. Compression ratio is improved by defining an exponent per vector of values, instead of per individual value. This amortizes the storage cost of the exponent across 1024 values instead of a single value. Additionally, ALP uses an additional multiplication with a factor $f$ to cut off trailing zeros from the integer part of the representation. This is mostly useful when the exponent is comparatively high, causing the resulting integer to contain many trailing zeros.

However, approximately one percent of decimal values cannot be mapped to exact double-precision values using a combination of an integer and exponent. These values are separated from the other values, and handled with the exception patching mechanism. To decode the data, first the bit-packed integers are decoded, and then mapped back to floating-point data. The values that could not be mapped to integers are then replaced with the original values. In cases where most of the floating-point values in a column cannot be represented as a pair of integer and exponent, ALP uses an alternative encoding called ALPrd. ALPrd refers to these values as *real doubles*, as they do not originate from data that could be stored as decimals with fixed precision. ALPrd is not implemented in this thesis.

## 2.3 Compression on GPUs

Compressing data can be promising for GPUs, as all data needs to be transferred via a comparatively slow PCIe link between the GPU's RAM and the CPU's RAM. Compressing data alleviates the problem of a slow PCIe link, because compressed data takes less time to transfer than non-compressed data, as less data needs to be moved. Additionally, more data can be fit into the GPU's RAM by compressing the data.

Shanbahg *et al.* [56] pioneered the idea of leaving the data compressed in the GPU's RAM. This ensures that no memory needs to be allocated in the GPU for the decompressed data, allowing one to store more data on the GPU. Shanbahg *et al.* use a cascaded compression scheme, where the compressed data needs to be read multiple times before the data is fully decompressed. To avoid reading data multiple times from RAM and further increasing pressure on the RAM throughput bottleneck, they store the compressed data in the GPU's shared memory. They then decompress that data during the execution of a query. They adopt the notion of a *tile* of data, which is similar to the concept of a vector of data. A tile is a fixed size group of values that fits in the GPU's shared memory.

This approach of loading compressed data has an additional benefit besides being able to store more data in the GPUs RAM. Loading compressed data from RAM into registers is faster than loading non-compressed data, as fewer bytes need to be loaded to transfer the same amount of data. However, the overhead of decompressing the

tile-based encoding was too large to make use of this transfer benefit.

The bitpacking encoding from FastLanes was also implemented for GPUs[4]. The data-parallel encodings that work well with SIMD instructions also fit the warp-based execution model of GPUs very well. This initial implementation showed that the data-parallel FastLanes encoding makes the decompression overhead much smaller. Because the decompression overhead is smaller, the kernels can leverage the benefit of loading compressed data, and the memory-throughput-bound kernels can actually be faster by loading compressed data. The initial GPU API uses the vectorized execution model—the most widely adopted execution model on the CPU. However, on GPUs this forces kernels to materialize 32 values per thread per column. On GPUs the amount of high-throughput, nearby memory (registers, shared memory, L1 cache) per thread is limited, and materializing 32 values per thread negatively affects occupancy, as will be further explained in Section 3.5.

## 2.4 nvCOMP

NVIDIA's nvCOMP compression framework[42, 39, 46, 53, 23, 30] offers multiple compression schemes to compress and decompress data on the CPU as well as the GPU. The framework is widely used, with more than 608,509 downloads[40]. nvCOMP compression encodings supports multiple data types: bytes, multiple integer types, strings, and half precision floats[38]. nvCOMP contains no direct compression methods for single precision or double precision floats, instead this data can be compressed as a stream of bytes. nvCOMP 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[30]. 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[46]. Within each thread block, only a single warp is used to ensure efficient coordination among threads via warp-level primitives.

CUDA, NVIDIA's programming language for GPUs, requires programmers to manually control when data, and which data, is copied to and from the GPU. There is no spilling mechanism that automatically spills memory to the CPU's RAM if the GPU's RAM is fully used. This lack of spilling poses a problem for heavy-weight compression schemes with arbitrary block sizes, if the format does not contain metadata on the decompressed size for each block. Because otherwise during decompression the program might segfault if there is no more memory to write the contents of the decompressed

block to. nvCOMP extends every compression method that they support with an additional header that contains decompressed block sizes. nvCOMP can also decompress data that does not contain this NVIDIA specific header, but then it will either allocate the maximum needed memory to decompress a block if that size is known, or nvCOMP will do an expensive preliminary pass first to calculate how much memory the decompressed data will need[37].

# Chapter 3

# NVIDIA Graphics Processing Units

This chapter explains what *graphics processing units* (GPUs) are, how they execute programs and how they differ from *central processing units* (CPUs). First the general execution model of GPUs is discussed, which introduces most related terminology regarding GPUs. After that the memory hierarchy of GPUs is discussed. Then warps, the smallest computational unit of GPUs, are discussed. Then there is a section on streaming multiprocessors and a section on occupancy. The chapter concludes with two microbenchmarks. The first microbenchmark highlights how the multi-pipeline architecture of GPUs influence execution time. The second microbenchmark shows that improving instruction-level-parallelism (ILP) increases the throughput of a memory latency bound kernel, using a variant of a pointer chasing benchmark.

This thesis only benchmarks, and refers to, NVIDIA GPUs, not GPUs from other manufacturers. This choice was made as most literature on GPUs uses NVIDIA GPUs, and NVIDIA GPUs dominate in the industry as well.

Generally, the exact hardware of NVIDIA GPUs is not officially documented. The official documentation only consists of white papers and sales information, both of which often lack detail and complete benchmarks. Instead there are a number of papers which perform microbenchmarks to dissect a specific architecture of NVIDIA GPUs, such as the Volta[19], Turing[18] and Hopper[22] architectures.

## 3.1 Execution model

GPUs excel in executing highly parallel programs, where the program needs to repeatedly execute the same computation on a large set of data. Developers can program GPUs by writing CUDA, which is essentially C++ with extra keywords. C++ functions can be configured to run on the GPU by simply writing the `__global__` keyword in front of the function signature. This indicates that this function will be called from the

host (CPU) and run on the device (GPU). These functions are written as if a single thread is independently executing the code.

Threads are grouped into *thread blocks*, which consist of up to 1024 threads. Thread blocks are in turn grouped into *grids*. When calling a `__global__` function, one needs to specify the grid and thread block configuration that will execute that function. For example, when adding two vectors, vector $A$ and vector $B$, that each contain 102400 elements, one could launch a grid of 100 thread blocks, each thread block containing 1024 threads. In that function thread $i$ would add $A[i]$ and $B[i]$ together and store the result at index $i$ in the output vector $C$. Listing 1 shows a minimal example of CUDA code that executes this vector addition.

```
// GPU Function
__global__
void sum_vectors(const int *A, const int *B,
                 int* C, const size_t size) {
  // Calculate index of thread by reading special CUDA variables
  int thread_id = blockDim.x * blockIdx.x + threadIdx.x;

  if (thread_id < size) {
    C[thread_id] = A[thread_id] + B[thread_id];
  }
}

// CPU Function that uses the GPU to execute the sum.
// The d_ prefix hint that these ptrs point to device memory
void sum_vectors(const int *d_A, const int *d_B,
                 int* d_C, const size_t size) {
  const size_t block_size = 1024;
  const size_t grid_size = size / block_size + 1;

  // The <<<int, int>>> is used to configure
  // the thread grid and thread block
  sum_vectors<<<grid_size, block_size>>>(d_A, d_B, d_C, size);
}
```

Listing 1: This code is a minimal example of how programmers can program a vector addition kernel in CUDA.

However, threads are an abstraction over how GPUs actually execute programs. The GPU differs from a CPU in that it does not contain a unique processing instruction pipeline for each individual thread. Instead GPUs contain *streaming multiprocessors* (SMs), which execute multiple thread blocks concurrently. Each SM consists of *warps*. An overview of the GPU internals is given in Figure 3.1. The SM distributes the threads of a thread block among these warps, where each warp is assigned 32 threads. All of the threads within a warp step through the same instruction for the program that they are

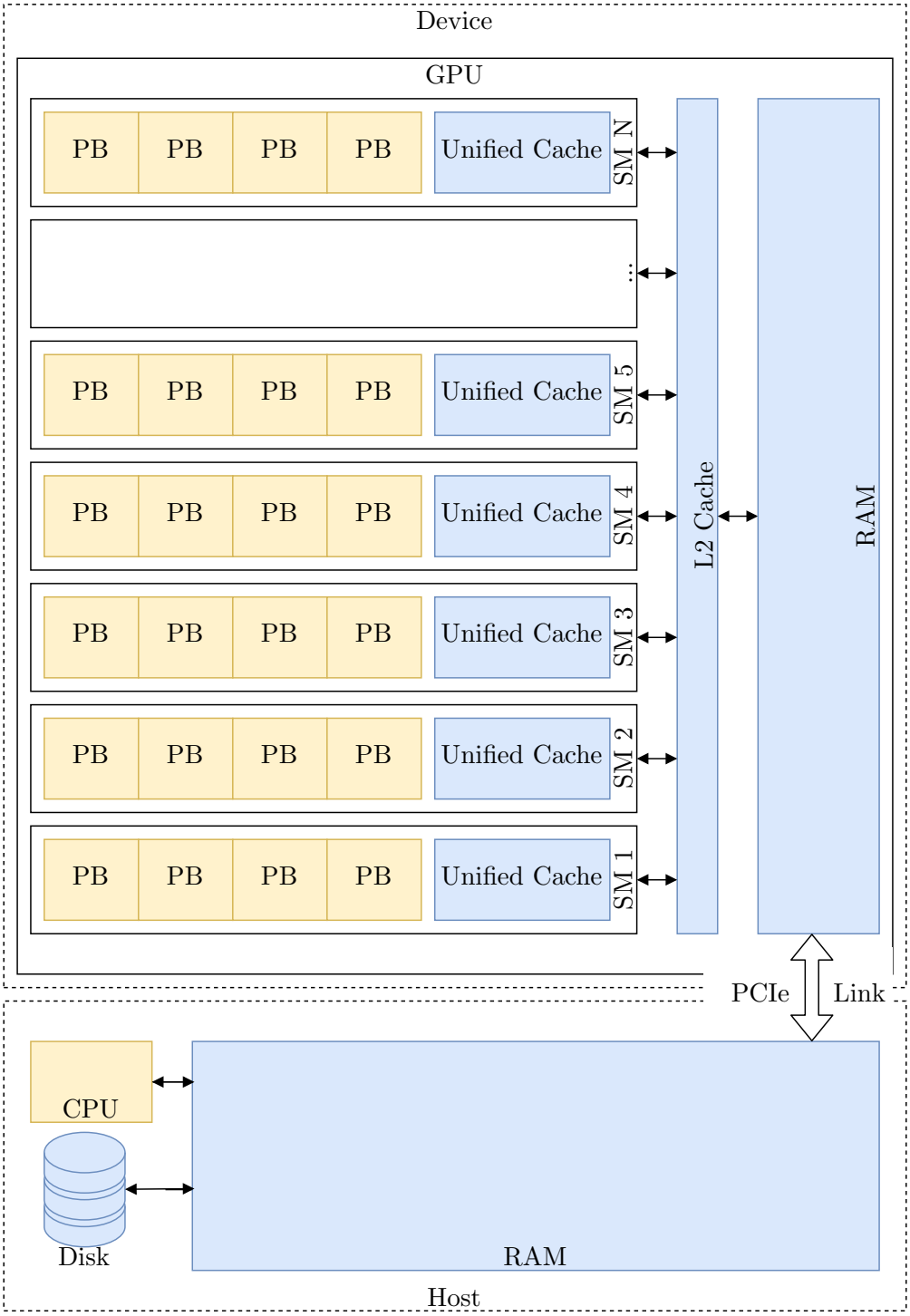


Figure 3.1: Diagram of the internal hardware of the GPU, and how the GPU (device) is connected to the host. Streaming Multiprocessors (SM) consist of multiple Processing Blocks (PB), the exact number of processing blocks varies per SM generation. Blue: Memory units. Yellow: Compute units.

executing. NVIDIA calls this the single instruction, multiple threads (SIMT) model[50].

The warp itself does not actually execute any instructions, the warp only issues instructions[34]. When a warp issues an instruction, the instruction is passed to a separate instruction pipeline which will execute that instruction. There are multiple heterogeneous instruction pipelines in the SM, where each type of pipeline executes only a specific set of instructions. The warps within a SM are divided into groups, *processing blocks*, which share a set of these instruction pipelines[44]. A diagram of a processing block is shown in Figure 3.2.

Each processing block has a *warp scheduler*, which at every cycle 'picks' a warp that may issue an instruction to the instruction pipelines within that processing block. Warps might not be able to issue an instruction if they encounter a *data hazard* or a *control hazard*, then they need to wait for a previous instruction to finish. This can effectively hide the latency of the instruction that the warp is waiting on, because the warp scheduler can then choose to execute the instructions of other warps[35].

## 3.2 Memory

The main memory hierarchy of a GPU consists of an L1 cache, an L2 cache and the device's RAM. Each SM has it's own L1 cache, The L2 cache is shared among all SMs. Global and local memory are logical subdivision of main memory that use the L1 cache conventionally. Additionally, a GPU has three other variants of memory that have a different version of the first level cache: constant memory, texture memory and shared memory. On some GPUs these memories are all still physically located on the L1 cache, and then the L1 cache is referred to as the *unified cache*.

Global memory is the term for regular memory, and is the most common form of memory. The memory resides in the GPU's RAM until it is read, then it is placed in the L1 cache, until it gets evicted into the L2 cache. Global memory accesses that are not cached have high latency and low throughput. For optimal read performance, global memory accesses need to be coalesced. Otherwise a single read transaction will result in loading multiple cache lines to fetch all necessary data[32].

Local memory functions the same as global memory, but the compiler chooses what memory is stored in local memory. The compiler will use local memory to store very large arrays, to store arrays that are dynamically accessed, to pass arguments to device functions that were not inlined, and to spill registers[29]. Because local memory functions the same as global memory, it has higher latency and lower throughput than reading from registers. Therefore it is necessary to be conscious about if a kernel uses local memory, and whether it is possible to avoid using local memory.

Local memory is used when calling functions that were not inlined. The compiler needs to store the state of the registers before entering the non-inlined function into local memory. Then when the function finishes the register state needs to be restored.

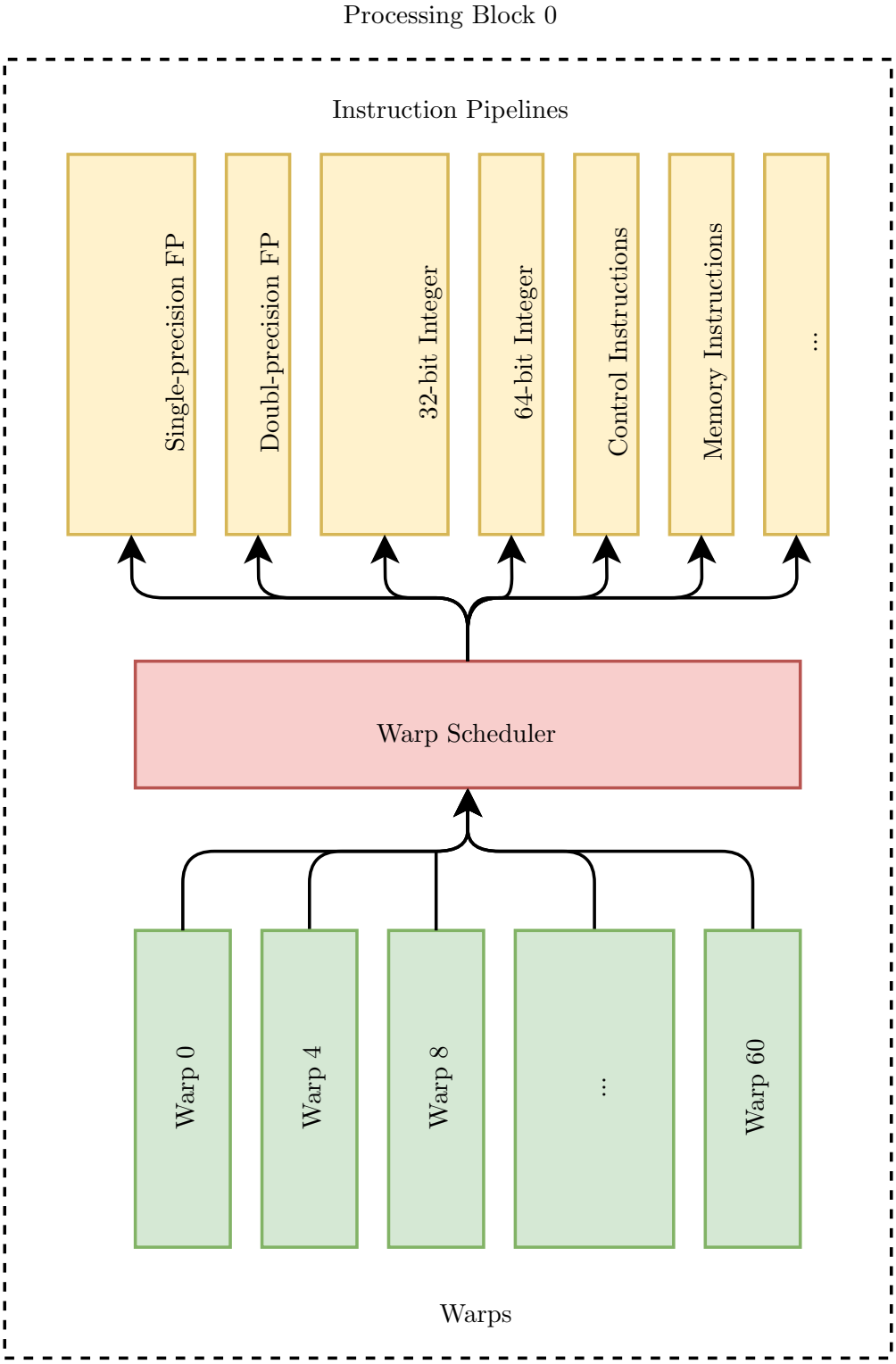


Figure 3.2: Layout of a processing block within a SM. Each cycle, the warp scheduler picks one warp to take an instruction from, and sends that instruction to one of the appropriate instruction pipelines. In this example the first of four processing blocks is depicted, with twice as many single precision as double precision instruction pipelines. Warps are evenly distributed among all processing blocks within a SM, where the function $warp\_index \bmod total\_processing\_blocks = block\_index$ decides to which processing block a warp belongs[19]. The number of processing blocks per SM can differ per compute capability, typically between two and four processing blocks per SM. Yellow: Instruction Pipelines. Red: Warp Scheduler. Green: Warps.

This is relatively slow and increase the load on the already scarce L1 cache memory read throughput. To avoid this problem, the NVIDIA compiler tries to aggressively inline functions.

Constant memory and texture memory are both read-only for code running on the GPU. The constant cache is relatively small, and is optimal when all threads within a warp need to access the same value[51]. If the value is not cached in the constant cache, it is loaded from the GPUs RAM. The texture cache is optimized for repeated 2D array accesses with spatial locality[29].

Shared memory is addressed by memory banks, which enables warps to do uncoalesced access without a performance penalty, conditional on that there are no bank conflicts. Bank conflicts occur when two threads read different memory addresses that are located in the same memory bank. Shared memory allocated to a block can be read from, and written to, at runtime by all threads within that block. This means it is possible to use shared memory for inter-thread communication [54]. Loading data from global memory to shared memory requires the memory to first be loaded into the registers. However there exists a direct copy PTX instruction[47] on compute capability 8.0 and newer, which can be called with inline PTX[28].

Registers represent the fastest form of memory, and the number of registers per thread is dynamic, with a maximum of 255 registers per thread[51]. Registers have the lowest latency and highest throughput of all memory types. Registers are actually not located in the warp themselves, but in the register bank of the SM. However, registers are cached within the warp itself. The warp can cache four registers for each thread. This register caching can help prevent register bank conflicts, although studying these conflicts requires manual inspection and adjustment of the SASS assembly, so it is in most cases not a feasible optimization[19].

The GPUs assembly not only contains normal load instructions, but also vectorized load instructions. Instead of loading a single value per thread per instruction, the vectorized load instructions fetches four consecutive values per thread per instruction. The vectorized load instructions therefore can fetch more memory per instruction. This enables the SM to use fewer load instructions to fully saturate the memory bandwidth. The vectorized load instructions are particularly useful on architectures with relatively low memory instruction throughput[26].

In CUDA the developer manually controls which memory is copied to and from the GPU over the PCIe link, there is no mechanism that can spill memory from the GPU's RAM to the CPU's RAM or to disk.

## 3.3 Warps

CUDA enables developers to program the GPU as if they were programming a single, independent thread. However, on the hardware side the smallest unit of computation

is not the thread, but the warp. Programming those warps as if they only execute a single thread can leave a lot of performance on the table. It is better to think of GPU instructions as vector instructions[58].

Warp divergence occurs when not all threads within the same warp take the same branch within the program[24]. In that case, the warp will execute both branches sequentially. The threads that do not enter the branch are disabled. At run-time the program can query which threads within the warp are disabled by querying the value of the *active mask*[52].

When issueing memory accesses to global memory, each thread can access an individual value. However, if threads access values that are aligned on the same 128-byte cache line, the warp will *coalesce* these memory accesses. This means that instead of fetching the same 128-byte cache line multiple times, it will only fetch this line once, and give each thread it's own requested part of the line. Coalescing is only applied to memory accesses of threads in the same warp, so memory accesses from different warps cannot be coalesced[32].

This coalescing is also the reason why sequential memory access is faster on GPUs than random memory accesses. Sequential memory accesses result in fully coalesced memory accesses, while random memory accesses will need to fetch a different 128-byte cache line from RAM for each thread in the warp, resulting in a 32x decrease in effective memory throughput. As each thread only needs part of the 128-byte cache line, but pays the cost in throughput for fetching the entire line.

## 3.4 Compute capability

The *compute capability* of a GPU indicates what the version is of the SMs in the GPU. Each compute capability can differ from previous ones, changing how much compute throughput, how large the cache sizes are, or changing other properties of a SM. Multiple GPUs can share the same compute capability. These GPUs then differ in other aspects, such as the number of SMs per GPU, the amount of RAM, the type of memory link, etc. The properties of each compute capability can be found in NVIDIA's documentation[25], as well as the compute capability of every NVIDIA GPU[55].

In Table 3.1 the memory and occupancy properties of a selection of compute capabilities can be seen. The selected compute capabilities are 6.1, 7.0, 8.9 and 9.0. Compute capability 6.1 is shown as it is the compute capability of the GPU on which all development for this thesis was done. Compute capability 7.0 is the compute capability of the V100 GPU. The 8.9 compute capability is a compute capability optimized for visual computing, used predominantly in consumer cards, and the development GPU of CWI has this compute capability. Compute capability 9.0 is the most recent datacenter GPU compute capability.

Several of the properties of the compute capabilities are identical. The total number

| Compute capability | 6.1 (GTX 1060) | 7.0 (V100) | 8.9 (RTX 4070) | 9.0 (H100) |
|---|---|---|---|---|
| Max threads per block | 1024 | 1024 | 1024 | 1024 |
| Resident threads | 2048 | 2048 | 1536 | 2048 |
| Resident warps | 64 | 64 | 48 | 64 |
| Resident blocks | 32 | 32 | 24 | 32 |
| 32-bit registers | 65536 | 65536 | 65536 | 65536 |
| Max registers per thread | 255 | 255 | 255 | 255 |
| L1 Cache (kb) | 48 | 128 | 128 | 256 |
| Shared memory (kb) | 96 | 96 | 100 | 228 |

Table 3.1: The resources available per SM for four different compute capabilities. Note that the total number of registers per SM stays constant, while the L1 cache and shared memory cache increases. The number of threads per SM is also the same for each SM, except the RTX 4070, which is a consumer GPU, optimized for visual compute.

| Compute capability | 6.1 (GTX 1060) | 7.0 (V100) | 8.9 (RTX 4070) | 9.0 (H100) |
|---|---|---|---|---|
| Registers | 32 | 32 | 42.7 | 32 |
| L1 Cache (bytes) | 24 | 64 | 85.3 | 128 |
| L1 Cache (4 byte values) | 6 | 16 | 21.3 | 32 |
| Shared memory (bytes) | 48 | 48 | 66.7 | 114 |
| Shared memory (4 byte values) | 12 | 12 | 16.7 | 28.5 |

Table 3.2: On-chip memory resources per thread at 100% occupancy. The number of registers per thread at 100% occupancy is equal for all SMs, except the RTX 4070, which was optimized for visual compute. Each thread has access to very little L1 cache space at 100% occupancy, only able to store 32 integers or 16 long integers on average with the latest SM.

of registers is the same, as well as the maximum number of registers that can be assigned to a thread, and the maximum number of threads that can be assigned to a single block.

All of the compute capabilities also have a maximum number of resident threads, warps and blocks. This means that there cannot be more threads, warps or blocks active than that maximum. This has important implications for the block sizes of kernels. Because all compute capabilities have twice the number of resident warps the number of resident blocks, a thread block should always consist of at least two warps. Otherwise the bottleneck is the maximum number of active blocks instead of active warps.

The compute capabilities that specialize in visual computing differ by having only 1536 maximum threads per SM, and also a reduced number of resident warps and blocks. Because other properties such as the total number of registers does not decrease for these cards, there are more registers available per thread.

Table 3.2 shows the available on-chip memory per thread. In comparison to CPUs the number of registers and the L1 cache of SMs is relatively large, but when sharing those resources among all threads, the on-chip memory resources are very scarce. A single thread can only have a resident set of up to 32 integers or single-precision floats in the

L1 cache. This effect is aggravated when considering that L1 cache lines on NVIDIA GPUs are 128 bytes wide[32], so when all threads access values that are on unique cache lines, the number of values that can be cached is further reduced. So a GPU thread can only have 32 values cached in the L1 cache under optimal circumstances, whereas CPU threads commonly can store KBs of data in the L1 cache.

## 3.5 Occupancy

The occupancy of a kernel is the ratio of active warps to the maximum total number of warps for a specific kernel[45]. High register usage, high shared memory usage, or low threadcount per block can decrease the occupancy for a specific kernel.

High occupancy indicates that there are many warps active on the SM. Having many warps active concurrently allows the GPU to hide latency better. While some warps are waiting on instructions with a long latency, other warps can issue instructions to make use of the SM's resources. This can hide the latency of instructions, as the SMs resources were still fully saturated while waiting for instructions with higher latency. So high occupancy increases the number of active warps, and therefore increase the chance that there are warps that can issue an instruction while other warps are stalled.

High register pressure means that the kernel uses many registers per thread. As the total amount of registers per SM is fixed, the SM will not activate every warp, thus decreasing the occupancy. By not activating every warp, the warps that are activated can still be assigned enough registers[49].

The exact maximum occupancy due to register usage is best calculated by using NVIDIA's Occupancy Calculator[45], as there are multiple factors that influence the occupancy for a certain number of used registers per thread. Registers cannot be allocated to a block in increments of one, neither are register allocated to a single thread, but only to a warp as a whole. Saving a single register can have a large effect on maximum occupancy, as it might allow the block to save a new allocation of a large number of registers. But saving a single register can also have no impact, if there is no lower threshold of allocated registers in reach.

High shared memory usage is also problematic for occupancy, as there also needs to be enough shared memory for all active blocks. Allocating all of the available shared memory to a single block will ensure that no other block of that kernel can run concurrently on the same SM. This limitation can however be leveraged in microbenchmarks to test the performance of a single thread in isolation[58], by allocating the maximum shared memory to a single block consisting of a single thread.

As mentioned in Section 3.4, allocating less than two warps per block can also decrease occupancy, as the number of maximum resident warps is at least twice as large than the maximum number of resident blocks. Any block using less than two warps will therefore suffer a decrease in occupancy.

Registers and shared memory are allocated to a block, until the last thread of that block finishes[27]. This is important, as any thread that is an outlier in regard to execution time will ensure that the resources of the entire block cannot be released until that thread finishes. So it is also possible that the occupancy is decreased by having a single long running thread, as the next block cannot be scheduled until that thread finishes.

## 3.6 Assembly

CUDA code is compiled by nvcc[41] to PTX[47], which is in turn compiled to SASS, which has a listed, but undocumented instruction set[43]. PTX is an intermediary language which ensures that every binary can run on every NVIDIA GPU. Each generation of SM can have different assembly instructions. SASS is the actual assembly language that is compiled to the machine code which runs on the GPU[48].

The most important note about the SASS instruction set is that, unlike the PTX instruction set[47], there is no actual division instruction. Instead, the PTX division instruction is translated into approximately forty SASS instructions that actually executes the division[1]. Other optimizations, such as inlining, are also not yet applied in the PTX[2] binary. This means that evaluating PTX is not particularly useful for any performance insights, instead it is more beneficial to study the SASS that is generated from that PTX.

The actual machine code generated from SASS differs in some aspects from architecture to architecture. Details about the machine code were uncovered by academic papers[18, 60]. Since the Kepler architecture, the machine code consists of two parts: the instruction itself, and the control logic. The control logic is read by the warp scheduler, and controls how warps are picked to issue instructions. There are a multitude of open source assemblers created by third parties in order to be able to manipulate the machine code by manually writing SASS code[3,4,5,6,7], as the NVIDIA provided toolchain does not offer an option to do this. Although these open source assemblers are unmaintained or archived, and only target a specific architecture, a developer can still study the source code to learn more about generating NVIDIA GPU machine code. There is also a utility available that can retrieve hardcoded instruction latencies from the NVIDIA assembler[8].

GPUs contain no branch predictor, and a warp issuing a branch instruction will simply stall[31]. There is currently also no out-of-order execution on GPUs[8]. Branching

[1] https://godbolt.org/z/q6GaWf6Ef
[2] https://godbolt.org/z/1WW7Whazx
[3] https://github.com/cloudcores/CuAssembler
[4] https://github.com/NervanaSystems/maxas/
[5] https://github.com/hyqneuron/asfermi
[6] https://github.com/daadaada/turingas
[7] https://github.com/laanwj/decuda
[8] https://github.com/0xD0GF00D/DocumentSASS

in GPU code therefore has different performance characteristics than branching in CPU code. Stalling is not necessarily negative, as another warp can then issue instructions if there is a warp waiting.

## 3.7 Microbenchmark: Heterogeneous Instruction Pipelines

As explained earlier, warps only issue instructions. These instruction are then executed on a heterogeneous set of instruction pipelines. Each instruction pipeline only executes a specific set of instructions. There are different instruction pipelines for memory instructions, single precision arithmetic, double precision arithmetic, control flow and more. Each compute capability can have different types of instruction pipelines, and can have different ratios of throughput per type[25]. The execution time of a kernel can be improved by changing it's instruction mix. If a specific type of instruction pipeline is currently the bottleneck, part of the kernel could be changed to use less instructions of that bottleneck type, in exchange for more instructions that are executed on a less used type of instruction pipeline.

To prove that changing the type of instructions works, we benchmark the execution times of two different functions. One function exclusively executes arithmetic instructions, and the other function uses mainly memory instructions in exchange for fewer arithmetic instructions. What these functions actually do is irrelevant, as long as they issue different kinds of instructions, and most importantly take approximately the same amount of time to execute. In a real world scenario one could equate this to two different implementations of the same function, where microbenchmarks show neither implementation to actually perform better than the other.

```
1  template<typename T>
2  __device__ T arithmetic(T value, const T zero) {
3  #pragma unroll
4    for (int i{0}; i < (N_ARITHMETIC_INSTRUCTIONS / 2); ++i) {
5      // Each of these instructions sets value to zero
6      value *= zero;
7      value &= zero;
8    }
9
10   return value;
11 }
```

Listing 2: Example function that exclusively issues arithmetic instructions. For loop is unrolled to remove branch and jump instructions. The variable zero contains the value zero at runtime, but is set to zero only at runtime to prevent the compiler from optimizing away the instructions in the for loop, as the function would always return zero.

The code of Listing 2 shows a bogus calculation that is repeated multiple times. One of the function parameters is called zero, this variable contains zero at runtime. The zero is not hardcoded, as then the compiler would be able to optimize away the instructions within the for loop. The for loop is also completely unrolled, to avoid any bound checking and branching instructions. The `N_ARITHMETIC_INSTRUCTIONS` constant is divided by two as there are two instructions per iteration. There are two different instructions per iteration, as the compiler can often optimize away a single repeated instruction. The main point of this function is that it exclusively issues a large number of arithmetic instructions.

```
template<typename T>
__device__ T memory(T value, const T zero, const T *ptr_to_zero) {
  value *= zero; // Ensure value is set to zero at runtime

#pragma unroll
  for (int i{0}; i < N_MEMORY_INSTRUCTIONS; ++i) {
    // Effectively repeatedly adds zero to zero
    value += ptr_to_zero[value];
  }

  return value;
}
```

Listing 3: Example function that exclusively issues memory load instructions. For loop is unrolled to remove branch and jump instructions. The pointer to zero variable points to a variable or array containing a single zero to ensure that the value that is returned is always zero, which will cause repeated accesses to the same memory address. Therefore the memory that is accessed in this function will most probably reside in the L1 cache, as all threads repeatedly access the same value.

The code of Listing 3 also shows a bogus calculation that is repeated multiple times. In this case the additional function parameter `ptr_to_zero` is a pointer to memory that consists of a single zero. As `value` is set to zero at runtime, it will effectively repeatedly load the same value, so the memory it loads resides in the L1 cache. The function is therefore bound by the memory throughput of the L1 cache. The value is only set to zero at runtime, as otherwise the compiler could optimize away the for loop. The main point of this implementation is that it exclusively issues memory instructions, instead of arithmetic instructions.

Both of these implementations were separately executed by 104,857,600 threads, and the resulting execution times can be seen in Table 3.3. The `N_ARITHMETIC_INSTRUCTIONS` is set to 900, and `N_MEMORY_INSTRUCTIONS` is set to 60. These values were chosen to ensure that the implementations take approximately the same amount of time to execute. Therefore, the execution times in the table are almost the same. This experiment

establishes the execution times of both the functions in isolation.

| Kernel | Execution time (ms) |
|---|---|
| Arithmetic | 109.46 |
| Memory | 106.94 |

Table 3.3: This table shows the execution times of the arithmetic function and memory function in isolation. The number of arithmetic instructions and memory instructions executed by each type of kernel was adjusted to approximately equalize the total execution times of the kernels.

To actually show how we can leverage the fact that warps share a set of heterogeneous instruction pipelines, we construct another kernel, which calls two functions, instead of only one function. There are four variants of this kernel, one where the arithmetic implementation is called twice, one where the memory implementation is called twice, and two variants where the first call uses the arithmetic implementation, and the second call the memory implementation, or vice versa.

| Kernel | Execution time (ms) |
|---|---|
| Arithmetic, Arithmetic | 185.10 |
| Arithmetic, Memory | 106.06 |
| Memory, Arithmetic | 112.98 |
| Memory, Memory | 210.37 |

Table 3.4: This table shows the execution times of all variants for a kernel that performs two function calls. Alternating the type of functions is faster than using the same function twice.

In Table 3.4 the results can be seen for all variants of the kernel that performs two function calls. The results table show that, as expected, calling the same function twice will approximately take twice as long as the execution time of that implementation in isolation. However, the execution times for the kernel where the first and second functions execute different types of instructions are faster than twice the execution speed of either implementation in isolation.

This speedup is caused by the fact that the kernel can use the combined throughput of two different instruction type pipelines, instead of only issuing instructions for one instruction pipeline. The fact that warps can be in different stages of the kernel, executing a different type and set of instructions, causes them to not have to wait for access on the same instruction pipeline type. In the kernels where the invocations do not alternate between implementations, both warps will be waiting on the same overloaded arithmetic, or memory instruction pipelines, instead of spreading the load on multiple instruction pipeline types.

## 3.8 Microbenchmark: Instruction-Level Parallelism

On CPUs, a naive compilation of a program might produce an instruction stream that contains many data or control dependencies between subsequent instructions. Compilers aggressively reorder instructions to exploit as much *instruction-level parallelism* (ILP) as possible. By executing independent instructions in between interdependent instructions, the processor is less likely to need to stall to wait for a dependency on a previous instruction to resolve[21].

GPUs can use occupancy, and switch warps, to hide latency when other warps stall. However this might not always be enough, or result in the most optimal kernels. Increasing ILP can also improve performance of kernels on GPUs. By avoiding stalling altogether, the throughput of individual warps can be increased[57]. The NVIDIA compiler is aware of this concept, and will aggressively reorder instructions to increase the parallelism between long latency instructions as much as possible. Instead of only issuing one read to global memory, and wait for the result, the compiler will put all independent memory instructions next to each other. Instead of needing to stall to wait for each latency, the latencies are overlapped and the warp will receive the results of the other reads roughly by the time the first read is resolved. Although the compiler will reorder instructions to leverage ILP as much as possible, kernels can sometimes be rewritten to expose more ILP for the compiler to exploit.

To show the effect of increasing ILP , and to show that sometimes even 100% occupancy is not enough to hide the latency of instructions, a variant of a pointer chasing benchmark is benchmarked. Pointer chasing benchmarks are often used to measure the latency of memory requests, but are also ideal to show the effect of ILP, because pointer chasing benchmarks almost exclusively consist of interdependent instructions with high latency.

A pointer chasing benchmark repeatedly accesses a large array that consists of a random permutation of the arrays indices. The index of the next array element to fetch is the value of the last array element that was fetched. This ensures that the compiler cannot reorder instructions to overlap latencies, as the address of the next memory access depends on the previous memory access. The latency of a read instruction can then be calculated by dividing the total execution time by the number of pointers chased.

The kernel that will show the importance of ILP on GPUs will let 10,240,000 threads chase 100 pointers each through an array of 10,000,000 elements. This array is too large to fit into the L1 cache or L2 cache of the GPU, and the random walk through the array ensures that most memory reads will need to read their data from RAM. To simplify the benchmark it is ensured that warps will always fetch a single cache line per read instruction by only enabling a single thread per warp. This avoids becoming throughput bound by issuing uncoalesced memory accesses. The warp to array element mapping is given in Figure 3.3. The pointer chasing benchmark will be executed with varying

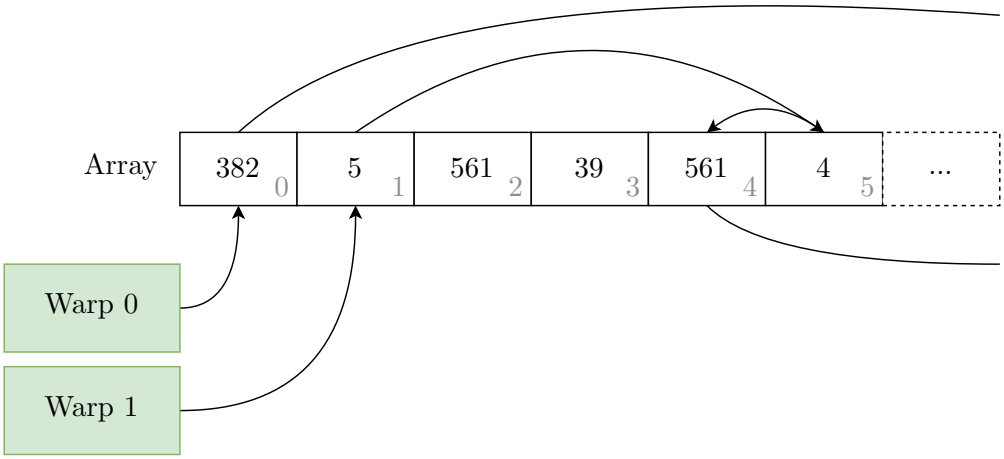


Figure 3.3: Example of how two warps chase pointers through a randomly permuted array of indices. Warp $n$ first fetches the value in the array at index $n$. Then it retrieves the next value at $array[value]$. This step is repeated 100 times. To simplify the benchmark measurements, only a single thread is active per warp.

amounts of occupancy. Occupancy is limited by allocating the maximum amount of shared memory per thread block, ensuring that only one or two thread blocks per SM are active. Then the desired occupancy can be configured by increasing or decreasing the number of threads per thread block.

The experiment will be repeated with varying amounts of ILP. ILP is increased in this case by letting a single thread concurrently chase multiple pointers. The number of launched threads is then decreased proportionally, to ensure that the total amount of pointers chased remains the same. Figure 3.4 depicts this new warp to element mapping, and Listing 4 shows the core code for this benchmark. The compiler is able to reorder the instructions to overlap memory read latencies if the `ILP_FACTOR` is higher than 1.

Figure 3.5 shows how effective exposing ILP to the compiler can be. In this kernel that mainly consists of instructions with very high latency, improving ILP is a strict improvement for each ILP factor. By chasing only a single pointer per thread, the maximum throughput is not even reached at 100% occupancy. Maximum throughput can only be obtained by chasing multiple pointers concurrently. For any level of occupancy, increasing ILP gives a higher throughput. With an ILP factor of 8, maximum throughput can already be obtained with an occupancy of just 40%.

In conclusion this microbenchmark shows that ILP is not only relevant on CPUs, but also on GPUs. Exposing more ILP to the compiler can be necessary at even 100% occupancy to reach optimal performance in latency bound kernels. Increasing ILP does not improve kernels that are throughput bound, as overlapping latencies is not needed when instruction pipelines are already saturated. Latency bound kernels can also be rewritten to focus less on reaching 100% occupancy, because with more ILP less occupancy is needed to offset long latency. Being able to focus less on occupancy opens up more design space for other optimizations.

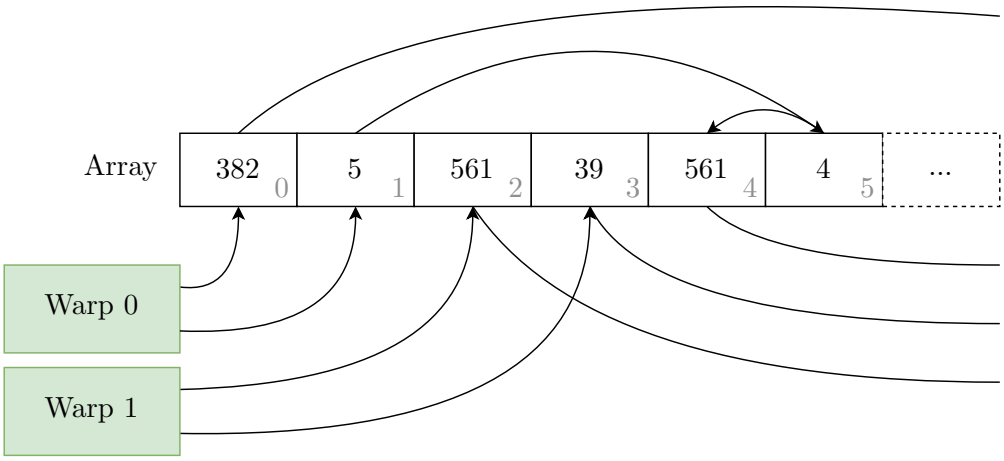


Figure 3.4: Example of how two warps chase multiple pointers through a randomly permuted array. In this case each warp chases two pointers through the array. Warp $n$ now starts at index $n*m$ when each warp chases $m$ pointers. When chasing multiple pointers, each warp can concurrently issue the read instructions for each pointer, without waiting for the previous read to finish. The total amount of activated warps is proportionally decreased, to correct for the fact that each warp chases multiple pointers. The total amount of pointers chased remains constant.

```
template <unsigned CHASE_N_PTRS, unsigned ILP_FACTOR>
__device__ int chase_ptr(const int *array, int* chasers) {
#pragma unroll
  for (int i{0}; i < CHASE_N_PTRS; ++i) {
#pragma unroll
    for (int v{0}; v < ILP_FACTOR; ++v) {
      chasers[v] = array[chasers[v]];
    }
  }
}
```

Listing 4: Function that chases multiple pointers concurrently through an array. For loops are unrolled to avoid branch and jump instructions, as well as to allow the compiler to put all the memory load instructions next to each other in the generated assembly. The `ILP_FACTOR` variable indicates how many pointers are concurrently chased.

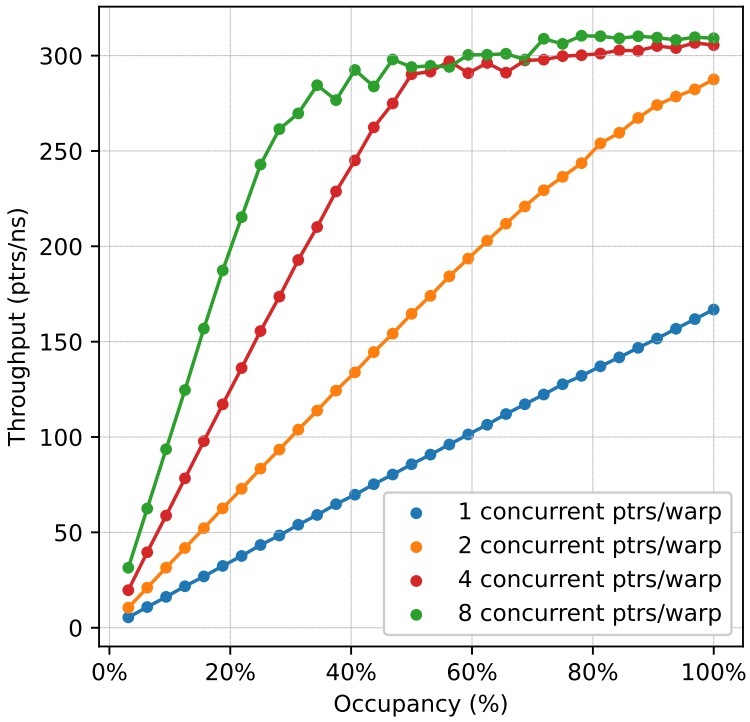


Figure 3.5: Increasing the ILP factor in benchmark increases throughput at any level of occupancy. Maximum memory read throughput can be reached with occupancy levels lower than 100%. Maximum memory read throughput is not reached, even at 100% occupancy, when each warp only chases 1 pointer concurrently.

# Chapter 4

# GPU Decoding Algorithms

This chapter explains how the FastLanes and ALP algorithms, that were designed to leverage SIMD instructions on CPUs, are redesigned to perform well on GPUs. The chapter starts off with studying what the initial implementation of FastLanes on GPUs did, and then explains both the advantages as well as the disadvantages of that implementation. To solve the disadvantages, a new implementation is proposed that decodes values one at a time, instead of entire vectors.

This chapter studies how ALP's CPU exception patching approach can be optimized by decoding one value at a time. After that a section follows which explains how the exception layout can be improved to allow for more data-parallelism during the patching of exceptions. The chapter concludes with an explanation of a minor performance improvement on the decoding algorithm of the data-parallel exception layout.

## 4.1 FastLanes Decoding

Regular FastLanes decodes entire vectors at a time, and all of the necessary decoding steps are based on the bitwidth that was used to pack, encode, the vector with. The implementation uses `switch` statements to the correct hardcoded block of instructions for a specific bitwidth. There is a hardcoded block for each possible bitwidth including 0. Algorithm 1 shows the pseudocode for this decoding approach. The initial FastLanes GPU implementation is copies the CPU implementation, except that each thread only decodes a single lane within a vector.

The hardcoded instruction blocks contain no redundant instructions, and the order of instructions can be fully optimized by the compiler. As most of the hardcoded instructions are independent of each other, the compiler can exploit instruction-level parallelism by grouping independent instructions together. Especially important is that all the load instructions can be grouped together, ensuring that the load latencies are overlapped and thus much cheaper. The hardcoded instruction blocks can be reached efficiently, as there is only a single branch, and the branching cost can be amortized

**Algorithm 1** FastLanes lane decoding

```
1: PACKED_VECTOR_POINTER ← load(PACKED_VECTOR_POINTER + LANE)
2: DECODED_INTEGERS[32]
3: switch BIT_WIDTH do
4:     case 1
5:         PACKED_INTEGER ← load(PACKED_VECTOR_PTR + 0)
6:         ...              ▷ Hardcoded instructions to unpack into DECODED_INTEGERS
7:         break
8:     case 2
9:         FIRST_PACKED_INTEGER ← load(PACKED_VECTOR_PTR + 0)
10:        SECOND_PACKED_INTEGER ← load(PACKED_VECTOR_PTR + 32)
11:        ...              ▷ Hardcoded instructions to unpack into DECODED_INTEGERS
12:        break
13:    case 3
14:        FIRST_PACKED_INTEGER ← load(PACKED_VECTOR_PTR + 0)
15:        SECOND_PACKED_INTEGER ← load(PACKED_VECTOR_PTR + 32)
16:        THIRD_PACKED_INTEGER ← load(PACKED_VECTOR_PTR + 64)
17:        ...              ▷ Hardcoded instructions to unpack into DECODED_INTEGERS
18:        break
19:    ...                                  ▷ Continue for all other bitwidths
20: return DECODED_INTEGERS
```

across all values in that vector.

However, decompressing one lane per thread is undesirable on GPUs as it requires an output array of 32 or 64 values. This heavily increases register pressure, reducing occupancy. In specialized kernels this might not be problematic, but as the goal is to produce a decoding API for general use, the API should not put limitations on occupancy.

Hence, the goal is to design an API that can decompress vectors one value at a time. This would require no output array, but a single register to store the result in, thus decreasing the register pressure of the API.

### 4.1.1 Single Value Decoding

The single value decoding API decodes the values in a lane in order. Every time the decoding function is called, the next value in the lane is decoded. When decoding a single value at a time, the `switch` statement based approach becomes more complex. Instead of a single branch on the bitwidth, an additional branch is now needed to pick the right instructions to decode that specific position in the lane. The cost of a single branch is no longer amortized over 32 values, but instead it is needed to branch twice every time a value is decompressed.

Splitting up the code blocks into code blocks that decode a single value also removes the benefits of hardcoded instructions. No longer can the loads be trivially grouped next to each other due to the control hazard of the new branches. This removes the

huge advantage of putting all loads together. You still retain the benefit of executing only the bare necessary instructions within those code blocks, but database kernels that benefit from compression on the GPU are limited by memory read throughput during decoding, not by compute throughput.

The number of hardcoded instruction blocks also increases. 32-bit types requires 32-bitwidths multiplied by 32 positions, is 1024 blocks. 64-bit types requires 64-bitwidths multiplied by 64 positions, is 4096 cases. This would heavily increase the binary size, especially considering that the compiler aggressively inlines functions as function calling is expensive on GPUs. Additionally, instruction cache latency might become a problem, because for each uniquely used bitwidth in the data, the common set of instructions grows, potentially larger than the instruction cache.

Instead, the GPU implementation of FastLanes single value decoding uses a more simple approach using branches, as shown in Algorithm 2. The implementation continuously buffers an integer from the packed vector, that contains multiple encoded values. Each time the decoding function is called, the next packed value is extracted from the buffer. When all values from a buffer are loaded, the buffer is refreshed by loading the next packed integer from the packed vector. If an encoded value spans multiple packed integers, the remaining bits from the next buffer are fetched and combined with the bits from the initial buffer.

The buffer can reside in cache, local memory, registers, or shared memory. The buffer can also be expanded to multiple packed integers. In that case refreshing the buffer executes two or more reads at a time, enabling the compiler to overlap the latencies of the two reads. When the first buffer is fully decoded the decoder will decode the second buffer before refreshing both buffers simultaneously.

**Algorithm 2** FastLanes single value decoding

*Phase – Initialization*

```
1: BUFFER ← load_next_packed_integer()
```

*Phase – Decode one value*

```
2: DECODED_INTEGER ← UNPACK_INTEGER_FROM(BUFFER)
3: if BUFFER_IS_EXHAUSTED then
4:     BUFFER ← load_next_packed_integer()
5:     if PACKED_INTEGER_SPANS_TWO_LINES then
6:         UNPACKED_REMAINING_BITS ← (UNPACK_INTEGER_FROM(BUFFER) << OFFSET)
7:         DECODED_INTEGER ← DECODED_INTEGER | UNPACKED_REMAINING_BITS
8:     end if
9: end if
10: return DECODED_INTEGER
```

The downside of the approach in Algorithm 2 in comparison to Algorithm 1 is that there are more instructions needed, as the implementation cannot rely on specialized hardcoded blocks of instructions. Additionally, this implementation contains branches.

However, compute throughput is not the bottleneck on GPUs when decoding for most value bit widths.

### 4.1.2 Branchless Single Value Decoding

Single value decoding can also be implemented branchlessly as shown in Algorithm 3. To enable branchless encoding, the implementation always fetches two packed integers from the packed vector. Then it selects which bits it needs from both packed integers and combines these into a single decoded integer. The implementation always fetches two packed integers to handle the case when a packed value spans two packed integers. For any value that is fully encoded into a single packed integer, the number of selected bits from the second packed integer is zero.

**Algorithm 3** FastLanes branchless single value decoding

*Phase – Initialization*

```
    BIT_MASK ← SET_FIRST_N_BITS(BIT_WIDTH)
 1: OFFSET[0] ← 0
 2: OFFSET[1] ← 32
```

*Phase – Decode one value*

```
 3: BUFFER[0] ← load(PACKED_VECTOR_PTR + 0)
 4: BUFFER[1] ← load(PACKED_VECTOR_PTR + 32)

 5: DECODED_FIRST_HALF ← BUFFER[0] >> OFFSET[0] & BIT_MASK
 6: DECODED_SECOND_HALF ← (BUFFER[1] & (BIT_MASK >> OFFSET[1])) << OFFSET[1]
 7: DECODED_INTEGER ← DECODED_FIRST_HALF | DECODED_SECOND_HALF

 8: PACKED_VECTOR_PTR ← PACKED_VECTOR_PTR + (OFFSET[1] ≤ BIT_WIDTH) · 32
 9: OFFSET[0] ← (OFFSET[0] + BIT_WIDTH) % 32
10: OFFSET[1] ← 32 - OFFSET[0]

11: return DECODED_INTEGER
```

A downside of the branchless implementation is that it always performs two reads instead of just one, and the reads cannot be buffered in registers or other forms of memory. The implementation relies on the L1 cache for fast access to values that were read previously. The performance of this implementation is also still fast, as the bottleneck is reading data from RAM, not from the L1 cache. The benefits of the branchless implementation are that the instructions do not contain control hazards, giving the compiler more opportunities to interleave independent instructions. The executed instructions is also constant for any bitwidth, this prevents warp divergence when one warp would be decoding multiple vectors, as is the case when one warp decodes two 64-bit integer vectors.

## 4.2 ALP Decoding

Algorithm 4 shows the process of patching exceptions on the CPU. First all packed data gets decoded into integers and mapped back to floating-point data. Then a loop construct follows that reinserts the original values for all positions in the output array that were originally exceptions[62]. This approach of patching after decoding avoids needing to branch during the decoding of every single integer to check whether it was an exception. The overhead of the loop is negligible, as exceptions are relatively rare, and the code within the loop small.

**Algorithm 4** ALP decoding

```
1: DECODED_INTEGERS ← FFOR(BIT_PACKED_ARRAY)
2: DECODED_DOUBLES ← ALP(DECODED_INTEGERS)
3: EC ← 0
4: while EC < EXCEPTION_COUNT do
5:     DECODED_DOUBLES[EXCEPTION_POSITIONS[EC]] ← EXCEPTIONS[EC]
6:     EC ← EC + 1
7: end while
8: return DECODED_DOUBLES
```

However, this approach requires that the decoding phase produces an output array with all decoded values. As the decoding process on the GPU only decodes single values at a time, there is no output array of values to be patched. Also, even if that array did exist, it could no longer be placed in registers, as patching consists of dynamic access into that array, forcing the array into slower local memory.

### 4.2.1 Single Value Decoding

Single value decoding needs to check upon decoding each value whether that value is an exception. Each thread needs to scan the whole exception positions array for exceptions that occurred in the thread's lane. Algorithm 5 shows how after decoding, this requires the thread to enter a loop to iterate over all exception positions to find the first position that is either equal or higher than the position of the value it is currently decoding. If the thread scans a position that is higher, the current value is not an exception. The check after the loop checks whether the position is equal to the position of the value it is currently decoding, and if so replaces the decoded value with the original exception value.

The scan loop is very costly, as scanning for whether a value is an exception can take longer than the actual decoding of the value. Additionally in most cases the warp diverges at this point, as each thread is scanning for different positions in the position array. Most positions that a thread scans will not be relevant for that thread, as the positions will not be part of that thread's lane. The threads therefore need to do redundant, duplicate work, and scan values that are only relevant for a single thread.

**Algorithm 5** ALP single value decoding

*Phase – Initialization*

```
1: EC ← 0
```

*Phase – Decode one value*

```
2: DECODED_INTEGER ← FFOR(BIT_PACKED_ARRAY)
3: DECODED_DOUBLE ← ALP(DECODED_INTEGER)
4: while INDEX_DECODED_TUPLE > EXCEPTION_POSITIONS[EC] do
5:     EC ← EC + 1
6: end while
7: if INDEX_DECODED_TUPLE == EXCEPTION_POSITIONS[EC] then
8:     DECODED_DOUBLE ← EXCEPTIONS[EC]
9: end if
10: return DECODED_DOUBLES
```

### 4.2.2 Data-Parallel Exception Patching

The solution to the expensive scan loop would be for threads to be able to only read their own exceptions, and not encounter any exceptions that are not within that thread's lane. This would enable the implementation to remove the expensive scanning loop, as it is guaranteed that the next exception position is the next exception that the thread needs to patch.

To enable the thread to only read exceptions that are relevant to that thread, the exceptions array and exception positions array needs to be reordered. Instead of ordering the arrays by position, the arrays are first ordered by the lane in which the exceptions occur. First all exceptions in lane 0 are written, then all exceptions in lane 1 and so on. Within each lane's section, the exceptions are sequentially ordered by position, with exceptions with lower positions first.

Now the threads only need to be able to directly find their lane's section within the overall exception array. The threads also need to know the number of exceptions within their lane, to avoid reading positions that are not in their own lane.

One solution would be to create a new array of lane exception counts. The maximum exception count within a lane in FastLanes is 64 (requiring $2^6$ bits to store the count), as that is the maximum amount of values in a single lane. Threads could then sum the exception counts of all preceding lanes to find the correct offset of their lane's exception section starts. Obviously, executing this prefix sum would be an expensive step during the decoding.

Alternatively, instead of storing the exception counts, the offsets within the exception array could be stored in the array. The maximum offset is $1023 - 8 = 1015$ (requiring $2^{10}$ bits to store the offset), as that is the offset of the last lane when all values of all previous lanes were exceptions, with the smallest granularity of lanes consisting of 8 values for the `uint_8t` data-type. To make sure that the thread does not read any values outside

of the thread's lane, it could calculate the number of exceptions in the lane by reading the offset of the next lane, and subtracting that offset with the thread's lane's own offset to obtain the exception count.

The offset approach is faster, as it requires no expensive loops, but just two read instructions for each thread to read the offsets it needs. However, the last lane needs to read the first offset of the next vector to calculate the exception count of the thread's lane. This is slightly awkward as the offset of the next vector can be located on a different cache line, requiring the warp to fetch two cache lines for a single read instruction, which is slightly more expensive than a perfectly coalesced read instruction.

There is a third approach that combines both previous approaches. As storing the count requires six bits, and storing the offset requires ten bits, both can be packed into a single 16 bit integer. This does not decrease the compression ratio, as storing the ten bit offset would already require an array of 16 bit integers. Now the thread can get the number of exceptions in lane, as well as the lane's offset within the overall exception array, with a single coalesced read instruction. Figure 4.1 shows a diagram of the new exception layout.

Because the algorithm guarantees threads to only read exception positions from their own lane, the scanning loop is no longer necessary. Algorithm 6 shows the improved algorithm that uses the new data-parallel exception layout to decode ALP vectors. The new exception layout is data-parallel because there are no dependencies in the exception patching process of two lanes.

**Algorithm 6** G-ALP single value decoding

*Phase – Initialization*

```
1: PACKED_COUNT_OFFSET ← load(PACKED_COUNTS_OFFSETS[LANE])
2: COUNT ← PACKED_COUNT_OFFSET >> 10
3: OFFSET ← PACKED_COUNT_OFFSET & 0x3FF
4: EXCEPTION_POSITIONS_PTR ← EXCEPTION_POSITIONS_PTR + OFFSET
5: EXCEPTIONS_PTR ← EXCEPTIONS_PTR + OFFSET
6: EC ← 0
```

*Phase – Decode one value*

```
7: DECODED_INTEGER ← FFOR(BIT_PACKED_ARRAY)
8: DECODED_DOUBLE ← ALP(DECODED_INTEGER)
9: WITHIN_BOUNDS ← EC < COUNT
10: IS_EXCEPTION ← CURRENT_POSITION == EXCEPTION_POSITIONS_PTR[EC]
11: if WITHIN_BOUNDS and IS_EXCEPTION then
12:     DECODED_DOUBLE ← EXCEPTIONS_PTR[EC]
13:     EC ← EC + 1
14: end if
15: return DECODED_DOUBLE
```

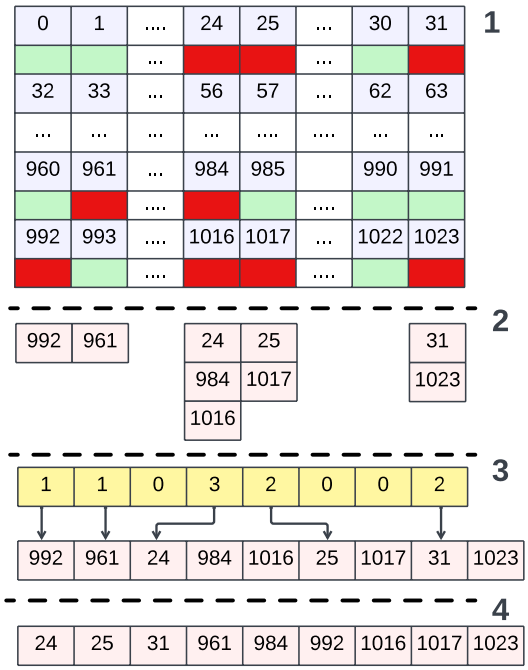


Figure 4.1: Example Layout for Data-Parallel Exception Patching. 1) A vector of 1024 values is represented with two components for each value: the top box represents the position of the value within a vector, ranging from 0 to 1023, while the bottom box is color-coded (red and green) to indicate whether the value is an exception, with red denoting an exception. 2) The second section illustrates the positions of all exceptions per lane. 3) The third section shows the data-parallel storage format for exceptions. Exceptions are sorted by lane, starting with exceptions in lane 0, followed by exception in 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) The fourth section shows the original layout of the same exception list, highlighting the structural differences between the two layouts.

### 4.2.3 Buffering

There is one final optimization made to the ALP decoding process. In Algorithm 6 the arrays with exceptions and exception positions reside in the L1 cache and are accessed every time a value is decoded. The result of the read instruction is needed immediately to verify whether the current position is an exception, and if so, the decoded value needs to be replaced with the exception value. This ensures that the decoding process needs to wait twice for the latency of two read instructions.

Instead, with Algorithm 7 the next exception position and value are buffered in registers. Verifying whether the currently unpacked value is an exception is fast, as the position can be read from memory. If the current value is an exception, the next exception position and value are read into the buffer. However, these two read instructions do not need to complete for the decoding to complete, so the compiler can leverage this instruction-level parallelism and overlap the latencies and continue with other instructions. The results of the read instructions only need to arrive before the next value is patched.

**Algorithm 7** G-ALP single value decoding with buffering

*Phase – Initialization*

```
1: PACKED_COUNT_OFFSET ← load(PACKED_COUNTS_OFFSETS[LANE])
2: COUNT ← PACKED_COUNT_OFFSET >> 10
3: OFFSET ← PACKED_COUNT_OFFSET & 0x3FF
4: EXCEPTION_POSITIONS_PTR ← EXCEPTION_POSITIONS_PTR + OFFSET
5: EXCEPTIONS_PTR ← EXCEPTIONS_PTR + OFFSET
6: BUFFERED_EXCEPTION_POSITION ← EXCEPTION_POSITIONS[0]
7: BUFFERED_EXCEPTION ← EXCEPTIONS[0]
8: EC ← 1
```

*Phase – Decode one value*

```
9: DECODED_INTEGER ← FFOR(BIT_PACKED_ARRAY)
10: DECODED_DOUBLE ← ALP(DECODED_INTEGER)
11: WITHIN_BOUNDS ← EC < COUNT
12: IS_EXCEPTION ← CURRENT_POSITION == BUFFERED_EXCEPTION_POSITION
13: if WITHIN_BOUNDS and IS_EXCEPTION then
14:     DECODED_DOUBLE ← BUFFERED_EXCEPTION
15:     BUFFERED_EXCEPTION_POSITION ← EXCEPTION_POSITIONS_PTR[EC]
16:     BUFFERED_EXCEPTION ← EXCEPTIONS_PTR[EC]
17:     EC ← EC + 1
18: end if
19: return DECODED_DOUBLE
```

# Chapter 5

# Benchmarks

## 5.1 Preliminaries

Before proceeding to the benchmarks, first the benchmarking setup is explained, as well as how the microbenchmark kernel was implemented, and why. Finally an example of the decoder API is presented, to show how the API should be used.

### 5.1.1 Benchmarking Setup

Microbenchmarks are executed on 102400 vectors of synthetic data, to generate enough thread blocks to saturate all SMs. Benchmarks are repeated ten times, and the resulting executing times are averaged with an arithmetic mean. Execution times of microbenchmarks are measured with NVVP, a kernel profiler from NVIDIA. These measurements have a resolution of nanoseconds. All benchmarks were executed on a GTX 1060 GPU, full specifications can be found in Appendix A.

For the comparisons of ALP and G-ALP to nvCOMP compressors on real data, NVIDIA's event based measurement method[33] is used, as nvCOMP compressors commonly launch multiple kernels to decompress data. NVIDIA also benchmarks these compressors themselves using events[36]. These event based measurements have a resolution of one half of a microsecond. A warm-up run of a compressor is executed before executing the actual run that is measured. This is done because the first launch of a kernel can trigger a CUDA 'runtime kernel module loading event' that is then included in the event based measurements.

### 5.1.2 Filter Benchmark

Commonly on CPUs an aggregation would be executed on a compressed column, however on a GPU this requires either significant thread coordination overhead to sum the sums of each thread, or requires overhead from writing to an atomic variable by thousands or millions of threads. This potentially masks the decoding performance by bottlenecking the kernel with coordination overhead.

```cpp
__global__ void scan_column(
                    const ALPColumn<double> column,
                    const double value_to_scan_for,
                    bool *out) {
  const lane_t lane_index = map_thread_to_lane(threadIdx.x);
  const vector_t vector_index = map_warp_to_vector(threadIdx.x);

  double buffer;
  bool not_found_value = true;

  Decoder decoder(column, vector_index, lane_index);

  for (int32_t i = 0; i < column::VALUES_PER_LANE; ++i) {
    decoder.decode_next_value_into(&buffer);

    not_found_value &= buffer != value_to_scan_for;
  }

  if (!not_found_value) {
    // Conditional write to memory;
    // if no value is found, no write is executed
    *out = true;
  }
}
```

Listing 5: This code is a minimal example of how the API could be used in a device kernel in CUDA to implement a scan kernel that isolates decoding throughput performance. Note that the decoding function can decode values into any kind of memory, in this case a single register. Calling the decoding function in this case automatically will calculate the offsets to decode the next value, so repeatedly calling the decoding function will unpack all values in the lane.

Instead the decoding performance is isolated by ensuring that there is as little additional memory read or memory write overhead as possible, and no thread coordination overhead at all. This can be achieved by executing a scan that answers the query 'does value $x$ exist in column $y$?' This is essentially a variant of a filter kernel, but does not return row ids, but simply a boolean answer on whether any value was scanned that was equal to value $x$. An example of how the scan kernel can be implemented is shown in Listing 5. To further limit any memory write throughput during microbenchmarks, it is ensured that only one value or none of the values in a column is equal to $x$, causing at most one warp to execute a write to the GPUs RAM.

### 5.1.3 API

The decoding API takes a column, vector index and a lane index as inputs to decode

a lane of a vector. A vector should be decoded by threads in a single warp, otherwise the compressed vector will not be read using coalesced memory accesses, increasing the memory read throughput bottleneck. This requirement is unavoidable, as reading a normal array also requires that threads read subsequent values in order to achieve an efficient and high memory read throughput. For `uint32_t` vectors, each thread in a warp should decode a different lane in the same vector, as the number of threads is equal to the number of lanes. When decoding a `uint64_t` vector, a single warp can decode two vectors, as there are only 16 lanes. Half the threads decode one vector, and the other half of the threads decode another vector.

In Listing 5 an example is shown how the API can be used. First an object is created that acts as an iterator over the compressed values in a single lane from a vector. With each call to `decode_next_value_into`, the next value in the lane is decoded into a form of memory. This memory can be a register, a local memory array, a shared memory array or even a global memory array.

## 5.2 FFOR Microbenchmarks

In this section the FFOR decoding is benchmarked by decoding 102400 vectors of `uint32_t` or `uint64_t` synthetic data. To benchmark how FFOR decoders perform on vectors decoded with various *value bit widths* (the number of bits a value is packed into), the measurements are repeated for each possible value bit width.

First the previous FFOR implementation from the FastLanesOnGPU paper is benchmarked, to setup a baseline. This FFOR implementation is based on decoding an entire lane with a single switch statement, as shown in Algorithm 1. The next benchmark tests how well switch statement based decoding performs, when decoding a single value at a time. After that alternative approaches to single value decoding as shown in Algorithm 2 are studied. The first alternative approach is a naive implementation of FFOR decoding that calculates all offsets at runtime for each value it decodes, and this approach is referred to as *static* decoding. After that a variant of the decoding API is benchmarked, in which a warp does not decode a single vector at a time, but decodes four vectors concurrently to attempt to expose more ILP for the compiler. The next alternative approach is called *streaming* based decoding, where the offsets of the previous value are reused to calculate the offsets of the next value, speeding up the decoding of consecutive single values. This streaming approach can also store packed integers in a buffer that were loaded but not fully decompressed yet. This buffer can be stored in a variety of memory locations, each of which is benchmarked as well. Next *branchless streaming* is benchmarked, in which the streaming approach is reimplemented to avoid branches, as shown in Algorithm 3. This section finishes off with a multi-column benchmark, in which various decoders are benchmarked when decoding up to ten columns concurrently in parallel. This benchmark investigates whether the alternative approaches to the

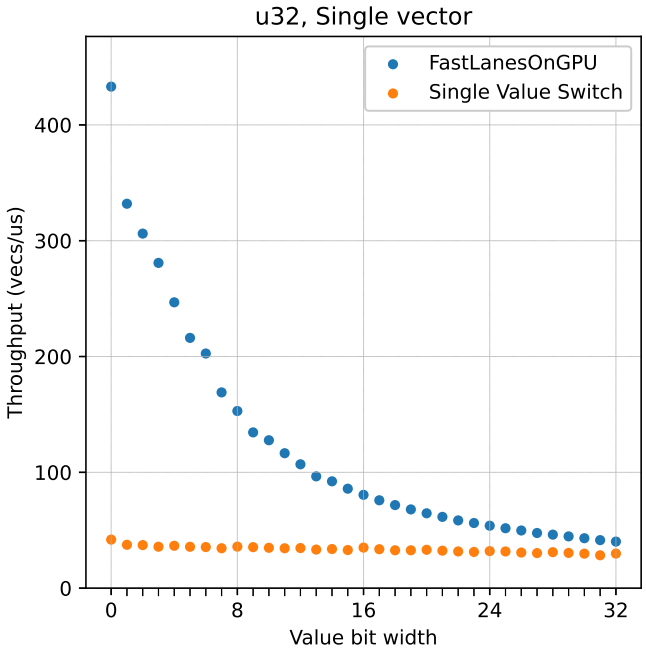


Figure 5.1: This figure shows the decoding throughput of two decoders in the filter benchmark where no decoded data is written back to RAM. The first decoder is the FastLanesOnGPU decoder that uses switch based decoding to decode full lanes, in comparison to a switch based decoder that decodes a single value at a time. The FastLanesOnGPU decoder achieves higher throughput than the single value switch based decoder for all bitwidths. The single value switch based decoder achieves very little throughput for even the lowest bitwidths.

FastLanesOnGPU decoder are indeed better to decode multiple columns concurrently, without suffering from reduced occupancy.

### 5.2.1 Switch Based Decoding

Switch-based decoding can decode entire lanes, as well as single values. The FastLanesOnGPU decoder decodes entire lanes of 32 values, and was only implemented for decoding `uint32_t`. This thesis also implemented switch-based decoding for single value decoding, to investigate whether the switch-based approach performs well when decoding a single value at a time.

The performance of the FastLanesOnGPU decoder and the switch-based single value decoder can be compared in Figure 5.1. The figure shows that the single value decoding performance is suboptimal. As described in Subsection 4.1.1, most of the benefits of switch case based decoding cannot be leveraged when decoding a single value at a time. The single value switch based decoder does not even reach the same throughput as the FastLanesOnGPU decoder for the largest value bit widths, when memory read throughput is the main bottleneck.

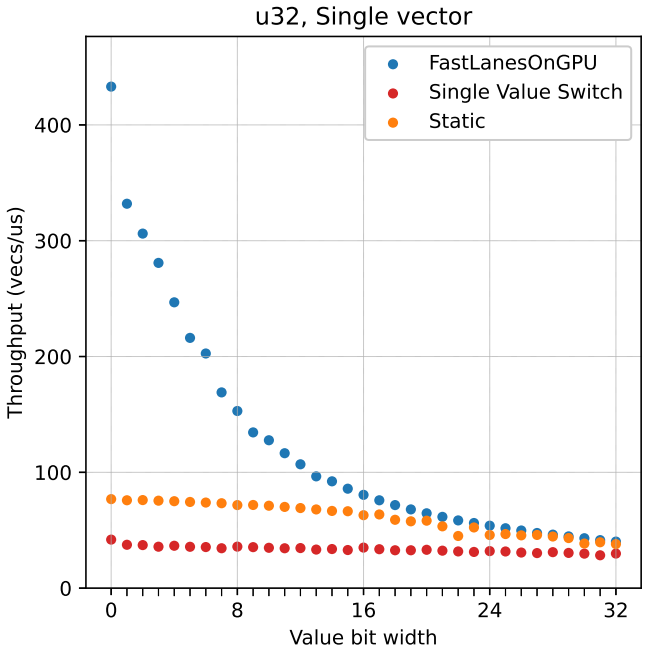


Figure 5.2: This figure shows the decoding throughput performance of the static single value (`uint32_t`) at a time decoder in comparison to the previous decoders. This new decoder achieves twice as much decoding throughput as the single value switch based decoder for lower bitwidths, and almost as much throughput as the FastLanesOnGPU decoder for the higher bitwidths.

### 5.2.2 Static Decoding

As switch based decoding is not suitable for single value decoding, a naive FFOR decoder is implemented. The pseudocode for this decoder can be found in Algorithm 2. The decoder repeatedly calculates the required offsets at runtime for each value individually, and branches are used to read the required compressed values from memory. This naive decoder will be referred to as the static decoder as there is no benefit in reading consecutive values from the same lane, as the required offsets are recalculated for each value, and no packed integers are buffered.

Figure 5.2 shows that the static decoder outperforms the switch based single value decoder for all bitwidths. However, the FastLanesOnGPU decoder achieves far higher throughput for 16 and lower value bit widths. The same data is plotted in Figure 5.3, but then with the execution time instead of the throughput on the y-axis. This figure shows that for the higher value bit widths, the execution time of the FastLanesOnGPU decoder and static decoder is similar, as both are bottlenecked by the read throughput. However, the static decoder does not reach the same excution times as the FastLanesOnGPU decoder for lower value bit widths, as the static decoder is bound by memory latency for those value bit widths.

The static decoder can also decode `uint64_t` data. The decoding performance can

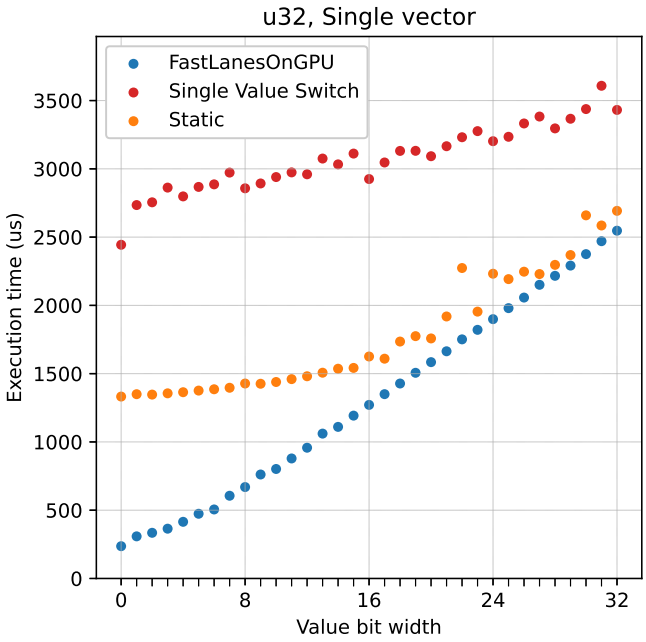


Figure 5.3: This figure shows the same data as Figure 5.2, but now plots the execution time on the y-axis instead of the decoding throughput. This figure shows that the FastLanesOnGPU decoder is bound by read throughput for almost all bitwidths, as execution time increases linearly, proportional to amount of extra data loaded. The single value static decoder matches the execution time of the FastLanesOnGPU decoder for higher bitwidths, but the execution times stop decreasing at around bitwidth 16, indicating that the decoder is running into a non-memory throughput bottleneck instead.

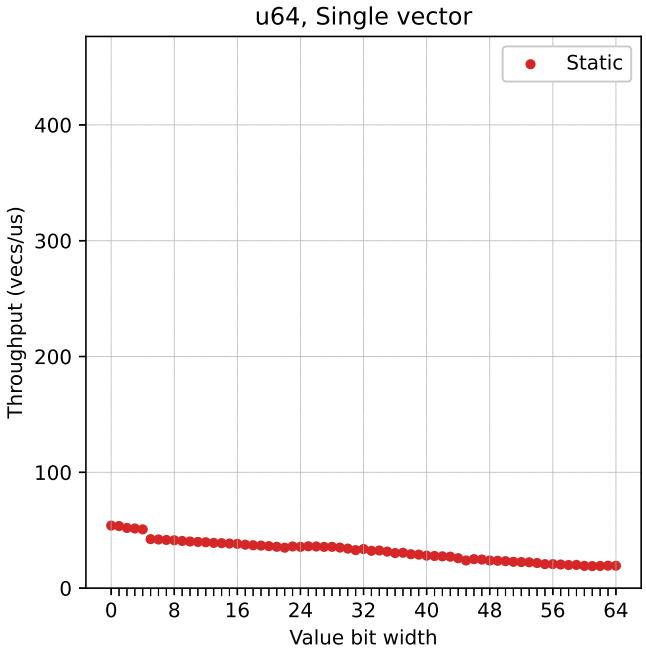


Figure 5.4: Decoding performance of the static decoder when decoding a `uint64_t` column. The decoding throughput is lower for all bitwidths in comparison to decoding a `uint32_t` with the same decoder, this is due to lower long integer instructions throughput.

be found in Figure 5.4. The performance pattern is very similar for most value bit widths, but achieves lower throughput overall, even for value bit widths in common with `uint32_t`. This can be due to a number of reasons, such as read instructions that need to load two 128-byte cache lines to full fill reads due to the larger data type, even if the reads are fully coalesced, or due to the reduced number of 64 bit integer instruction pipelines.

### 5.2.3 Multi-Vector Decoding

For lower value bit widths, the long memory latency is the bottleneck, the warp is mostly waiting for reads to complete. This latency can be mitigated by overlapping the latencies of multiple instructions by exposing more ILP to the compiler. If four consecutive vectors are packed with the same value bit width, the decompression instructions for those vectors would be identical. In that case, instead of only decompressing a single vector per warp, a warp could decompress four vectors concurrently. Whenever the warp needs to read a new line from the first packed vector, it can also issue read instructions for the other three vectors, as the read instructions are independent.

When disassembling a kernel that decodes a single value, the SASS will contain isolated read instructions, followed by unpacking instructions that need to wait for the read instruction to complete, as shown in Listing 6. However, when decompressing

```
...
LDG.E.CLI. R1 [R2]; // LDG -> Load from Global
// Decoding instructions for R1
...
```

Listing 6: SASS Assembly for unpackers that decompress a single vector.

```
...
LDG.E.CLI. R1 [R2]; // LDG -> Load from Global
LDG.E.CLI. R3 [R4];
LDG.E.CLI. R5 [R6];
LDG.E.CLI. R7 [R8];
// Decoding instructions for R1, R3, R5, R7
...
```

Listing 7: SASS Assembly for unpackers that decompress multiple vectors concurrently. The actual assembly is sometimes more convoluted, as the compiler might interleave other instructions in between the loads. However the load instructions will be found near eachother, with the bulk of the unpacking instructions found after the last load instruction.

multiple vectors concurrently, the assembly of the disassembled kernel indeed shows that the read instructions are grouped together by the compiler, as shown in Listing 7.

Figure 5.5 shows the performance of the static decoder for decompressing multiple vectors concurrently. Multi-vector decompression has much higher decoding throughput, achieving up to twice the throughput of the single vector static decoder for low value bit widths, for both `uint32_t` and `uint64_t`. The graph also confirms that decoding vectors with high value bit widths are bound by memory throughput, as the decoding throughput is only increased for the lower value bit widths which are latency bound.

### 5.2.4 Streaming Decoding

With the static decoding approach, the offsets of the packed value in compressed vector is recalculated completely when decoding each value. The static decoder also needs to load the compressed integers from global memory each time it decodes a value, and relies on the fact that previously accessed integers are still stored in the L1 cache. However, a *streaming* approach could be faster. The streaming approach, reuses the offsets of the previous value to calculate the next offset, and keeps previously loaded compressed integers in buffers until they are fully decompressed. This streaming manner requires the user of the decoder to sequentially decode values, but it is cheaper to decode every subsequent value.

There are different techniques for buffering compressed data that is being decompressed, until all values that are packed into those integers are decoded via the single

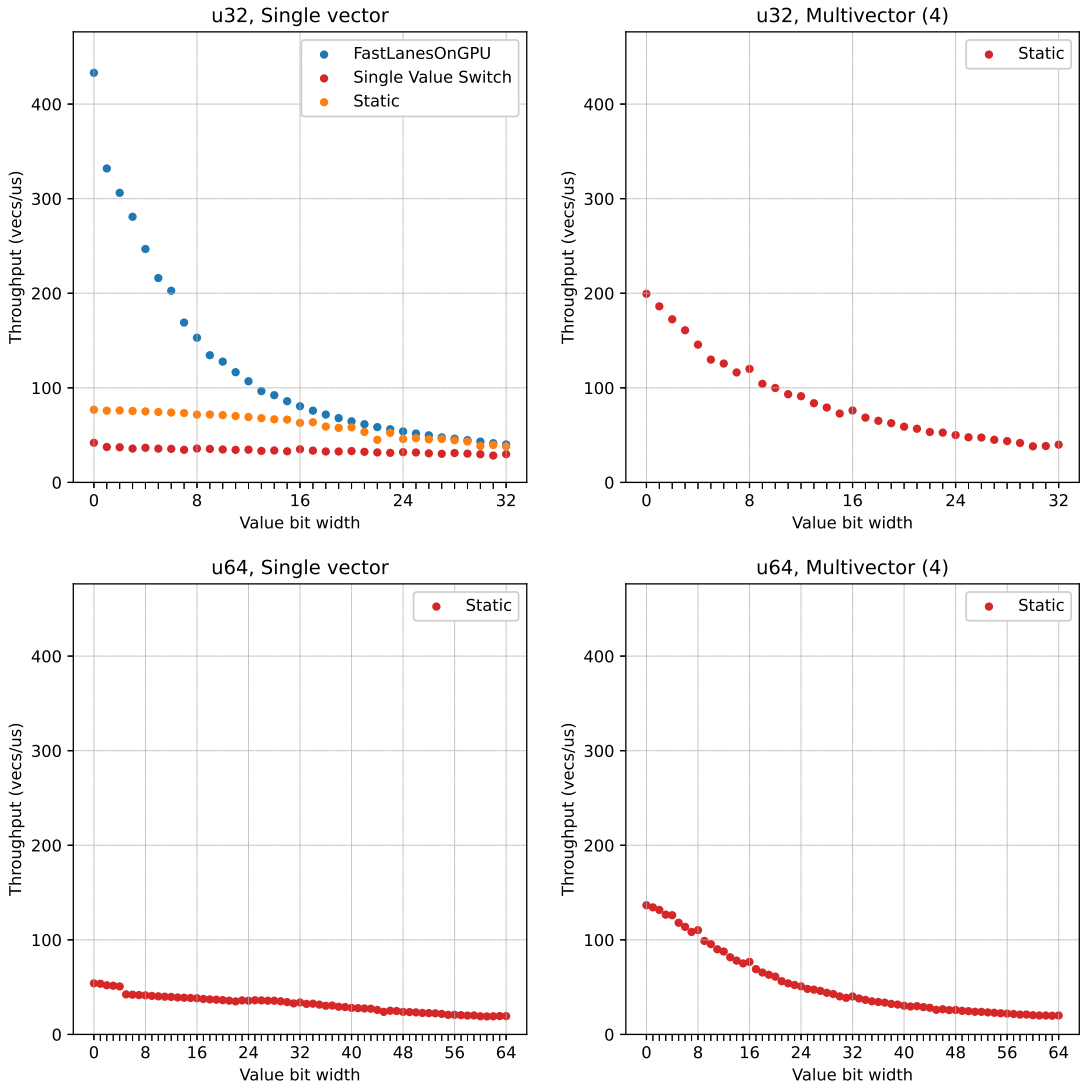


Figure 5.5: This figure shows the previous two decoding throughput graphs that decode a single vector at a time on the left. The right graphs show the decoding throughput of the static decoder when decoding four vectors concurrently. Decoding four FFOR vectors concurrently results in large increases in decoding throughput, approaching the decoding throughput of the single vector FastLanesOnGPU decoder.

value at a time API. You can even buffer multiple compressed integers before fully decoding them. In this case, one would create a buffer and directly load the first $n$ compressed integers. When all compressed integers in the buffer are fully decoded, the buffer is refreshed by loading the next $n$ compressed integers into the buffer. The benefit of this approach would be to overlap the read latencies of the instructions that load compressed integers into the buffer. The downside is that these compressed integers are loaded unconditionally, so if the buffer size is four, and the value bit width is three, four compressed integers are loaded anyway.

As a baseline for the streaming decoder buffer strategies, it is possible to not use a buffer and only rely on reusing the offsets of the previous value to improve upon the static decoder, in this case one would rely on the automatic L1 cache behaviour to quickly read recently accessed memory.

The buffers could be stored in local memory, shared memory or registers. The buffer can be stored in local memory by creating a normal array and dynamically accessing that array, based on the required buffer index. The buffer can also be stored in shared memory by prepending the array declaration with the `__shared__` keyword. However, the shared memory array is declared per thread block, so in the API the user would also need to provive the blocksize at compile time.

The buffers can also be stored in registers, but this requires that the dynamic access based on buffer index is eliminated. This can be done using a switch statement, as shown in Listing 8. This avoids dynamic access into an array by using the switch statement to jump to the correct load instruction. The switch statement can also be avoided by leveraging the fact that the second buffer is never accessed before the first buffer is decoded, and the third never before the second, etc. In this case we can *rotate* the registers, so when the first buffer is fully decoded, the contents of the second buffer are loaded into the first, and the contents of the third into the second, etc., this is shown in Listing 9.

```
__device__ __forceinline__ void load_next_buffer(
                                    const uint32_t buffer_index,
                                    UINT_T *out) {
  switch (buffer_index) {
    case 0:
      *out = buffer[0];
      break;
    case 1:
      *out = buffer[1];
      break;
    case 2:
      *out = buffer[2];
      break;
    case 3:
      *out = buffer[3];
      break;
  }
}
```

Listing 8: Retrieving the right buffer without dynamic access by relying on branching.

```
__device__ __forceinline__ void load_next_buffer(
                                    const uint32_t buffer_index,
                                    UINT_T *out) {
  out* = buffer[0];

  // Rotating the buffers
#pragma unroll
  for (int b{1}; b < BUFFER_SIZE; ++b) {
    buffer[b - 1] = buffer[b];
  }
}
```

Listing 9: Retrieving the right buffer without dynamic access by rotating the registers. The buffer index can be ignored, as the load pattern of the buffers is sequential.

The decoding throughput for all buffer variants is shown in Figure 5.6, for buffer sizes one, two and four, and with single vector decompression and multivector decompression. Local memory and shared memory have the lowest throughput in every configuration. This is not surprising, as loading data into shared or local memory requires data to first be loaded into registers from global memory, and then be moved into local or shared memory, which is an extra step. The register based decoders are much faster, with the register rotation based decoder narrowly beating the switch based decoder in all cases. The streaming decoders reached the highest decoding throughput when using a buffer of

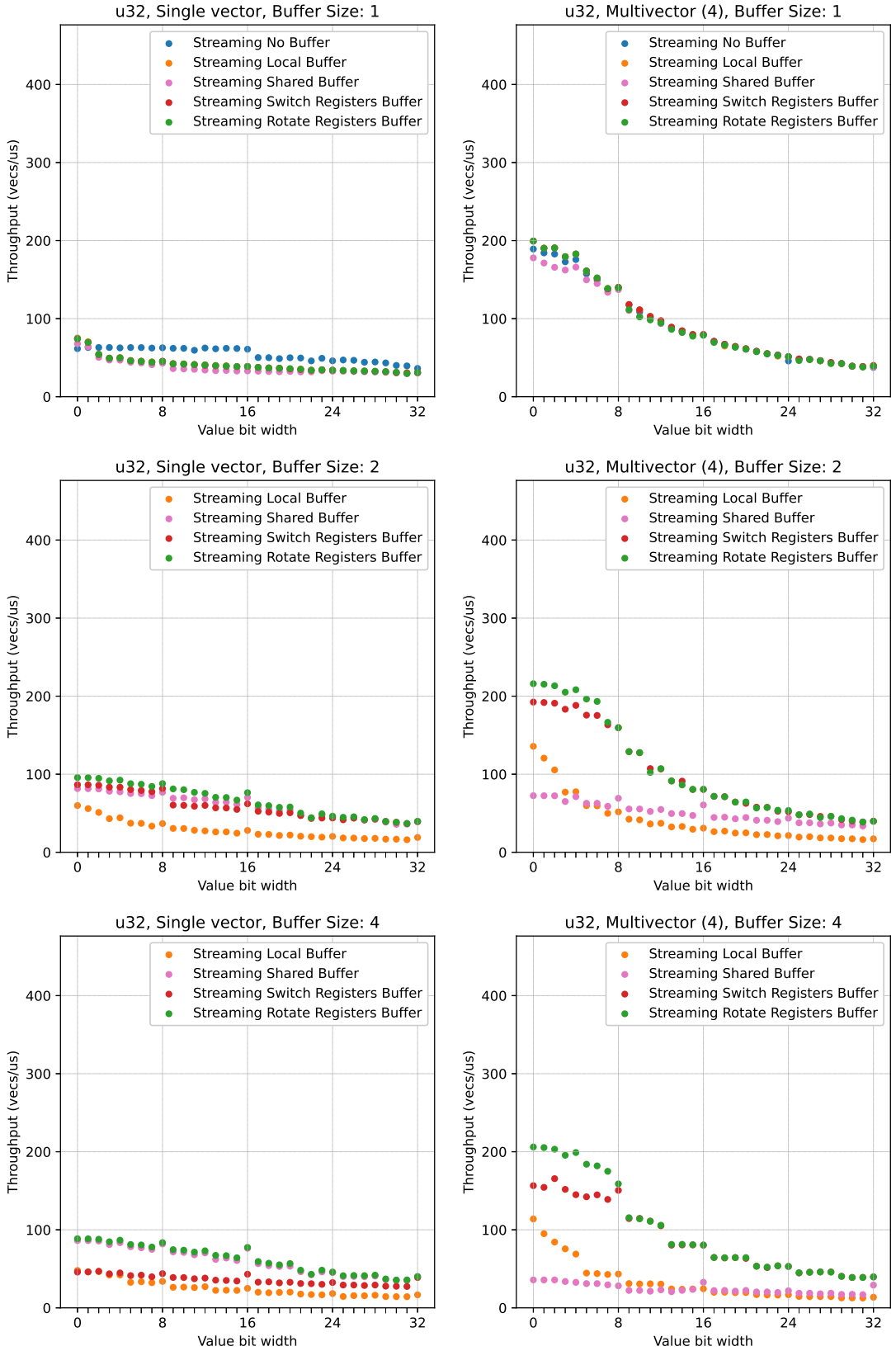


Figure 5.6: Streaming buffer strategy throughput comparison for `uint32_t` columns. On the left side the decoding throughput for single vector decoders are plotted, the right side shows the decoding throughput of decoding four vectors concurrently. The decoders in the top row graphs have a buffer size of one, the middle row decoders have a buffer size of two, and the bottom row decoders have a buffer size of four. A buffer size of two is best for all buffering strategies. The best buffer type is the register buffers, the other buffer types have much lower decoding throughput.

two compressed integers. For single vector decompression the top throughput is almost 100 vectors per microsecond, and multivector decompression decodes over 200 vectors per microsecond.

The streaming decoders therefore also beat the static decoder, which had a top decompression throughput of up to 80 vectors per microsecond for single vector decompression, and reaching only 200 vectors per microsecond in multivector decompression for value bit width 0.

In figure Figure 5.7 the decoding throughput for `uint64_t` decompression is shown. The figure shows that the performance pattern is the same as when decoding `uint32_t`, with the register rotating decoder again achieving the highest throughput and beating the static decoder. However, as with the static decoder, the `uint64_t` decoding throughput is lower than the `uint32_t` throughput, even when vectors are compressed with the same value bit widths, showing that the reduced number of 64 bit integer instruction pipelines causes the lower decoding throughput.

### 5.2.5 Branchless Decoding

The streaming decoder can also be implemented completely branchlessly, removing all control hazards, which may allow the compiler to generate more efficient assembly by reordering instructions, and warps need to stall less often. The pseudocode of the branchless decoder was shown in Algorithm 3, and cannot buffer values, so it relies on the L1 cache for fast access to previously accessed compressed integers.

In Figure 5.8 the throughput of all decoding approaches are shown. The branchless streaming decoder is slightly slower than the previous streaming decoder when decompressing a single vector. The branchless streaming decoder benefits less from decoding multiple vectors concurrently in comparison to single vector decompression. For single column decoding the branchless streaming decoder does not improve upon previous decoders.

### 5.2.6 Full Decompression

All previous graphs showed the performance of the decoders when only decoding and scanning the compressed data. In Figure 5.9 the full decompression performance of the decoders is shown. With full decompression, the column is completely materialized and written back to global memory. The figure shows that with full decompression all decoders hit the same maximum throughput, as they are completely bottlenecked by memory write throughput of writing decoded data to global memory. Efficiently decoding data now matters less, as the amount of data written is constant and greater than the amount of data read.

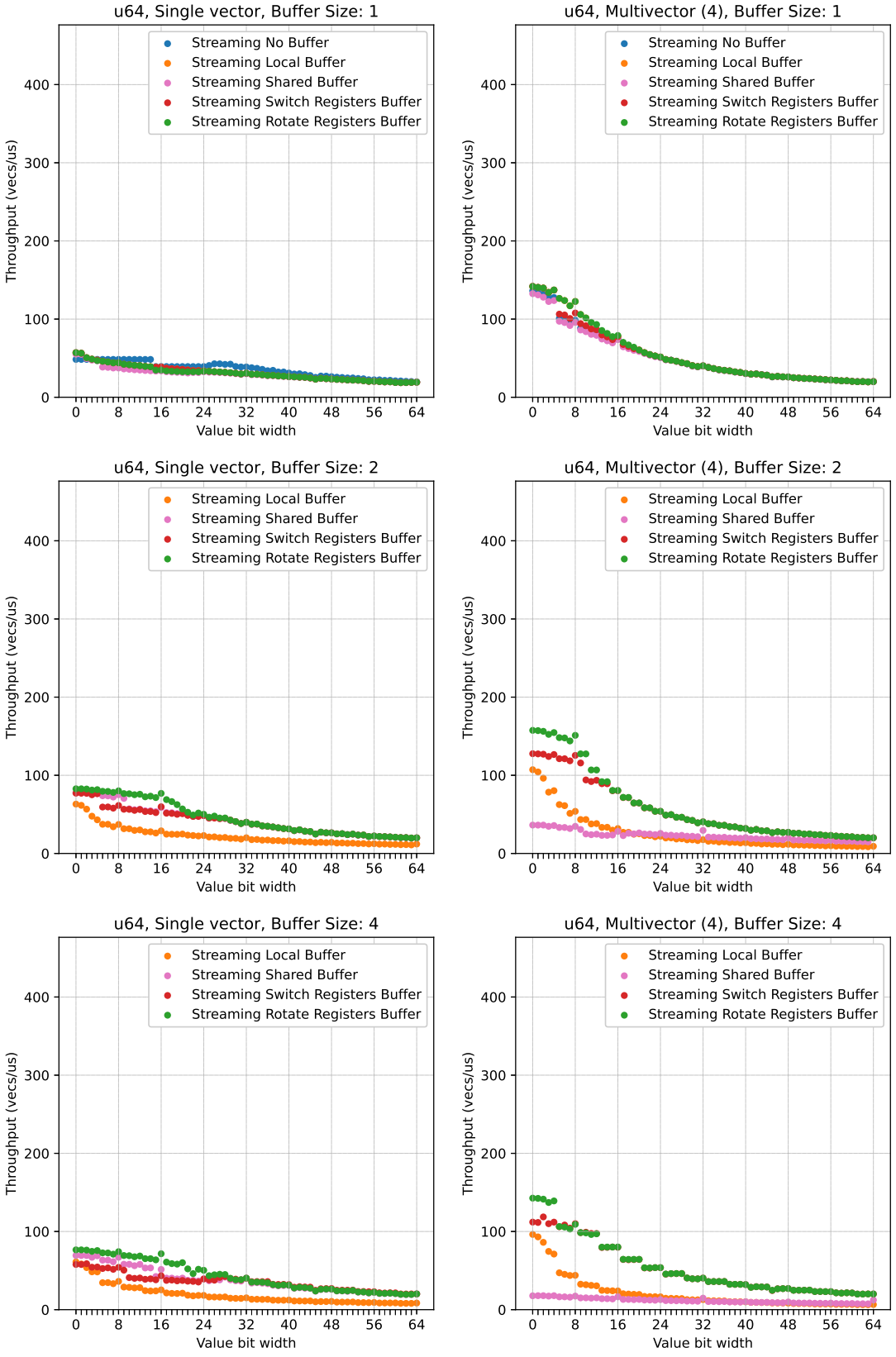


Figure 5.7: Streaming buffer strategy throughput comparison for `uint64_t` columns. On the left side the decoding throughput for single vector decoders are plotted, the right side shows the decoding throughput of decoding four vectors concurrently. As for `uint32_t` columns, a buffer size of two is best, but the differences in decoding throughput between buffer strategies are smaller, as peak decoding throughput is lower than for `uint32_t` columns.

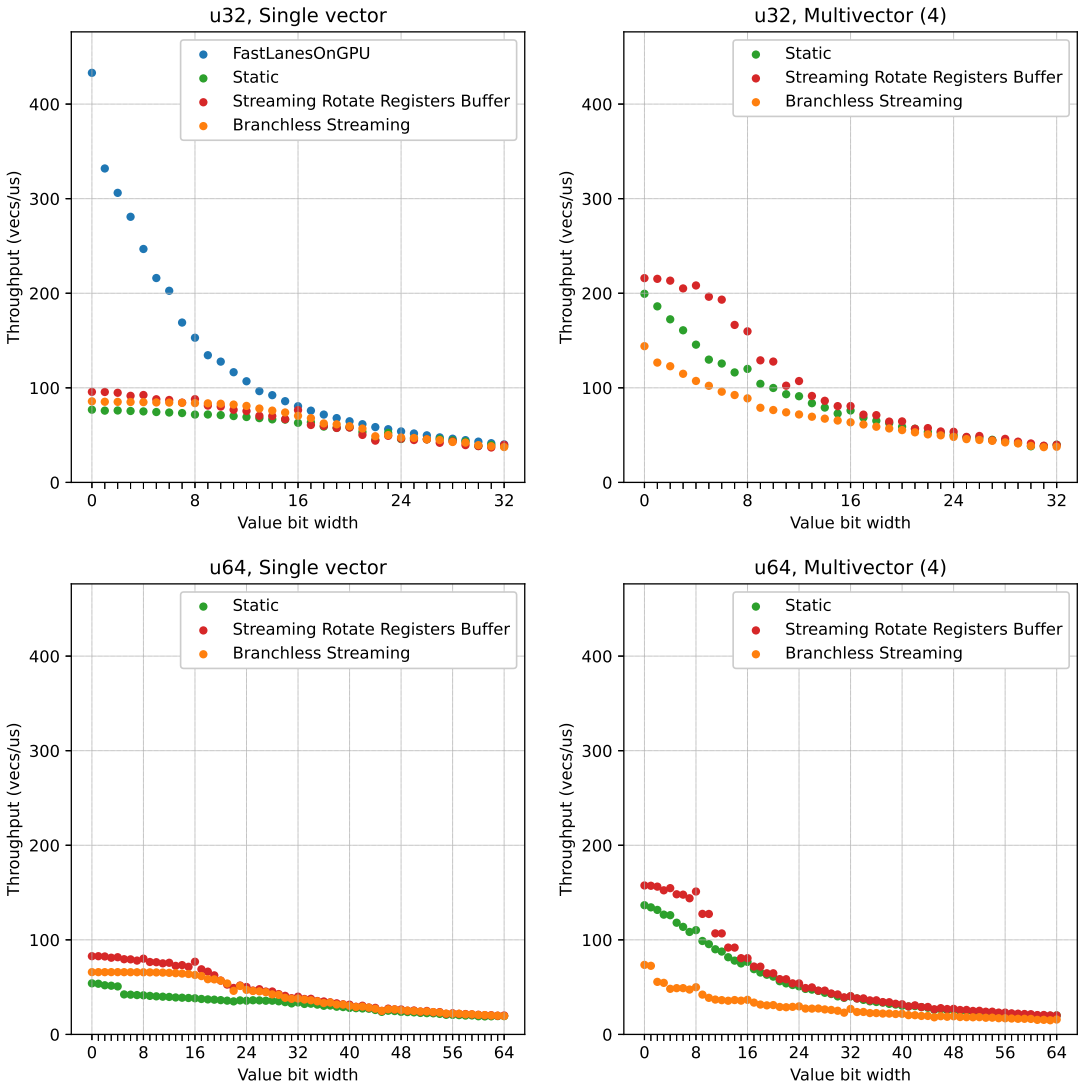


Figure 5.8: Comparison of decoding throughput of the branchless streaming decoder with the FastLanesOnGPU decoder, the static decoder, and the best streaming decoder with buffer. The branchless streaming decoder is close in performance to the buffering decoder for single vector decoding, but benefits less from decoding multiple vectors concurrently.

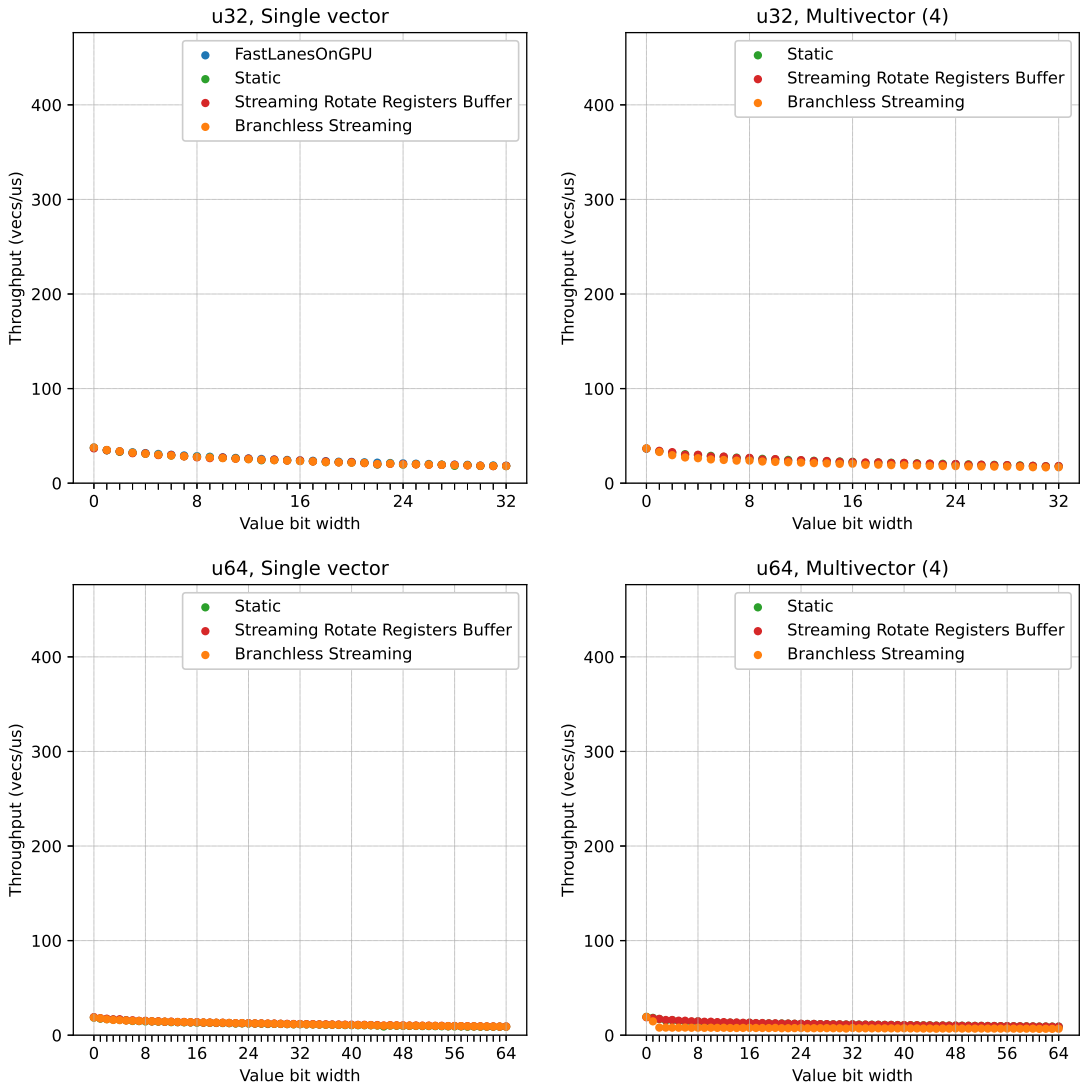


Figure 5.9: Comparison of full decompression into RAM throughput of the FastLanesOnGPU decoder, the static decoder, the best streaming decoder with buffer, and the branchless streaming decoder. For full decompression into RAM the decoding performance is bottlenecked by memory write throughput. This masks the efficiency of the decoding approach, so for full decompression of data into RAM the decoding approach does not matter.

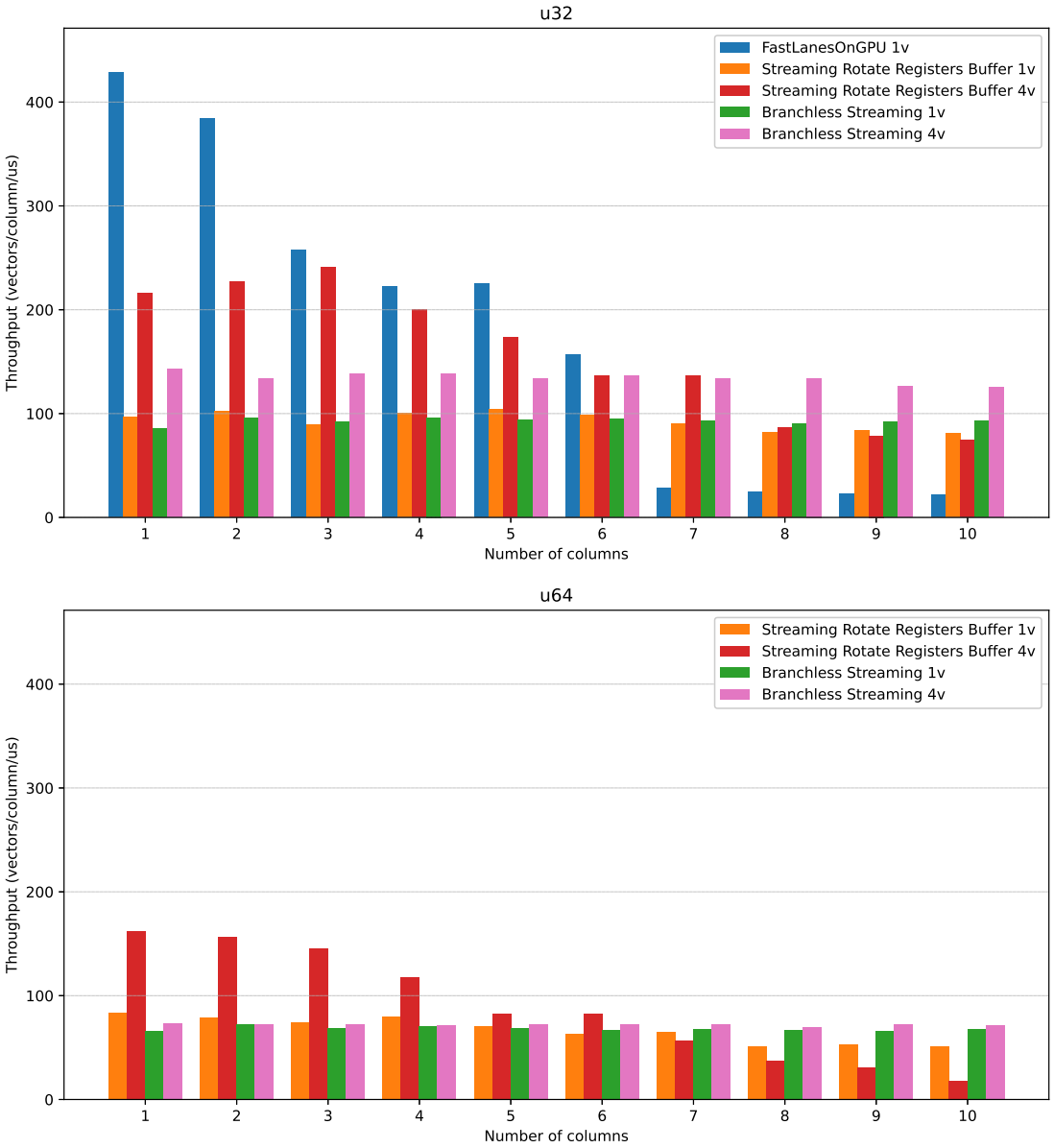


Figure 5.10: Comparison of FFOR decoding throughput when decoding multiple columns concurrently. Note that the throughput is in vectors/columns/us, correcting for the fact that with ten columns, ten times more data is loaded. Each vector in the column is encoded with a random bitwidth between 0 and the maximum bitwidth for that datatype. The FastLanesOnGPU decoder achieves high decoding throughput when decoding a single column, but performance quickly deteriorates when decoding multiple columns concurrently. The streaming decoder with buffer performs well for low number of columns, but throughput deteriorates for higher number of columns. The branchless streaming decoder achieves a very stable, consistent throughput, for any number of columns. Decompressing multiple vectors concurrently does not result in much higher throughput for larger numbers for columns.

### 5.2.7 Multi-Column

As the FastLanesOnGPU decoder's performances tended to degrade when multiple columns are scanned, the last microbenchmark for FFOR studies the performance of the decoders when decoding up to ten columns concurrently. For the FastLanesOnGPU decoder each warp fully decompresses a vector from each column before comparing the values, whereas with the other decoders each warp decodes a single value from each column, before comparing the values and scanning the next value in the lane. In Figure 5.10 the performance of the FastLanesOnGPU decoder, the static decoder, the streaming decoder and the branchless streaming decoder are shown.

The FastLanesOnGPU decoder is the fastest decoder for FFOR up until decompressing 7 columns. The bottleneck for FastLanesOnGPU is memory latency, as the memory read throughput is not high and neither is the instruction throughput. The other decoders suffer less from this latency problem as the occupancy is higher, therefore allowing the GPU to hide the latency of the read instructions better. For single vector decoding, the throughput of the streaming decoder and the branchless streaming decoder is similar. The streaming decoder achieves slightly higher throughput for smaller numbers of columns, while the branchless streaming decoder achieves higher throughput for large numbers of columns. Multivector decoding achieves higher throughput than single vector decoding. The throughput of the branchless streaming decoder is consistent, achieving the same throughput per vector independent of how many columns are decoded. The branchy streaming decoder slightly suffers in terms of throughput when decoding more columns.

The throughput of multi-column `uint64_t` decoding is again slower than decoding `uint32_t` columns. The throughput of the streaming decoder suffers more from decoding multiple columns than the throughput of the branchless streaming decoder. The benefit of multivector decoding is less pronounced, with the branchless streaming decoder now almost not benefitting at all from decoding multiple vectors concurrently.

### 5.2.8 Summary

The alternative decoding approaches to FastLanesOnGPU decoding did not achieve higher decoding throughput than the original decoder in single vector decoding. Static decoding performed worst. Streaming decoding is faster, especially when keeping multiple packed integers in buffers. Branchless streaming decoding is slightly slower than normal streaming decoding in single column decoding, but performs slightly better, and more consistent, when decoding values from multiple columns concurrently. Decoding multiple vectors is beneficial for FFOR, as multivector decoding can help hide the high memory read latency. Decoding `uint64_t` columns is slower than decoding `uint32_t` columns. Fully decompressing columns is bottlenecked by memory write throughput, causing all decoders to achieve the same decompression throughput.

## 5.3 ALP Microbenchmarks

In this section ALP decoding is benchmarked. ALP decoding builds upon FFOR decoding. After FFOR decoding a value, ALP applies some casts and multiplications to map the decoded integer into a floating point datatype. As the FFOR step is the same, we can reuse the branchless streaming decoder for this stage of ALP. The branchless streaming decoder was chosen as it offers constant and consistent decoding performance. In the final multi-column benchmark the FastLanesOnGPU and normal streaming decoder are also tested in combination with ALP. After the FFOR decoding, values that were actually exceptions need to be patched with their exception value. This exception patching will be the primary subject of this section, and in the microbenchmarks in this section the number of exceptions per vector will be varied to study the performance of the exception patchers. The exceptions are uniformly and randomly distributed within the vector. The value bit width is random per vector, and in the range of $[3 - 8]$.

### 5.3.1 ALP

The first microbenchmark compares the performance of static ALP patching (Algorithm 4) with streaming ALP patching (Algorithm 5). To avoid confusion with streaming FFOR decoding, streaming ALP patching will be referred to as just ALP patching. Static ALP patching needs to scan the exception positions vector from the start for each value that is decoded. The (streaming) ALP decoder however can benefit from the fact that for patching the previous value, part of the exception positions vector is already scanned, and cannot contain the position of the current value, saving a costly rescan.

In Figure 5.11 the performance of both patchers is shown. (Streaming) ALP patching is much faster than static ALP patching, having more than twice the throughput for most exception counts. However, single vector FFOR decoding reaches a decoding throughput of 100 vectors per microsecond, but ALP achieves only 20 vectors per microsecond for most exception counts. Patching exceptions is an expensive additional step that requires more instructions when decoding a single value at a time, as you need to check for each value whether the position is in the exception list. When decoding an entire vector on the CPU, one can simply replace values in the decoded data array, requiring fewer instructions. Also, neither of the patchers benefit much from multi-vector decoding. Multi-vector decoding even has lower throughput when patching more than ten exceptions, or when patching `double` exceptions. Interestingly, patching `float` exceptions is almost as slow as patching `double` exceptions.

### 5.3.2 G-ALP

In Subsection 4.2.2 a new exception layout is proposed, that can be processed with a much simpler exception patching algorithm as, shown in Algorithm 6. In Figure 5.12 the performance of two variants of this algorithm are shown. The first variant is branchy,

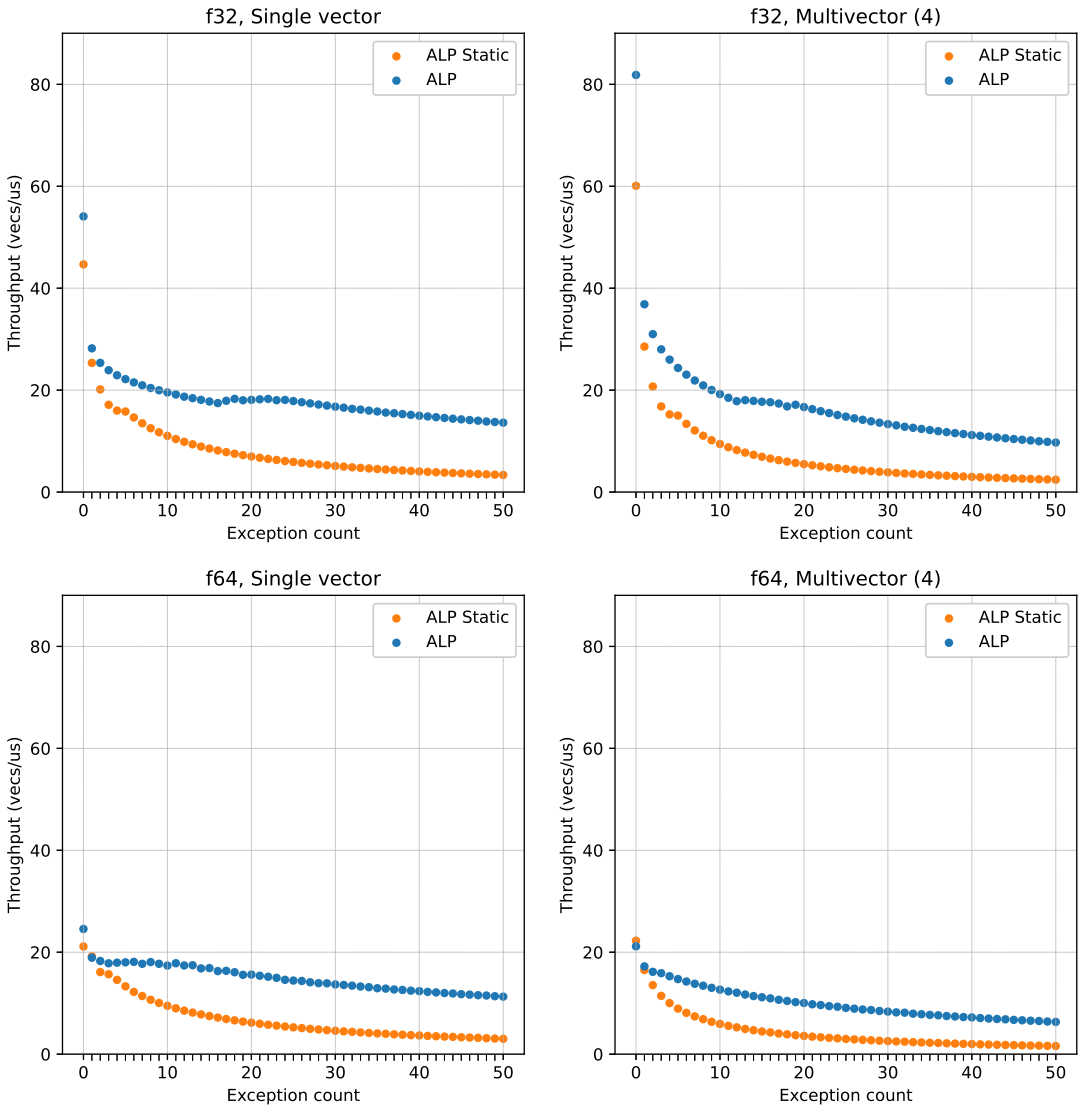


Figure 5.11: Decoding throughput of the static ALP patcher and the ALP patcher, both use the branchless streaming decoder for decoding the bitpacked integers. The bitwidth for each vector is random between 3 and 8. The x-axis shows the number of exceptions per vectors, showing the decoding throughput of each patching approach for a various number of exceptions per vector. Decoding multiple vectors concurrently now results in very little increased throughput, for some numbers of exceptions even resulting in less throughput. The ALP patcher is better than the static ALP patcher for any number of exceptions per vector.

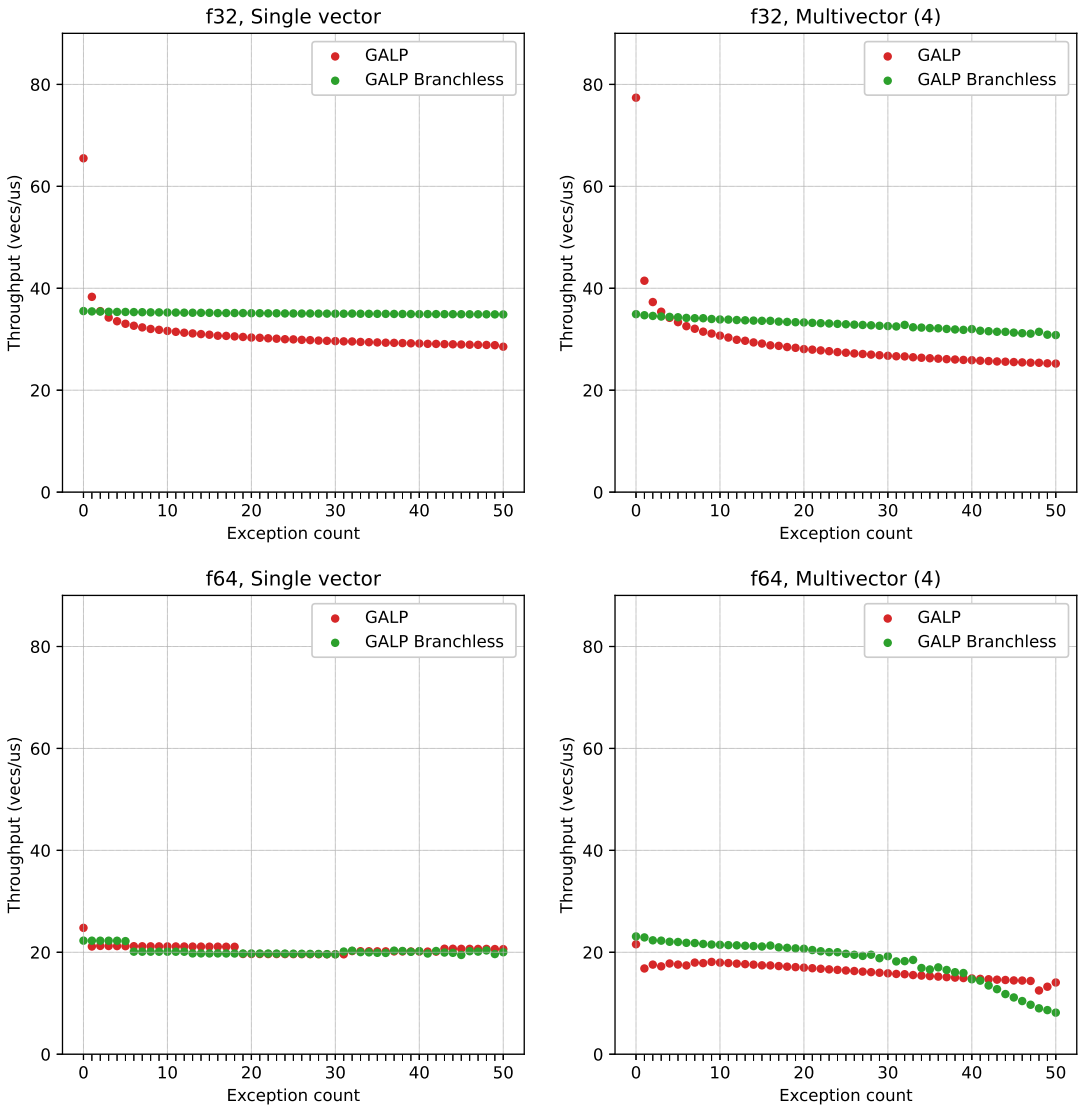


Figure 5.12: Decoding throughput of the G-ALP patcher and the branchless G-ALP patcher, both use the branchless streaming decoder for decoding the bitpacked integers. The bitwidth for each vector is random between 3 and 8. Branchless G-ALP achieves a higher, and more constant decoding throughput than the G-ALP patcher. The decoding throughput of the G-ALP patchers is more than twice as much as the throughput of the ALP patchers for almost all numbers of exceptions per vector. Decoding multiple vectors concurrently results in slightly lower decoding throughput.

the second is branchless. The branchless variant is more consistent and has slightly higher throughput than the branchy variant, except for the smallest exception counts. The throughput of the `float` G-ALP patchers is almost twice as high as that of the ALP patchers, decoding almost 40 vectors per microsecond. However, for `double` the decoding throughput is the same, at 20 vectors per microsecond. In comparison to the ALP patchers, the G-ALP patchers are less sensitive to an increasing exception count, still achieving high throughput even for vectors that contain 50 exceptions. The G-ALP patchers are less sensitive than ALP patchers, as there is less warp divergence when a vector contains more exceptions. With G-ALP the maximum amount of exception positions that a thread needs to scan per value it decodes is 1. With ALP the maximum amount of exception positions that a thread needs to scan when decoding a single value is equal to the number of lanes, when the thread is assigned the last lane and all lanes contain an exception.

### 5.3.3 G-ALP With Buffering

In Subsection 4.2.3 an optimization of G-ALP was discussed, in which the next exception position and exception value is buffered in the registers to reduce the memory read latency of fetching those values from cache. In Figure 5.13 the throughput of the G-ALP patchers with this optimization is shown. The throughput of the branchless G-ALP patcher barely changes. However, the branchy variant now beats the branchless variant, achieving a throughput of close to 50 vectors per microsecond. For multivector decompression, the throughput is even higher, reaching almost 60 vectors per microsecond. These are significant improvements upon the G-ALP patchers in the previous section, achieving almost 50% more throughput by buffering the next exception position and exception value in registers.

The throughput of patching `double` does not change however, and peaks at patching 20 vectors per microsecond, just as the previous patchers.

### 5.3.4 Multi-Column

As for the FFOR decoders, the ALP patchers are also evaluated in a multi-column decoding benchmark. Figure 5.14 and Figure 5.15 show the decoding performance for all patchers, and for the three best decoders; the FastLanesOnGPU decoder, the streaming decoder that rotates registers as a buffer strategy, and the branchless streaming decoder. Each column is packed with a random bit width in the range $[3, 8]$ and a constant exception count of 20.

The decoding throughput of the patchers in combination with FastLanesOnGPU is good for decoding single columns. However, the throughput is much worse than the other decoders when decoding two columns or more, achieving just half the throughput of the other decoders for high numbers of columns. The streaming decoder is better

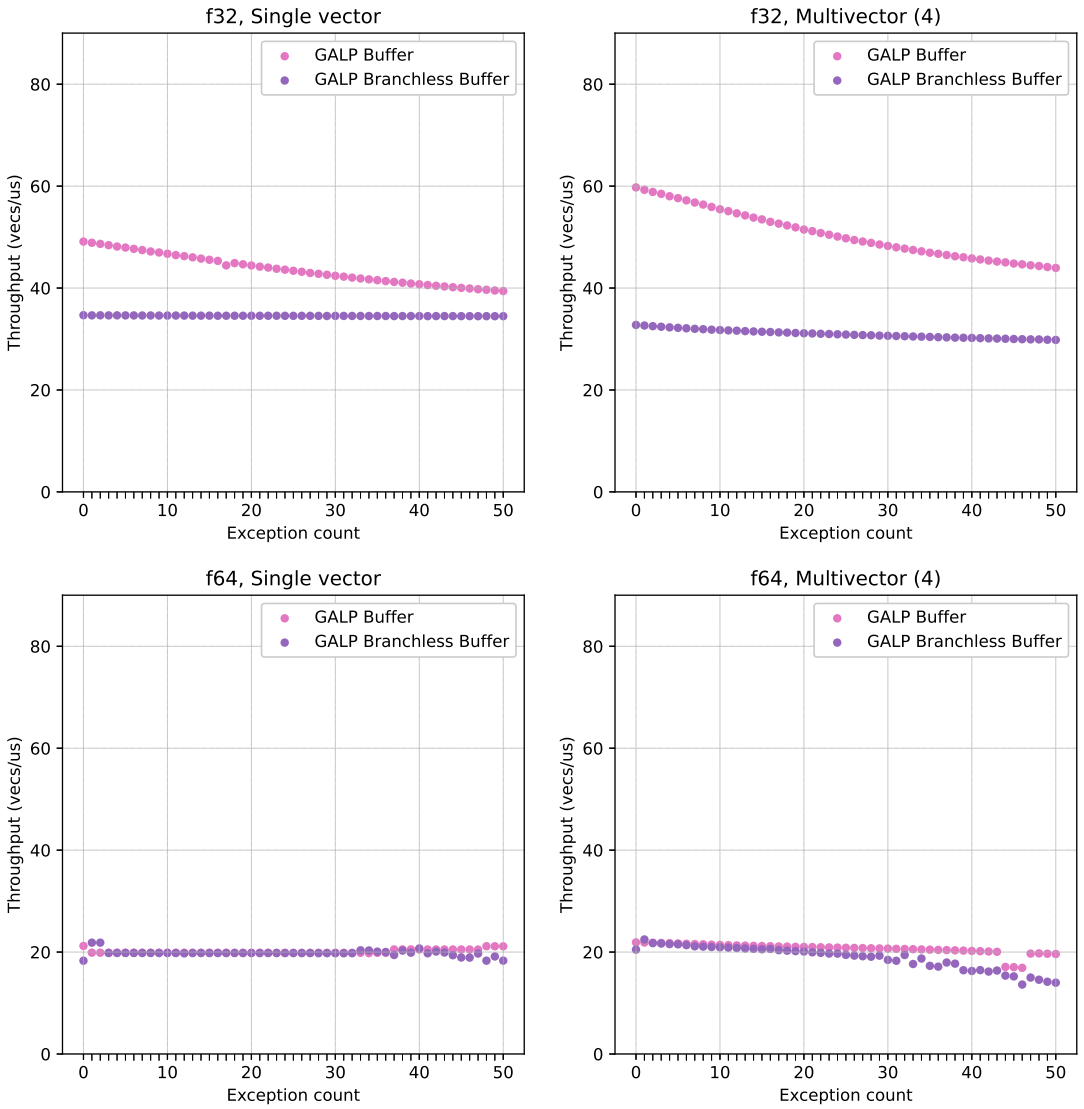


Figure 5.13: Decoding throughput of the G-ALP patcher with exception buffer and the branchless G-ALP patcher with exception buffer, both use the branchless streaming decoder for decoding the bitpacked integers. The bitwidth for each vector is random between 3 and 8. With the exception buffer the throughput of the branchless G-ALP patcher is not improved. For the normal G-ALP patcher however, the throughput is improved for all exception counts, especially the lower exception counts. For the normal G-ALP patcher decoding multiple vectors concurrently now also improves throughput again. The decoding throughput of the `double` vectors is not increased, the decoding throughput of those vectors is bottlenecked at around 20 vectors/$\mu$s, due to the limited `double` instructions throughput.

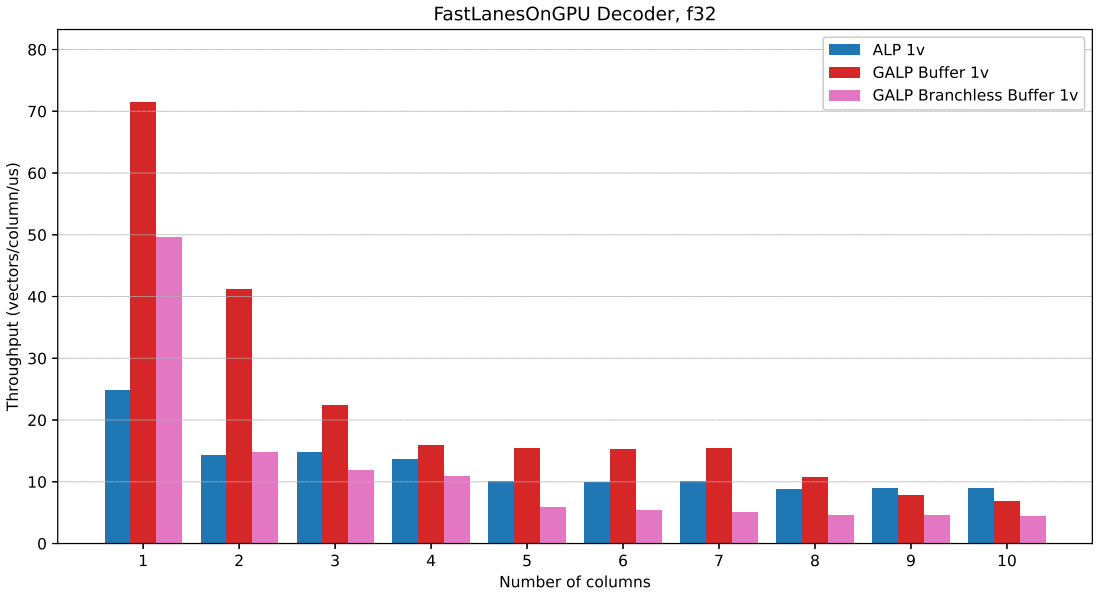


Figure 5.14: Comparison of the decoding throughput for multiple ALP patchers for `float` columns when decoding multiple columns concurrently. All patchers use the FastLanesOnGPU decoder for decoding the bitpacked integers. The bitwidth is random per vector, between 3 and 8, and the number of exceptions for each vector is constant at 20 exceptions per vector. The decoding throughput heavily degrades for all patchers when decoding multiple columns concurrently, mainly due to the fact that the decoding throughput of the underlying integer decoder degrades when decoding multiple columns concurrently.

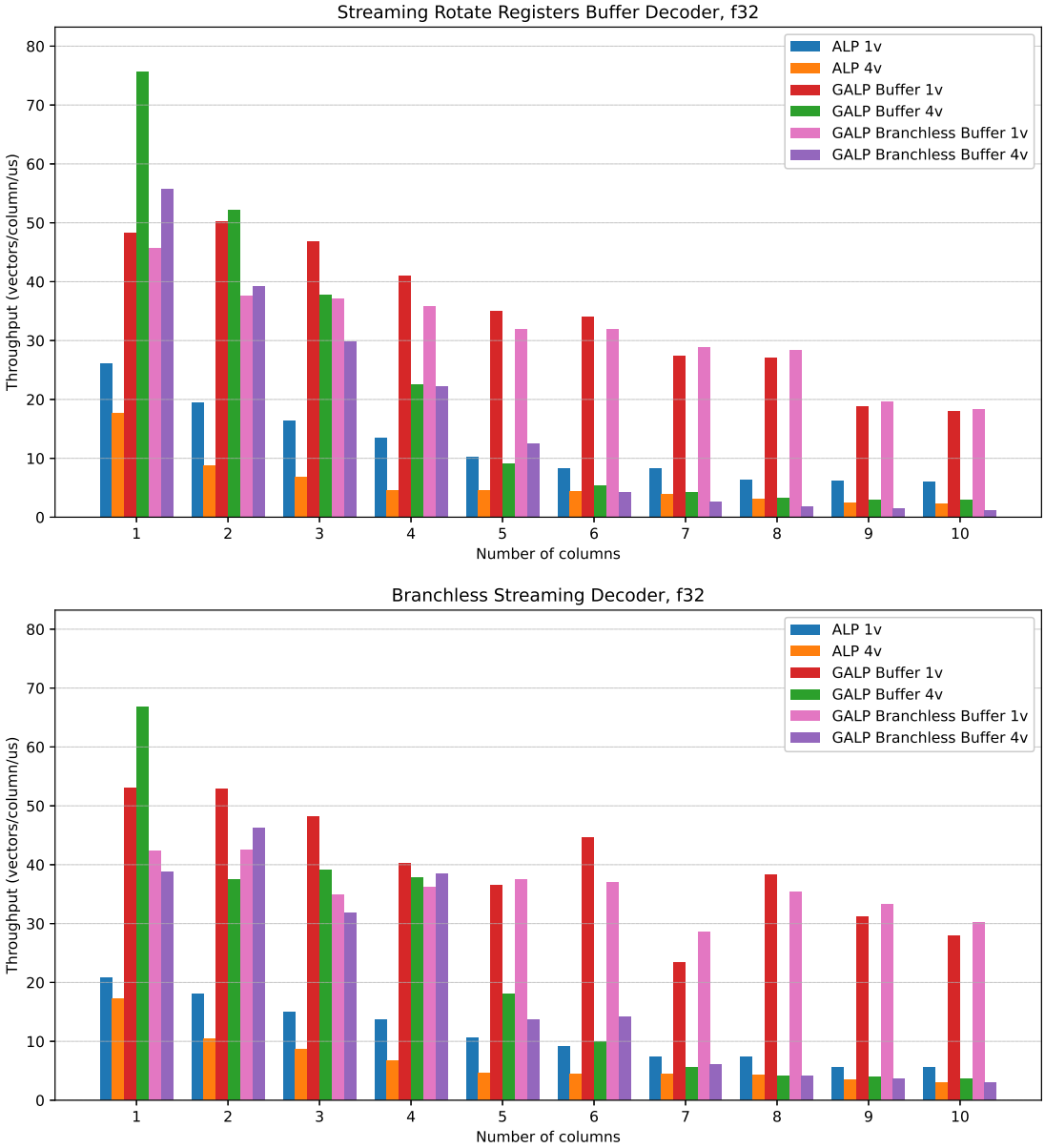


Figure 5.15: Comparison of the decoding throughput for multiple ALP patchers for `float` columns when decoding multiple columns concurrently. The top graph shows the decoding throughput of the patchers in combination with the best streaming decoder with buffer, the bottom graph shows the decoding throughput of the patchers in combination with the branchless streaming decoder. The bitwidth is random per vector, between 3 and 8, and the number of exceptions for each vector is constant at 20 exceptions per vector. Both decoders perform better than the FastLanesOnGPU decoder in Figure 5.14. The streaming decoder with buffer is better when decoding a single column, but the branchless streaming decoder is better when decoding multiple columns concurrently. The single vector G-ALP patcher with buffer achieves the highest decoding throughput. For some numbers of columns the branchless version is better, for other numbers of columns the normal one is better. Multivector patching results in lower throughput when decoding higher numbers of columns concurrently.

than the branchless streaming decoder when decoding a single column, but branchless streaming is faster when two or more columns are decoded. Additionally, multivector decoding now only improves throughput when decoding one or two columns in parallel, with a heavy performance drop-off in comparison to single vector decoding four or more columns.

For the branchless streaming decoder, the best patcher is the single vector G-ALP with buffer patcher. This patcher achieves slightly higher throughput for low column counts, while performing approximately the same as the branchless G-ALP with buffer patcher for higher column counts.

In Figure 5.16 the performance of the streaming decoder and branchless streaming decoder for decoding `double` columns is shown. The performance pattern is almost the same as for decoding `float` columns, with the exception that multivector decoding is now worse in all cases than single vector decoding. The branchless streaming decoder is best in all cases, however the decoding throughput does not exceed far beyond 20 vectors per column per microsecond, as was the case in the previous micro-benchmarks. This can be caused by the fact that there are fewer double precision floating-point instruction pipelines than single-precision floating-point pipelines.

### 5.3.5 Summary

The FastLanesOnGPU decoder in combination with ALP results in low throughput when decoding multiple columns simultaneously. The branchless streaming decoder performed best, offering consistent decoding throughput, even when decoding multiple columns simultaneously. Multi-vector decoding is slow in combination with ALP when decoding more than two columns in parallel, diminishing the value of a multivector decoding API, as the performance is not consistent across compression schemes, and can even degrade performance in comparison to single vector decoding, while also making the API more complex. Decoding `float` columns is faster than decoding `double` columns. The throughput of decoding `double` columns does not exceed 25 vectors per microsecond, while the throughput of decoding `float` columns can reach speeds of up to 75 vectors per microsecond, approaching FFOR single vector decoding speeds.

## 5.4 Real Data Benchmarks

ALP and G-ALP are compared to Thrust and nvCOMP in a scenario where compressed data exists in the GPUs RAM and needs to be scanned. The scan operator will be the same operator as used in the microbenchmarks for FFOR and ALP. Executing the scan requires the nvCOMP compressors to first fully decompress the data, write it to RAM, and then launch a separate kernel that reads the data from RAM and executes the scan. ALP and G-ALP do not have to fully decompress the data, and can launch a scan kernel that executes the scan by loading compressed data directly.

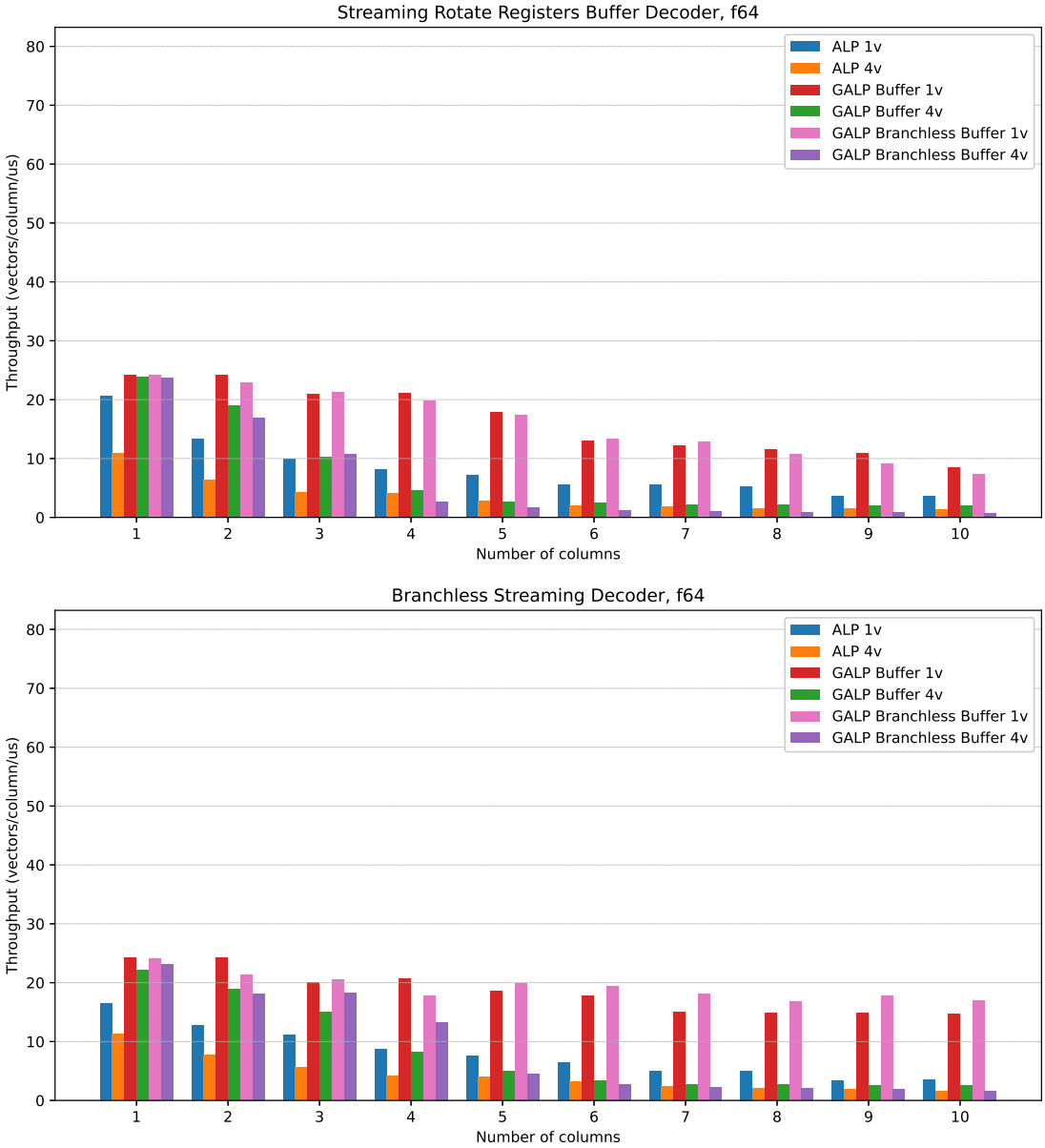


Figure 5.16: Comparison of the decoding throughput for multiple ALP patchers for `double` columns when decoding multiple columns concurrently. The top graph shows the decoding throughput of the patchers in combination with the best streaming decoder with buffer, the bottom graph shows the decoding throughput of the patchers in combination with the branchless streaming decoder. The bitwidth is random per vector, between 3 and 8, and the number of exceptions for each vector is constant at 20 exceptions per vector. The branchless streaming decoder is again the best decoder, resulting in more constant decoding throughput for high numbers of columns. The single vector, branchless G-ALP patcher with buffer has the highest decoding throughput. None of the patchers exceed a throughput of 25 vectors per column per $\mu$s.

Thrust, an NVIDIA library consisting of ready-made kernels, contains a scan operator as well. Thrust is also benchmarked to prove that executing a scan by loading compressed data is faster than loading normal data.

The nvCOMP compressors were configured to read the data as a stream of bytes, as nvCOMP currently does not contain compressors for `float` or `double` data.

The benchmark is executed on real datasets, collected from the ALP dataset[5], the public BI dataset[12], and the FC-Bench dataset[9]. Columns in those datasets that were not compressible with ALP or only with ALPrd were left out of the evaluation. For reasonable benchmarks the columns need to be large enough, to be able to saturate all SMs with multiple thread blocks for consistent measurements. In order to achieve this, columns of data that were smaller than 25600 vectors were repeatedly copied until they reached a size of 25600 vectors. Columns that were larger than 25600 vectors were cut off at 25600 vectors. This was done as some nvCOMP compressors crashed during compression of larger data sizes as they ran out of GPU RAM memory. These crashes are remarkable, as it is therefore hard to reliably compress more than 400 MB of double precision floating-point data on a GPU with 6 GB of RAM.

In these benchmarks, throughput is defined as the uncompressed size of the data, divided by the total execution time of the kernel.

### 5.4.1 Compression Ratio

A boxplot with the compression ratio for each compressor is shown in Figure 5.17. The exact bits per value per compressor and column are listed in Table B.1 and Table C.1. The exact compression ratio per compressor and column are listed in Table B.2 and Table C.2. For ALP and G-ALP, the average value bit width and average exception count per vector are listed in Table B.3 and Table C.3.

The `float` columns are less compressible than the `double` columns. For the `float` columns, ALP, G-ALP and nv-zstd achieve the best compression ratios, with median compression ratios of 1.70, 1.65 and 1.95 respectively. nv-LZ4, nv-Snappy, nv-Deflate and nv-GDeflate have median compression ratios of 1.33 and 1.43, 1.58 and 1.58 respectively. Bitcomp and BitcompSparse barely achieve any compression and have median compression ratios of 1.10 and 1.10.

The `double` columns are more compressible. ALP, G-ALP, and nv-zstd are again the strongest compressors, obtaining median compression ratios of 4.00, 3.93, and 3.72 respectively. nv-LZ4, nv-Snappy, nv-Deflate and nv-GDeflate have median compression ratios of 2.39 and 2.41, 1.35 and 1.34 respectively. nv-Deflate and nv-GDeflate therefore have a slightly lower median compression ratio than for the `float` columns. Bitcomp has a median compression ratios of 1.08, and BitcompSparse a median compression ratio of 1.00, meaning that BitcompSparse is not able to compress the majority of the columns.

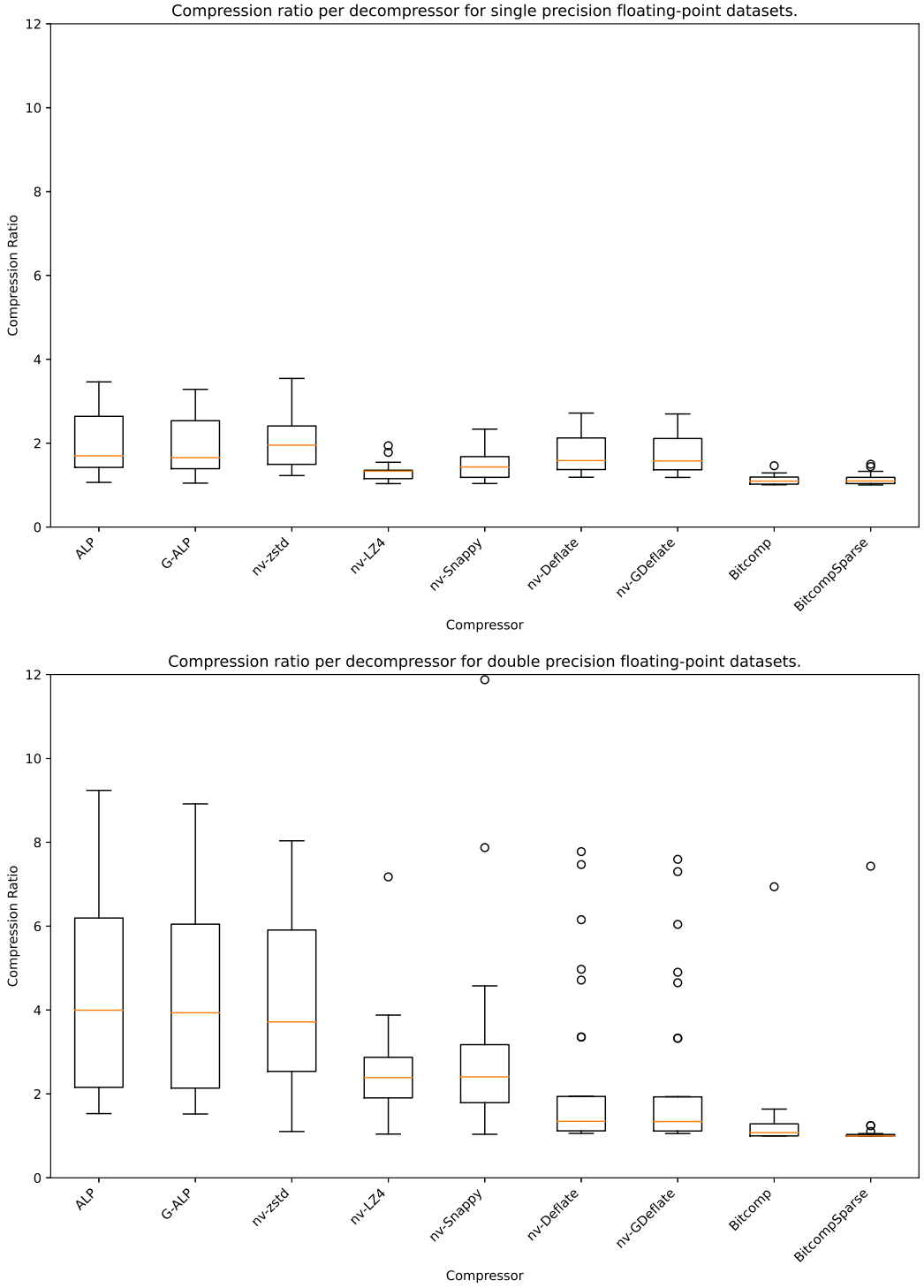


Figure 5.17: Compression ratio comparison for every datasets for all compressors as a boxplot. The single-precision floating-point data sets compression ratios are low for all compressors, also due to the fact that `float` is a smaller data type than `double`, so the decompressed size is relatively smaller. The median compression ratios for the `float` datasets are similar for all compressors except nv-zstd, which achieved the highest median compression ratio, and the bitcomp compressors, which achieved the lowest compression ratios. Nv-zstd, ALP and G-ALP get the best compression ratios for the `double` datasets . G-ALP has a slightly lower compression ratio than ALP, which is expected as the exception layout contains more metadata. All other compressors achieve a lower compression ratio, with the Bitcomp compressors achieving almost no compression.

### 5.4.2 Filter Throughput

Now the scan throughput of all compressors and Thrust are compared. The results can be found in Table B.4 and Table C.4. The throughput for each compressor and Thrust is also plotted as a boxplot in Figure 5.18.

For `float` data, G-ALP achieves a median throughput of 131.71 GB/s, almost 5x as large as the throughput of ALP, which is 27.52 GB/s. ALP only barely has higher throughput than Thrust's throughput of 20.71 GB/s. The median throughput of Bitcomp and BitcompSparse are 32.84 GB/s and 37.36 GB/s beating Thrust. Presumably Bitcomp and BitcompSparse store in a header which blocks are actually compressed, completely skipping blocks of data that it could not compress. For many datasets Bitcomp and BitcompSparse obtain no compression at all. The scan operator which loads uncompressed data is faster to execute full scans than the general purpose scan operator of Thrust, which is how Bitcomp and BitcompSparse manage to have a higher throughput than Thrust. The median scan throughput of the other compressors is really low, none exceeding 6 GB/s. Even though the compression ratio of nv-zstd is quite high, the scan throughput on compressed data is only 2.31 GB/s, 57x smaller than the scan throughput of G-ALP.

For `double` data, the throughput of G-ALP is slightly higher, at 148.34 GB/s. From Figure 5.18 can also be seen that the boxplot is very narrow, indicating very consistent scan throughput. The reason why the scan throughput on the real data columns is higher than for the `float` columns, is because the `double` columns contain far fewer exceptions per vector, as can be seen in Table B.3 and Table C.3. With the fewer amount of exceptions, ALP now also has a much higher throughput than Thrust, Bitcomp and BitcompSparse. The other compressors again do not achieve a median scan throughput higher than 6 GB/s. Zstd has a median scan throughput of 2.28, 65x smaller than the scan throughput of G-ALP.

The compression ratios and scan throughput per column and compressor are also plotted as a scatter diagram in Figure 5.19 for `float` columns, and in Figure 5.20 for `double` columns. In both figures it is visible that ALP and G-ALP achieves both high scan throughput and reasonable compression ratio. with most other compressor dots located mainly in the bottom left corner, indicating both low scan throughput and low compression ratio, with some compressors having a few dots further along the compression ratio axis, but with minimal scan throughput.

### 5.4.3 Full Decompression Throughput into Global Memory

Additionally, the full decompression throughput is shown in Figure 5.21. With full decompression each compressor fully decompresses the compressed data into the GPUs RAM. For ALP and G-ALP this results in a lower throughput, as the bottleneck is no longer reading compressed data, but writing the much larger decompressed data to

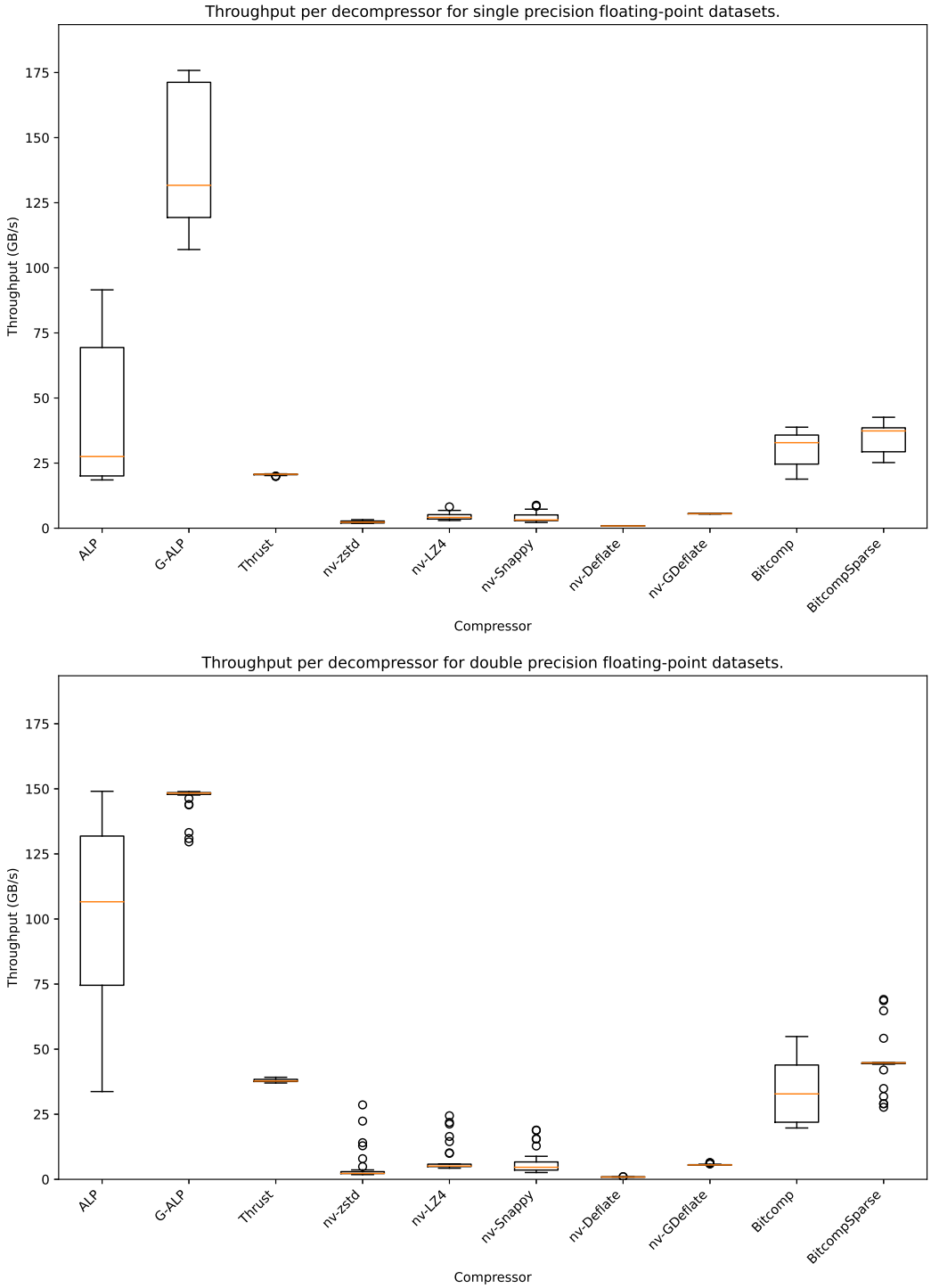


Figure 5.18: Throughput comparison for the filter benchmark for every dataset for all compressors. The ALP and G-ALP decoders are the fastest, achieving the highest throughput. G-ALP is faster than ALP, and much more consistent. None of the nvCOMP general purpose compressors exceed a throughput of 10 GB/s. The Bitcomp compressors achieve fairly high throughput, but these compressors barely compress the data.

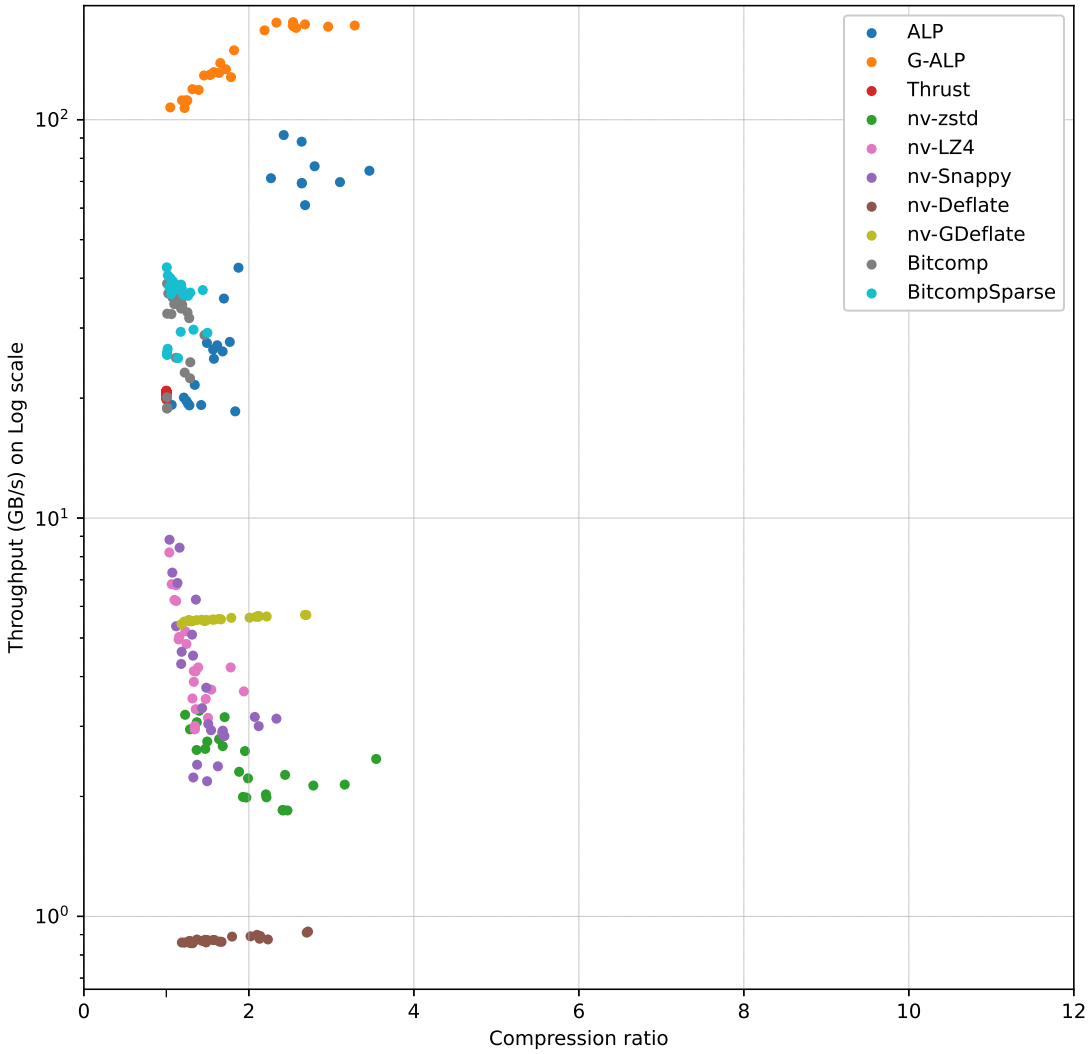


Figure 5.19: Compression ratio and throughput on log scale comparison for every compressor for the filtering benchmark for all single precision floating-point datasets. Each dot represents a single dataset decompressed by one compressor. From the scatter diagram it is visible, that ALP's and G-ALP's throughput increases when the compression ratio increases. For nv-Snappy and nv-zstd the reverse is true, the throughput decreases when the compression ratio is increased. The deflate compressors seem to be insensitive in regards to how compression ratio affects the throughput.

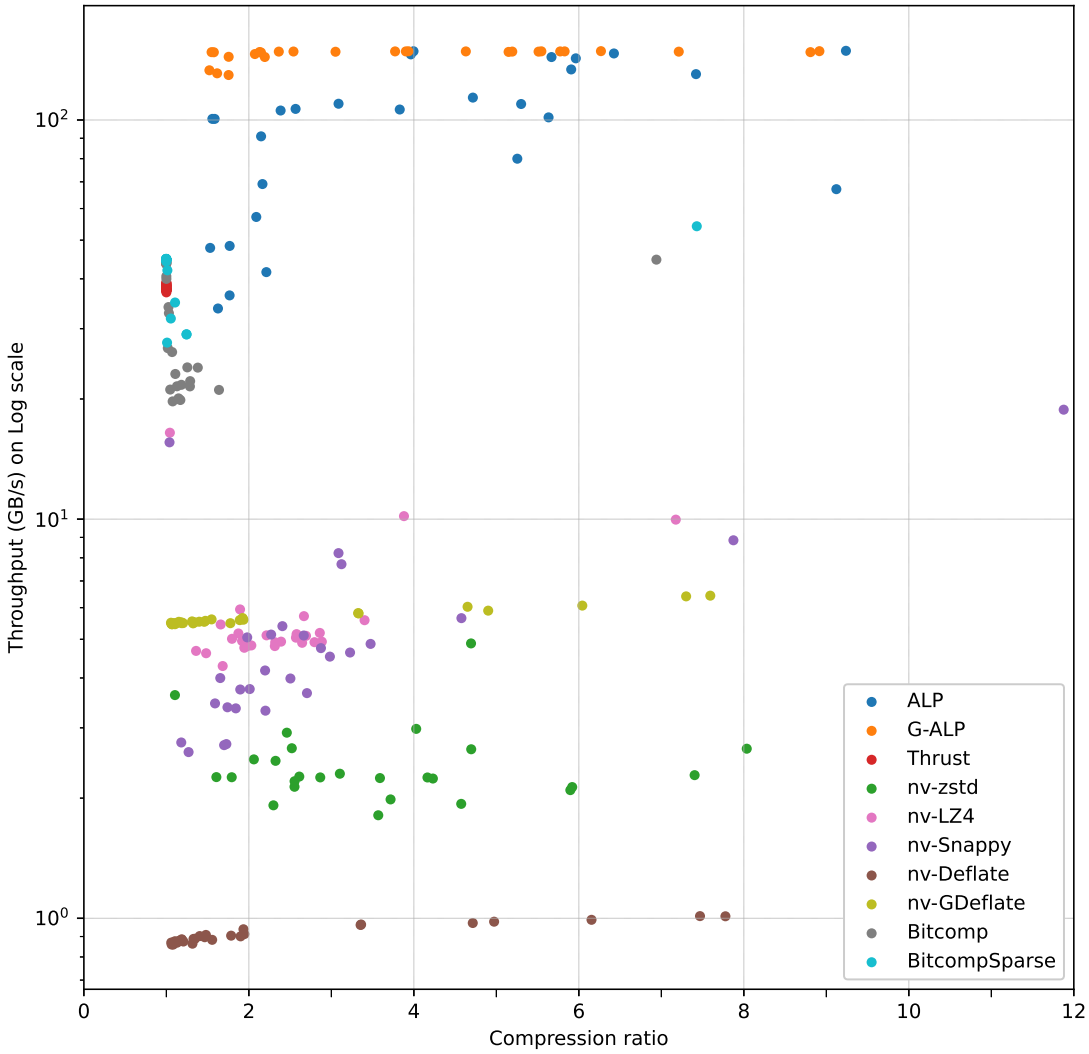


Figure 5.20: Compression ratio and throughput comparison for every compressor for the filtering benchmark for all double precision floating-point datasets. Each dot represents a single dataset decompressed by one compressor. All compressors achieve higher compression ratios, with larger spreads in compression ratio and throughput. There no longer appears to be a link between compression ratio and throughput for the nvCOMP compressors. G-ALP seems to reach a decoding throughput limit for almost all compression ratios, while ALP still decodes faster if the compression ratio is higher.

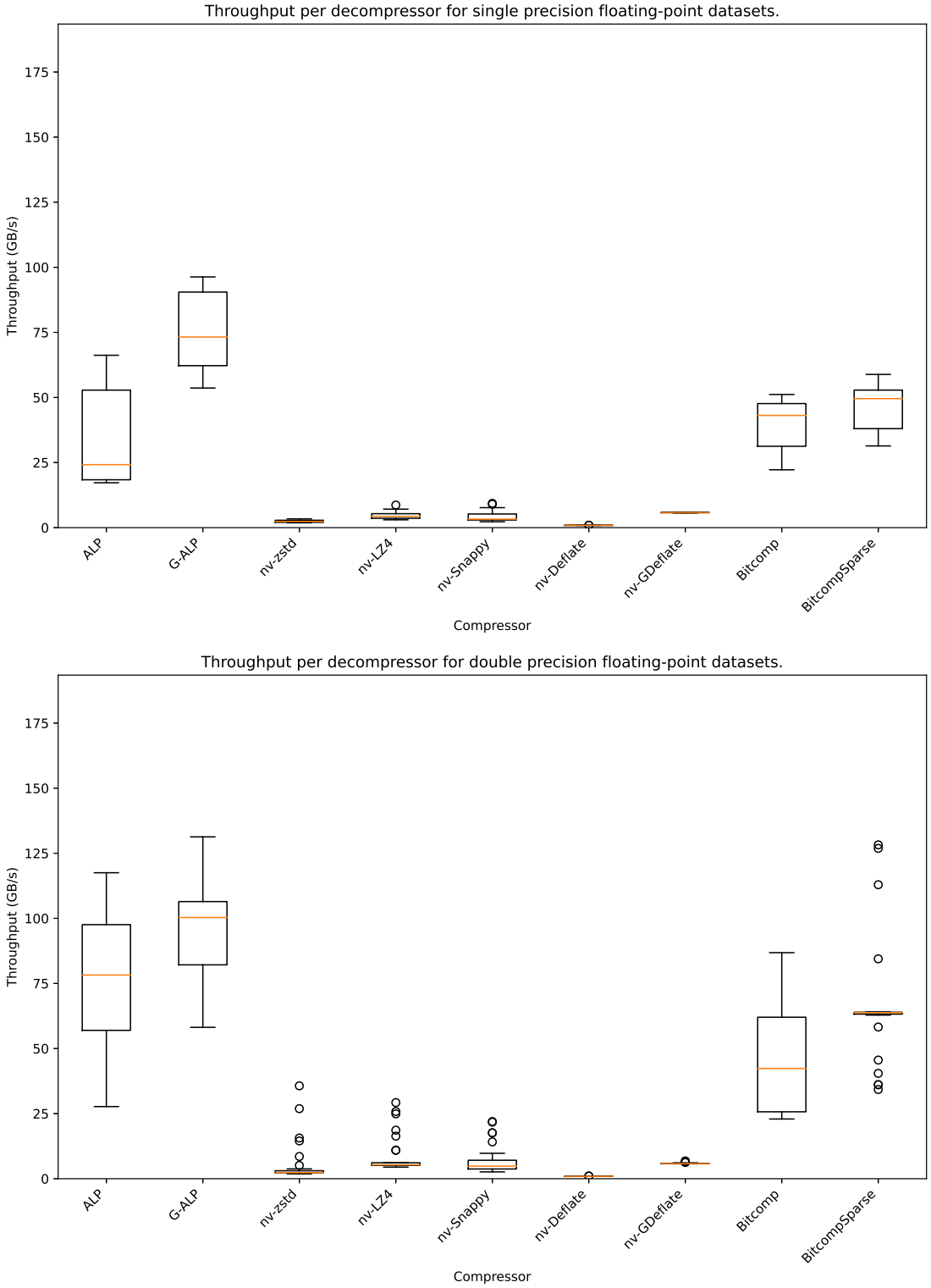


Figure 5.21: Throughput comparison for fully decompressing every dataset into the GPU's RAM, for all compressors. The difference in decoding throughput between the ALP, G-ALPand the nvCOMP compressors is now less pronounced in comparison to the filter throughput benchmark, as the ALP compressors now have to also write the data to RAM, costing a lot of memory throughput. Meanwhile the nvCOMP compressors achieve a slightly higher throughput than in Figure 5.18 as they can skip the filtering step, and were decompressing into RAM regardless.

RAM. For the other compressors, however, this benchmark improves the throughput, as it is no longer necessary to scan the data after decompressing the data into RAM, essentially removing a step in the process. The exact decompression throughput data can be viewed in Table B.5 and Table C.5.

G-ALP still has the highest throughput of all compressors, with a median decompression throughput of 73.20 GB/s for `float` columns, and 100.32 GB/s for `double` columns. ALP has a median decompression throughput of 24.20 GB/s for `float` columns, and 78.24 GB/s. Bitcomp and BitcompSparse have a median decompression throughput of 43.09 GB/s and 49.56 GB/s respectively for `float` columns, and a median decompression throughput of 42.34 GB/s and 63.80 GB/s respectively for `double` columns. All other compressors do not have a median decompression throughput higher than 6 GB/s. nv-zstd has a median decompression throughput of 2.38 GB/s for `float` columns, and a median decompression throughput of 2.34 GB/s for `double` columns.

### 5.4.4 Summary

The ALP, G-ALP and nv-zstd compressors achieve similar compression ratios for the real data columns. However, with G-ALP, compressed data can be scanned more than 50x faster than scanning data that is compressed by nv-zstd. All other compressors achieved both lower compression ratios as well as lower scan and decompression throughput than G-ALP. ALP performed worse than G-ALP, especially for the `float` columns, which contained a large amount of exceptions. Scanning compressed data with G-ALP is even faster than scanning normal data with NVIDIA's scan operator in the Thrust library, proving that loading compressed data can speed up memory read throughput bound GPU kernels.

# Chapter 6

# Conclusion

This thesis showed that the current state of the art compressors made by NVIDIA can be beaten in both decompression throughput as well as compression ratio for floating-point data by implementing ALP on the GPU. To implement ALP on the GPU, this thesis first discussed why compression is valuable for GPUs, what heavy-weight compression is, what light-weight compression is, and what the benefits of light-weight compression are over heavy-weight compression. Then the FastLanes file format was discussed, both how it can encode integers with FFOR compression and how floating-point data is compressed using the ALP encoding. Finally, the nvCOMP compression library from NVIDIA that contains both HWC as well as LWC encodings was discussed. The next chapter provided an in-depth description of the internal hardware of NVIDIA GPUs.

The following chapter used this background knowledge on compression and GPU internals, to create new decoding algorithms for FFOR, and a new exception layout to patch exceptions for ALP, thereby creating G-ALP. Finally, the decoding throughput of all decoders and exception patchers was thoroughly measured with microbenchmarks, on both 32-bit as well as 64-bit data types. To prove that G-ALP achieves a higher decoding throughput than the current state of the art in GPU floating-point data decompressors, G-ALP was compared with all of NVIDIA's compressors in the nvCOMP library on real datasets.

This thesis consists of three main contributions. The first contribution of this thesis is a newly optimized decoder for the FastLanes interleaved bitpacking format, that also offers a new single value decoding API. This single value API offers fine-grained, high throughput access to compressed data. With this API data can be left compressed in RAM and directly accessed and read by GPU kernels. The second contribution of this thesis is a new exception layout, which data-parallelizes the exception patching stage for the FastLanes file format. The third contribution of this thesis is creating the G-ALP decoder, a new GPU based floating-point data decoder that utilizes the new exception layout to comprehensively beat the state of the art in floating-point data decompression on GPUs.

With these contributions it is possible to decompress floating-point data at high throughput. This makes compression a viable technique for GPU workloads to save PCIe link bandwidth to speed up data transfers to GPUs, and to store more data in the GPU's RAM. With the fine-grained, high-throughput, unobtrusive decoding API it is also possible to decode data just before the data is processed within a data processing kernel. This has two benefits for data intensive GPU workloads. The first benefit is that this removes the need for a separate data decompression kernel. The second benefit is that more RAM space can be saved and used for other purposes, because the data does not need to be decompressed into RAM.

A paper was written based on the work of this thesis, which was accepted for the International Workshop on Data Management on New Hardware (DaMoN). The paper will be presented at the DaMoN workshop that is held with the ACM SIGMOD/PODS conference in 2025 in Berlin.

## 6.1 Research Questions

To answer the research questions from Chapter 1, each question is individually revisited and answered.

> *RQ 1: How should a general purpose API for single value decoding of bit-packed data be implemented?*

A general purpose API for high throughput single value decoding of light-weight encodings should be able to both fully decompress encoded columns, and scan encoded columns, at high throughput. For fully decompressing columns, the decoding algorithm is not important. All FastLanes decoding approaches that were benchmarked in this thesis were equally fast in fully decoding columns, as the decoders are completely bound by memory write throughput. For scanning single columns, the FastLanesOnGPU decoder performs best. However, when decoding multiple columns concurrently, as can be expected when executing real queries, the FastLanesOnGPU decoding performance quickly deteriorates. The FastLanesOnGPU decoder suffers from decreased occupancy, because the decoder processes entire lanes at a time.

To improve upon the FastLanesOnGPU decoder, a single value API is better. The switch based decoding approach however does not perform well when decoding a single value at a time. Instead, a branchless streaming decoder is better, offering consistent performance in a variety of benchmarks. Even when decoding multiple columns the decoding throughput is constant, achieving higher decoding throughput than the FastLanesOnGPU decoder.

Decoding multiple vectors concurrently is promising for FFOR, achieving higher throughput in all scenarios. However, when decoding multiple columns in ALP, the decoding throughput is lower than the single vector decoding approach. The API for

multivector decoding is also slightly more complex and inflexible. Therefore, as the performance benefit is not consistent and not large enough, a general purpose decoding API should only offer a single vector decoding API.

*RQ 2: How can exceptions be efficiently patched on GPUs?*

The original ALP exception layout cannot be efficiently processed on GPUs, as every thread needs to read and process all values in the exception position array. For efficient patching, each lane needs to be decoded independently from the other lanes. This removes the need for a thread to scan positions that do not belong in the thread's own lane. This can be achieved by reordering the exception positions and exception values by lane. Now threads can start reading exceptions positions beginning at their own offsets in the exception positions array, reading only the minimal amount of exception positions. To allow each thread to find their section efficiently does require additional metadata, containing the offset of the lane section and the number of exceptions per lane. The additional metadata does decrease the compression ratio slightly, but the improved decoding throughput is significant and very consistent.

The decoding throughput of G-ALP does not exceed the memory throughput of the GPU on which all benchmarks were executed. Unlike on the CPU, where data can be processed faster by loading and decoding that data using ALP, the decoding throughput of G-ALP is approximately equal to the normal data loading throughput. However, because the loading throughput of decompressed data and the decoding throughput of compressed data is approximately equal, it is still unnecessary to launch a separate decompressing kernel and decompress data into RAM. The limited decoding speed is due to limited `double` instruction throughput on GPUs, and the fact that the single value patching approach uses more instructions to patch a value than the CPU patching approach. On the CPU each value can be patched by loading the index and value of an exception and issuing a write instruction to patch the exception in the decompressed data array. With the single value decoding API on the GPU, the decoder needs to check for each decoded value whether it was an exception, and then find the corresponding exception value and patch the value. A specialized, purpose-built GPU kernel might be able to patch exceptions more efficiently using the CPU patching approach. But this decoder would not be an unobtrusive, general purpose, single value API that does not use any special memory or make any assumptions on which threads load which values, which was the goal of the single value decoding API.

*RQ 3: Can floating-point data be decoded faster with ALP than with common GPU compression schemes?*

G-ALP, which is ALP but with the new data-parallel exception layout, beats all compressors in the nvCOMP NVIDIA library, the state of the art in floating-point data

decompression on GPUs. G-ALP can fully decompress data faster than nvCOMP's compressor by an order of magnitude. G-ALP also achieves a higher compression ratio, only being equalled by nv-zstd. The difference in decoding throughput is even larger when considering real world scenarios such as having to scan encoded data. nvCOMP's compressors first need to fully decompress data back into the GPU's RAM, before launching a separate kernel to scan the data. However, G-ALP is able to decode the data in the same kernel that is scanning the data, allowing the kernel to load compressed data, reducing the impact of the memory read throughput bottleneck. Due to this optimization, scanning encoded floating-point data using G-ALP is even faster than scanning unencoded data using Thrust, NVIDIA's standard library of general purpose kernels.

## 6.2 Future Work

### 6.2.1 CPU Benchmarks Data Parallel Exception

The new exception layout proposed in this thesis was only tested for decoding throughput on GPUs. No benchmarks were performed to investigate the impact on the encoding throughput and the decoding throughput on CPUs. The exception encoding now requires an ordering based on in which lane the exception occurred. Additionally, the new layout does not require a total exception count per vector to be included in the format. However, including this total exception count could speed up exception decoding on CPUs, as otherwise the exception count needs to be extracted from the per lane headers, which contain the lane's exceptions count and offset. To adopt the new exception layout for real world usage, the encoding and decoding throughput of the new exception layout would need to be benchmarked on CPUs and compared to the performance of the original ALP exception layout.

### 6.2.2 Benchmarking Other GPUs

All benchmarks in this thesis were executed on NVIDIA GPUs. However not all compute capabilities were tested in this thesis. To confirm that there are no significant deviations in performance on different compute capabilities, the decoders need to be tested on more NVIDIA GPUs. Additionally there are also other vendors of GPUs, such as AMD, Apple and Intel. The GPU designs of these vendors differ significantly from each other, AMD for example uses warps with 64 threads instead of 32. Benchmarking FastLanes decoding on these GPUs would require either rewriting the CUDA code to other vendor-specific languages, or require a rewrite into a language that runs on any GPU, such as webGPU.

### 6.2.3 New Benchmarks

There is currently no adequate benchmark to test the benefits of compression in database GPU kernels, due to the lack of widely adopted GPU database systems. The advantages

and disadvantages of compression could more thoroughly be proven if the decoding process could be integrated in these kind of kernels. Important questions such as how data should be transferred to the GPU and managed on the GPU remain open.

### 6.2.4 FastLanes GPU File Reader

This thesis implemented a decoding API for high-throughput bitpacking decoding and high-throughput exception patching, but did not cover all encodings of the FastLanes file format. A GPU decoder that is able to decompress, and is fully compliant with, the FastLanes file format is still necessary. There are three benefits to implementing this GPU decoder. First, the implementation would show that FastLanes can be used for GPU workloads. Second, this decoder would enable more experimentation with using compression and file formats in GPU workloads. Third, implementing this decoder could show which format changes need to be made to efficiently decode the FastLanes file format on GPUs.

The first step to creating a fully compliant GPU decoder for FastLanes would be to create kernels that can fully decompress the data into RAM. These kernels are less performance sensitive as write throughput will dominate the execution time, except perhaps for any encodings that need to randomly access data during decoding such as the DICT encoding. These kernels are therefore easier to implement. Fully decompressing into RAM also does not require data processing kernels to be adapted to use the FastLanes decoder, the kernels can simply read the decompressed data from RAM.

To test whether any changes to the FastLanes format need to be made in order to efficiently decode the format on the GPU, the unobtrusive single-value decoding API should be implemented for the other encodings in the format. To create an unobtrusive single-value decoding API the API should preferably not rely on any special memory allocations such as shared memory, should use few registers to enable the user of the API to achieve high occupancy, and should be able to decompress the data into any kind of memory region. This last requirement means that the decoding API should take a pointer for the output memory, and should not dynamically access that output memory as then the data cannot be decoded into registers.

A last step for enabling GPU workloads to use FastLanes would be to implement a GPU encoder, to also be able to send the results of kernels back to the host in compressed form over the PCIe link. This could also surface changes that could be made to the format to enable fast encoding of data on GPUs.

#### Expression Encodings

In the FastLanes file format encodings can be combined using expression encodings, where the result of one encoding is encoded in turn by another encoder. This makes the decoding of columns dynamic. This is problematic for the single value decoding

API, as the cost of selecting the right decoding functions can only be amortized across a single value, instead of an entire lane or vector. Decoding ALP is already similar in throughput to loading the data normally, so creating an unobtrusive API that can also flexibly decode values at high throughput with a single value granularity might prove difficult.

For single value decoding within the data processing kernel, there are at least two possible approaches to decode columns that were encoded with expression encodings. The first approach would be to simply select the right decoding function at runtime for every single value that is decoded. This approach might be difficult to achieve high throughput with. The second approach would be to compile data-processing kernels at run time. The data processing kernels could then be compiled just in time to use hardcoded, fused decoders for the expression that the target column was compressed with.

However, the most simple approach to decoding expression encodings on the GPU would be to simply create a decoding kernel for each encoding. Each decoding kernel would then read data from RAM and decode that data back into RAM. But, this requires memory in RAM to store all intermediary decoding results, and also space for the decoded data. Additionally it requires a separate decompression step instead of loading compressed data into data processing kernels directly. This approach could be slightly improved by fusing the most common sequences of encodings into a single decoding kernel.

### Implications for the FastLanes File Format for GPU Decoding

While this thesis proved that the core bit interleaved layout of FastLanes can be decoded at high throughput on GPUs, two possible modifications to the file format for efficient GPU decoding arose from the research. The first modification would be to use the new exception layout for exception patching if CPU encoding and decoding throughput is not too negatively impacted by this new layout. The second modification, or rather design guideline, would be for any encoding to store the decompressed size per vector, rowgroup or column, unless the decompressed size can be derived from the compression parameters. Knowing the decoded size is important as then enough memory can be allocated on the GPU to decode the data without needing to overprovision.

# Bibliography


[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 671–682. ISBN: 1595934340. DOI: 10.1145/1142473.1142548. URL: https://doi.org/10.1145/1142473.1142548.

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

[3] Azim Afroozeh and Peter Boncz. "The FastLanes Compression Layout: Decoding ¿ 100 Billion Integers per Second with Scalar Code". In: *Proc. VLDB Endow.* 16.9 (May 2023), pp. 2132–2144. ISSN: 2150-8097. DOI: 10.14778/3598581.3598587. URL: https://doi.org/10.14778/3598581.3598587.

[4] Azim Afroozeh, Lotte Felius, and Peter Boncz. "Accelerating GPU Data Processing using FastLanes Compression". In: *Proceedings of the 20th International Workshop on Data Management on New Hardware*. DaMoN '24. Santiago, AA, Chile: Association for Computing Machinery, 2024. ISBN: 9798400706677. DOI: 10.1145/3662010.3663450. URL: https://doi.org/10.1145/3662010.3663450.

[5] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. "ALP: Adaptive Lossless floating-Point Compression". In: *Proc. ACM Manag. Data* 1.4 (Dec. 2023). DOI: 10.1145/3626717. URL: https://doi.org/10.1145/3626717.

[6] Peter Boncz Azim Afroozeh. *The FastLanes File Format*. 2025.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. www.cidrdb.org, 2005, pp. 225–237. URL: http://dblp.uni-trier.de/db/conf/cidr/cidr2005.html#BonczZN05.

[8] Ishita Chaturvedi et al. "GhOST: a GPU Out-of-Order Scheduling Technique for Stall Reduction". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 2024, pp. 1–16. DOI: 10.1109/ISCA59077.2024.00011.

[9] Xinyu Chen et al. “FCBench: Cross-Domain Benchmarking of Lossless Compression for Floating-point Data”. In: *Proceedings of the VLDB Endowment* 17.7 (2021).

[10] Yann Collet. *lz4*. Accessed: 2025-03-10. 2025. URL: https://github.com/lz4/lz4.

[11] Yann Collet. *RFC8478: Zstandard Compression and the application/zstd Media Type*. USA, 2018.

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

[13] P. Deutsch. *RFC1951: DEFLATE Compressed Data Format Specification version 1.3*. USA, 1996.

[14] J. Goldstein, R. Ramakrishnan, and U. Shaft. “Compressing relations and indexes”. In: *Proceedings 14th International Conference on Data Engineering*. 1998, pp. 370–379. DOI: 10.1109/ICDE.1998.655800.

[15] Goetz Graefe and Leonard Shapiro. “Data Compression and Database Performance”. In: (Dec. 2001). DOI: 10.1109/SOAC.1991.143840.

[16] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[17] Steinar Gunderson Jeff Dean Sanjay Ghemawat. *snappy*. Accessed: 2025-03-10. 2011. URL: https://github.com/google/snappy.

[18] Zhe Jia et al. *Dissecting the NVidia Turing T4 GPU via Microbenchmarking*. 2019. arXiv: 1903.07486 [cs.DC]. URL: https://arxiv.org/abs/1903.07486.

[19] Zhe Jia et al. *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*. 2018. arXiv: 1804.06826 [cs.DC]. URL: https://arxiv.org/abs/1804.06826.

[20] Maximilian Kuschewski et al. “BtrBlocks: Efficient Columnar Compression for Data Lakes”. In: *Proceedings of the ACM on Management of Data* 1 (June 2023), pp. 1–26. DOI: 10.1145/3589263.

[21] M.S. Lam and R.P. Wilson. “Limits of Control Flow on Parallelism”. In: *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 1992, pp. 46–57. DOI: 10.1109/ISCA.1992.753303.

[22] Weile Luo et al. *Benchmarking and Dissecting the NVIDIA Hopper GPU Architecture*. 2024. arXiv: 2402.13499 [cs.AR]. URL: https://arxiv.org/abs/2402.13499.

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

[24] NVIDIA. *Coalesced Groups*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#coalesced-groups.

[25] NVIDIA. *Compute Capabilities*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities.

[26] NVIDIA. *CUDA Pro Tip: Increase Performance with Vectorized Memory Access*. NVIDIA Developer Blog. 2025. URL: https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/.

[27] NVIDIA. *CUDA Warps and Occupancy*. NVIDIA Webinar. 2025. URL: https://developer.download.nvidia.com/CUDA/training/cuda_webinars_WarpsAndOccupancy.pdf.

[28] NVIDIA. *CUTLASS Source code*. 2025. URL: https://github.com/NVIDIA/cutlass/blob/main/include/cutlass/arch/memory_sm80.h.

[29] NVIDIA. *Device memory accesses*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses.

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

[31] NVIDIA. *Hardware Implementation*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/%5C#hardware-implementation.

[32] NVIDIA. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. NVIDIA Developer Blog. 2025. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/.

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

[34] NVIDIA. *Issue Efficiency*. NVIDIA GameWorks Documentation. 2025. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm.

[35] NVIDIA. *Multiprocessor Level*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#multiprocessor-level.

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

[37] NVIDIA. *nvCOMP Low Level Decompression API*. nvCOMP Documentation. 2025. URL: https://github.com/NVIDIA/nvcomp/blob/main/doc/lowlevel_c_quickstart.md#decompression-api.

[38] NVIDIA. *nvCOMP Supported Datatypes*. nvCOMP Documentation. 2025. URL: https://docs.nvidia.com/cuda/nvcomp/c_api.html#_CPPv412nvcompType_t.

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

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

[41] NVIDIA. *NVIDIA CUDA Compiler Driver NVCC*. NVIDIA Documentation. 2025. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[42] NVIDIA. *NVIDIA nvCOMP Library*. Accessed: 2025-03-10. 2025. URL: https://docs.nvidia.com/cuda/nvcomp/.

[43] NVIDIA. *NVIDIA SASS Instruction Set Reference*. NVIDIA Documentation. 2025. URL: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-reference.

[44] NVIDIA. *NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[45] NVIDIA. *Occupancy Calculator*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#occupancy-calculator.

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

[47] NVIDIA. *Parallel Thread Execution ISA Version 8.7*. 2025. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[48] NVIDIA. *PTX and SASS Assembly Debugging*. NVIDIA GameWorks Documentation. 2025. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/ptx_sass_assembly_debugging.htm.

[49] NVIDIA. *Register Pressure*. CUDA C++ Best Practices Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#register-pressure.

[50] NVIDIA. *SIMT Architecture*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture.

[51] NVIDIA. *Technical Specifications per Compute Capability*. CUDA C++ Programming Guide. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#features-and-technical-specifications-technical-specifications-per-compute-capability.

[52] NVIDIA. *Using CUDA Warp-Level Primitives*. NVIDIA Developer Blog. 2025. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives.

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

[54] NVIDIA. *Using Shared Memory in CUDA C/C++*. NVIDIA Developer Blog. 2025. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[55] NVIDIA. *Your GPU Compute Capability*. NVIDIA Documentation. 2025. URL: https://developer.nvidia.com/cuda-gpus.

[56] Anil Shanbhag et al. "Tile-based Lightweight Integer Compression in GPU". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 1390–1403. ISBN: 9781450392495. DOI: 10.1145/3514221.3526132. URL: https://doi.org/10.1145/3514221.3526132.

[57] Vasily Volkov. "Understanding Latency Hiding on GPUs". PhD thesis. EECS Department, University of California, Berkeley, Aug. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html.

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

[59] Till Westmann et al. "The implementation and performance of compressed databases". In: *SIGMOD Rec.* 29.3 (Sept. 2000), pp. 55–67. ISSN: 0163-5808. DOI: 10.1145/362084.362137. URL: https://doi.org/10.1145/362084.362137.

[60] Xiuxia Zhang et al. *Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning*. Jan. 2017. DOI: 10.1145/3018743.3018755.

[61] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

[62] M. Zukowski et al. “Super-Scalar RAM-CPU Cache Compression”. In: *22nd International Conference on Data Engineering (ICDE'06)*. 2006, pp. 59–59. DOI: 10.1109/ICDE.2006.150.

# Appendix A

# Benchmarking Environment

| Software | Version |
|---|---|
| g++ | 12.3.0 |
| Clang | 14.0.0 |
| nvcc | 12.5 |
| NVIDIA Driver | 555.42.06 |
| nvCOMP | 4.2.0 |

Table A.1: Versions of all software used in benchmarking.

| GPU Properties | |
|---|---|
| Name | GTX 1060 6GB |
| Compute Capability | 6.1 |
| Shared memory | 96 KB |
| L1 Cache | 48 KB |
| L2 Cache | 1536 KB |
| RAM | 6 GB |
| Memory bandwidth | 192.2 GB/s |
| Clock speed | 1506 MHz |
| Warps per SM | 64 |
| SM | 20 |

Table A.2: Hardware specifications of the GPU that used in all benchmarks in this thesis.

# Appendix B

# Single precision floating-point datasets results

| | ALP | G-ALP | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|
| **Average** | **18.32** | **18.82** | **17.15** | **24.63** | **22.64** | **19.55** | **19.64** | **28.54** | **28.38** |
| **Median** | **18.85** | **19.35** | **16.39** | **24.01** | **22.33** | **20.19** | **20.29** | **29.21** | **29.11** |
| CMSprovider-8-line_srvc_cnt | **11.44** | **11.94** | 12.97 | 23.77 | 18.78 | 14.97 | 15.06 | 31.61 | 31.50 |
| CityMaxCapita-13-lpf | **17.08** | 17.58 | **14.46** | 21.67 | 20.75 | 22.30 | 22.39 | 26.19 | 28.03 |
| MedPaymenT1-8-line_srvc_cnt | **12.11** | **12.61** | 13.27 | 23.80 | 19.05 | 15.06 | 15.16 | 31.75 | 31.73 |
| PanCreactomy1-8-line_srvc_cnt | **11.93** | **12.43** | 13.23 | 23.78 | 19.00 | 15.03 | 15.12 | 31.74 | 31.72 |
| Physicians-8-line_srvc_cnt | **12.11** | **12.61** | 13.27 | 23.80 | 19.05 | 15.06 | 15.16 | 31.75 | 31.73 |
| Telco-110-rechrg_total_load_p1 | **13.21** | **13.71** | 14.50 | 23.61 | 21.42 | 15.84 | 15.93 | 30.16 | 30.24 |
| Telco-111-rechrg_total_load_p2 | **12.12** | **12.62** | 13.12 | 20.72 | 19.70 | 15.25 | 15.34 | 26.85 | 27.17 |
| Telco-112-rechrg_total_load_p3 | **10.31** | 10.81 | **10.12** | 21.27 | 15.10 | 11.85 | 11.94 | 24.81 | 24.09 |
| Telco-113-rechrg_total_load_p4 | **9.25** | 9.75 | **9.03** | 16.50 | 13.71 | 11.78 | 11.87 | 21.88 | 22.20 |
| Telco-114-rechrg_total_load_p6 | **14.11** | **14.61** | 16.08 | 24.01 | 24.12 | 17.81 | 17.90 | 30.18 | 30.47 |
| Telco-151-total_mins_p1 | 25.35 | 25.85 | **23.37** | 29.13 | 28.20 | **25.05** | 25.14 | 29.76 | 29.85 |
| Telco-152-total_mins_p2 | 25.72 | 26.22 | **23.42** | 28.60 | 28.63 | **24.86** | 24.96 | 31.68 | 30.88 |
| Telco-153-total_mins_p3 | **20.30** | 20.80 | **19.01** | 25.73 | 24.19 | 21.59 | 21.69 | 27.15 | 27.00 |
| Telco-156-total_outgoing_min_p1 | 26.40 | 26.90 | **24.88** | 30.07 | 29.85 | **26.19** | 26.28 | 31.28 | 31.43 |
| Telco-157-total_outgoing_min_p2 | 25.02 | 25.52 | **22.87** | 28.62 | 27.58 | **24.78** | 24.87 | 28.79 | 28.92 |
| Telco-158-total_outgoing_min_p3 | **20.42** | 20.92 | **19.52** | 26.01 | 24.40 | 21.79 | 21.88 | 27.16 | 27.03 |
| Telco-159-total_outgoing_min_p4 | **19.03** | 19.53 | **18.75** | 25.14 | 23.57 | 21.43 | 21.52 | 25.47 | 25.36 |
| Telco-171-total_outgoing_rev_p1 | 30.05 | 30.55 | **26.07** | 30.90 | 30.83 | **26.94** | 27.03 | 31.73 | 31.84 |
| Telco-22-chrgd_rev_p1 | **21.46** | 21.96 | **21.40** | 27.72 | 27.13 | 23.34 | 23.43 | 29.50 | 29.58 |
| Telco-33-free_mins_p1 | 22.50 | 23.00 | **16.59** | 23.10 | 21.23 | **19.45** | 19.54 | 28.66 | 27.26 |
| Telco-35-free_mins_p3 | 17.44 | 17.94 | **11.50** | 17.98 | 15.44 | **14.35** | 14.44 | 24.86 | 21.37 |
| Telco-83-offnet_rev_p1 | **18.85** | 19.35 | **16.25** | 24.31 | 23.31 | 19.17 | 19.26 | 29.21 | 29.11 |
| Telco-93-onnet_mins_p1 | **23.81** | 24.31 | **21.74** | 27.85 | 27.00 | 24.28 | 24.38 | 29.55 | 29.59 |
| Telco-95-onnet_mins_p3 | **19.80** | 20.30 | **17.01** | 24.02 | 22.33 | 20.44 | 20.54 | 26.86 | 26.51 |
| Telco-96-onnet_mins_p4 | **18.09** | 18.59 | **16.39** | 23.64 | 21.57 | 20.19 | 20.29 | 25.06 | 24.81 |

Table B.1: Bits per value for each dataset and compressor. Lower is better, 64 bits per value is the upper bound for compressing data. Best bits per value per dataset in green, second best in yellow.

| | ALP | G-ALP | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|
| **Average** | **1.94** | **1.88** | **2.02** | **1.33** | **1.48** | **1.74** | **1.73** | **1.13** | **1.14** |
| **Median** | **1.70** | **1.65** | **1.95** | **1.33** | **1.43** | **1.58** | **1.58** | **1.10** | **1.10** |
| CMSprovider-8-line_srvc_cnt | **2.80** | **2.68** | 2.47 | 1.35 | 1.70 | 2.14 | 2.12 | 1.01 | 1.02 |
| CityMaxCapita-13-lpf | **1.87** | 1.82 | **2.21** | 1.48 | 1.54 | 1.43 | 1.43 | 1.22 | 1.14 |
| MedPaymenT1-8-line_srvc_cnt | **2.64** | **2.54** | 2.41 | 1.34 | 1.68 | 2.12 | 2.11 | 1.01 | 1.01 |
| PanCreactomy1-8-line_srvc_cnt | **2.68** | **2.57** | 2.42 | 1.35 | 1.68 | 2.13 | 2.12 | 1.01 | 1.01 |
| Physicians-8-line_srvc_cnt | **2.64** | **2.54** | 2.41 | 1.34 | 1.68 | 2.12 | 2.11 | 1.01 | 1.01 |
| Telco-110-rechrg_total_load_p1 | **2.42** | **2.33** | 2.21 | 1.36 | 1.49 | 2.02 | 2.01 | 1.06 | 1.06 |
| Telco-111-rechrg_total_load_p2 | **2.64** | **2.54** | 2.44 | 1.54 | 1.62 | 2.10 | 2.09 | 1.19 | 1.18 |
| Telco-112-rechrg_total_load_p3 | **3.10** | 2.96 | **3.16** | 1.50 | 2.12 | 2.70 | 2.68 | 1.29 | 1.33 |
| Telco-113-rechrg_total_load_p4 | **3.46** | 3.28 | **3.54** | 1.94 | 2.33 | 2.72 | 2.70 | 1.46 | 1.44 |
| Telco-114-rechrg_total_load_p6 | **2.27** | **2.19** | 1.99 | 1.33 | 1.33 | 1.80 | 1.79 | 1.06 | 1.05 |
| Telco-151-total_mins_p1 | 1.26 | 1.24 | **1.37** | 1.10 | 1.13 | **1.28** | 1.27 | 1.08 | 1.07 |
| Telco-152-total_mins_p2 | 1.24 | 1.22 | **1.37** | 1.12 | 1.12 | **1.29** | 1.28 | 1.01 | 1.04 |
| Telco-153-total_mins_p3 | **1.58** | 1.54 | **1.68** | 1.24 | 1.32 | 1.48 | 1.48 | 1.18 | 1.19 |
| Telco-156-total_outgoing_min_p1 | 1.21 | 1.19 | **1.29** | 1.06 | 1.07 | **1.22** | 1.22 | 1.02 | 1.02 |
| Telco-157-total_outgoing_min_p2 | 1.28 | 1.25 | **1.40** | 1.12 | 1.16 | **1.29** | 1.29 | 1.11 | 1.11 |
| Telco-158-total_outgoing_min_p3 | **1.57** | 1.53 | **1.64** | 1.23 | 1.31 | 1.47 | 1.46 | 1.18 | 1.18 |
| Telco-159-total_outgoing_min_p4 | **1.68** | 1.64 | **1.71** | 1.27 | 1.36 | 1.49 | 1.49 | 1.26 | 1.26 |
| Telco-171-total_outgoing_rev_p1 | 1.06 | 1.05 | **1.23** | 1.04 | 1.04 | **1.19** | 1.18 | 1.01 | 1.00 |
| Telco-22-chrgd_rev_p1 | **1.49** | 1.46 | **1.50** | 1.15 | 1.18 | 1.37 | 1.37 | 1.08 | 1.08 |
| Telco-33-free_mins_p1 | 1.42 | 1.39 | **1.93** | 1.39 | 1.51 | **1.65** | 1.64 | 1.12 | 1.17 |
| Telco-35-free_mins_p3 | 1.83 | 1.78 | **2.78** | 1.78 | 2.07 | **2.23** | 2.22 | 1.29 | 1.50 |
| Telco-83-offnet_rev_p1 | **1.70** | 1.65 | **1.97** | 1.32 | 1.37 | 1.67 | 1.66 | 1.10 | 1.10 |
| Telco-93-onnet_mins_p1 | **1.34** | 1.32 | **1.47** | 1.15 | 1.19 | 1.32 | 1.31 | 1.08 | 1.08 |
| Telco-95-onnet_mins_p3 | **1.62** | 1.58 | **1.88** | 1.33 | 1.43 | 1.57 | 1.56 | 1.19 | 1.21 |
| Telco-96-onnet_mins_p4 | **1.77** | 1.72 | **1.95** | 1.35 | 1.48 | 1.58 | 1.58 | 1.28 | 1.29 |

Table B.2: Compression ratio floats datasets. Best compression ratio per dataset in green, second best in yellow.

| | Avg. value bit width | Avg. exceptions / vector | Avg. bits / value ALP | Avg. bits / value GALP | Compression ratio ALP | Compression ratio GALP |
|---|---|---|---|---|---|---|
| **Average** | **14.08** | **89.21** | **18.32** | **18.82** | **1.94** | **1.88** |
| **Median** | **14.42** | **101.30** | **18.85** | **19.35** | **1.70** | **1.65** |
| CMSprovider-8-line_srvc_cnt | 10.99 | 8.49 | 11.44 | 11.94 | 2.80 | 2.68 |
| CityMaxCapita-13-lpf | 15.24 | 38.07 | 17.08 | 17.58 | 1.87 | 1.82 |
| MedPaymenT1-8-line_srvc_cnt | 11.59 | 10.08 | 12.11 | 12.61 | 2.64 | 2.54 |
| PanCreactomy1-8-line_srvc_cnt | 11.16 | 15.42 | 11.93 | 12.43 | 2.68 | 2.57 |
| Physicians-8-line_srvc_cnt | 11.58 | 10.09 | 12.11 | 12.61 | 2.64 | 2.54 |
| Telco-110-rechrg_total_load_p1 | 13.02 | 2.87 | 13.21 | 13.71 | 2.42 | 2.33 |
| Telco-111-rechrg_total_load_p2 | 11.78 | 6.15 | 12.12 | 12.62 | 2.64 | 2.54 |
| Telco-112-rechrg_total_load_p3 | 9.65 | 12.94 | 10.31 | 10.81 | 3.10 | 2.96 |
| Telco-113-rechrg_total_load_p4 | 8.61 | 12.38 | 9.25 | 9.75 | 3.46 | 3.28 |
| Telco-114-rechrg_total_load_p6 | 13.66 | 8.53 | 14.11 | 14.61 | 2.27 | 2.19 |
| Telco-151-total_mins_p1 | 16.52 | 187.09 | 25.35 | 25.85 | 1.26 | 1.24 |
| Telco-152-total_mins_p2 | 17.44 | 175.42 | 25.72 | 26.22 | 1.24 | 1.22 |
| Telco-153-total_mins_p3 | 15.24 | 106.89 | 20.30 | 20.80 | 1.58 | 1.54 |
| Telco-156-total_outgoing_min_p1 | 17.99 | 178.30 | 26.40 | 26.90 | 1.21 | 1.19 |
| Telco-157-total_outgoing_min_p2 | 16.05 | 190.21 | 25.02 | 25.52 | 1.28 | 1.25 |
| Telco-158-total_outgoing_min_p3 | 15.22 | 109.79 | 20.42 | 20.92 | 1.57 | 1.53 |
| Telco-159-total_outgoing_min_p4 | 13.71 | 112.35 | 19.03 | 19.53 | 1.68 | 1.64 |
| Telco-171-total_outgoing_rev_p1 | 21.14 | 188.84 | 30.05 | 30.55 | 1.06 | 1.05 |
| Telco-22-chrgd_rev_p1 | 16.63 | 101.92 | 21.46 | 21.96 | 1.49 | 1.46 |
| Telco-33-free_mins_p1 | 14.42 | 171.22 | 22.50 | 23.00 | 1.42 | 1.39 |
| Telco-35-free_mins_p3 | 9.38 | 170.87 | 17.44 | 17.94 | 1.83 | 1.78 |
| Telco-83-offnet_rev_p1 | 16.19 | 55.62 | 18.85 | 19.35 | 1.70 | 1.65 |
| Telco-93-onnet_mins_p1 | 16.47 | 155.31 | 23.81 | 24.31 | 1.34 | 1.32 |
| Telco-95-onnet_mins_p3 | 15.00 | 101.30 | 19.80 | 20.30 | 1.62 | 1.58 |
| Telco-96-onnet_mins_p4 | 13.34 | 100.11 | 18.09 | 18.59 | 1.77 | 1.72 |

Table B.3: Average value bit width, average number of exceptions per vector, and the bits per value and compression ratios for both ALP and G-ALP per double precision floating-point dataset. Value bit width refers to how many bits were used to encode values in the packed vectors, bits per value refers to the compressed data size in bits divided by total number of compressed values.

| | ALP | G-ALP | Thrust | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|---|
| **Average** | **42.65** | **141.60** | **20.59** | **2.43** | **4.52** | **4.19** | **0.88** | **5.58** | **30.44** | **34.75** |
| **Median** | **27.52** | **131.71** | **20.71** | **2.31** | **4.13** | **3.17** | **0.87** | **5.56** | **32.84** | **37.36** |
| CMSprovider-8-line_srvc_cnt | **76.43** | **173.60** | 20.77 | 1.84 | 2.99 | 2.84 | 0.89 | 5.67 | 18.92 | 26.60 |
| CityMaxCapita-13-lpf | **42.52** | **149.42** | 20.70 | 1.99 | 3.51 | 2.93 | 0.87 | 5.56 | 23.19 | 25.20 |
| MedPaymenT1-8-line_srvc_cnt | **69.20** | **172.20** | 20.88 | 1.85 | 2.96 | 2.88 | 0.89 | 5.67 | 18.84 | 26.08 |
| PanCreactomy1-8-line_srvc_cnt | **61.04** | **170.07** | 20.64 | 1.85 | 3.01 | 2.93 | 0.88 | 5.66 | 20.10 | 25.68 |
| Physicians-8-line_srvc_cnt | **69.40** | **171.84** | 20.79 | 1.85 | 2.95 | 2.91 | 0.89 | 5.65 | 18.86 | 25.77 |
| Telco-110-rechrg_total_load_p1 | **91.54** | **175.34** | 20.60 | 2.02 | 3.31 | 2.18 | 0.89 | 5.62 | 32.52 | 36.47 |
| Telco-111-rechrg_total_load_p2 | **88.09** | **175.84** | 20.79 | 2.27 | 3.71 | 2.38 | 0.90 | 5.65 | 34.32 | 38.55 |
| Telco-112-rechrg_total_load_p3 | **69.74** | **171.26** | 19.85 | 2.14 | 3.15 | 3.00 | 0.91 | 5.71 | 24.62 | 29.71 |
| Telco-113-rechrg_total_load_p4 | **74.46** | **172.40** | 20.78 | 2.48 | 3.67 | 3.14 | 0.92 | 5.71 | 28.81 | 37.36 |
| Telco-114-rechrg_total_load_p6 | **71.32** | **167.67** | 20.56 | 2.22 | 3.88 | 2.23 | 0.89 | 5.62 | 38.14 | 40.00 |
| Telco-151-total_mins_p1 | 19.37 | **111.96** | 20.41 | 3.07 | 6.23 | 6.87 | 0.87 | 5.53 | 35.78 | **39.36** |
| Telco-152-total_mins_p2 | 19.67 | **107.03** | 20.01 | 2.62 | 6.19 | 5.35 | 0.87 | 5.51 | 32.59 | **38.06** |
| Telco-153-total_mins_p3 | 25.11 | **130.06** | 20.84 | 2.68 | 4.83 | 4.51 | 0.86 | 5.53 | 33.57 | **37.77** |
| Telco-156-total_outgoing_min_p1 | 20.07 | **111.87** | 20.74 | 2.95 | 6.82 | 7.30 | 0.86 | 5.50 | 36.63 | **40.68** |
| Telco-157-total_outgoing_min_p2 | 19.18 | **111.73** | 20.73 | 3.28 | 6.78 | 8.43 | 0.86 | 5.53 | 35.77 | **38.66** |
| Telco-158-total_outgoing_min_p3 | 26.49 | **129.56** | 19.99 | 2.78 | 5.20 | 5.10 | 0.87 | 5.51 | 34.91 | **37.58** |
| Telco-159-total_outgoing_min_p4 | 26.21 | **131.09** | 20.72 | 3.16 | 5.54 | 6.24 | 0.87 | 5.55 | 32.84 | **36.14** |
| Telco-171-total_outgoing_rev_p1 | 19.24 | **107.40** | 20.41 | 3.21 | 8.20 | 8.82 | 0.86 | 5.42 | 38.79 | **42.61** |
| Telco-22-chrgd_rev_p1 | 27.52 | **129.16** | 20.72 | 2.75 | 5.03 | 4.30 | 0.87 | 5.54 | 37.65 | **39.23** |
| Telco-33-free_mins_p1 | 19.22 | **118.83** | 20.86 | 1.99 | 4.22 | 3.04 | 0.86 | 5.58 | 25.26 | **29.35** |
| Telco-35-free_mins_p3 | 18.53 | **127.92** | 20.21 | 2.13 | 4.22 | 3.17 | 0.88 | 5.66 | 22.43 | **29.18** |
| Telco-83-offnet_rev_p1 | 35.57 | **138.80** | 20.69 | 1.99 | 3.52 | 2.40 | 0.86 | 5.57 | 34.44 | **37.40** |
| Telco-93-onnet_mins_p1 | 21.59 | **119.34** | 20.71 | 2.63 | 4.96 | 4.62 | 0.86 | 5.51 | 36.41 | **38.06** |
| Telco-95-onnet_mins_p3 | 27.15 | **131.71** | 20.54 | 2.31 | 4.13 | 3.33 | 0.87 | 5.56 | 33.75 | **36.38** |
| Telco-96-onnet_mins_p4 | 27.69 | **133.90** | 20.83 | 2.60 | 4.12 | 3.75 | 0.87 | 5.56 | 31.76 | **36.82** |

Table B.4: Throughput (GB/s) filtering floats datasets. Best throughput per dataset in green, second best in yellow.

| | ALP | G-ALP | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|
| **Average** | **34.43** | **75.52** | **2.48** | **4.67** | **4.34** | **0.88** | **5.78** | **39.50** | **46.28** |
| **Median** | **24.20** | **73.20** | **2.38** | **4.32** | **3.27** | **0.88** | **5.76** | **43.09** | **49.56** |
| CMSprovider-8-line_srvc_cnt | **57.85** | **92.17** | 1.86 | 3.06 | 2.90 | 0.89 | 5.90 | 22.22 | 31.38 |
| CityMaxCapita-13-lpf | **35.26** | **80.81** | 2.03 | 3.59 | 3.00 | 0.87 | 5.77 | 30.54 | 31.72 |
| MedPaymenT1-8-line_srvc_cnt | **52.82** | **90.87** | 1.88 | 3.03 | 2.93 | 0.89 | 5.89 | 24.00 | 33.10 |
| PanCreactomy1-8-line_srvc_cnt | **48.05** | **90.48** | 1.88 | 3.03 | 2.94 | 0.89 | 5.60 | 22.29 | 31.79 |
| Physicians-8-line_srvc_cnt | **52.83** | **91.07** | 1.88 | 3.01 | 2.96 | 0.88 | 5.90 | 23.05 | 31.89 |
| Telco-110-rechrg_total_load_p1 | **66.20** | **89.53** | 2.05 | 3.39 | 2.26 | 0.89 | 5.83 | 44.08 | 48.05 |
| Telco-111-rechrg_total_load_p2 | **64.25** | **91.55** | 2.31 | 3.77 | 2.45 | 0.90 | 5.85 | 44.06 | 51.84 |
| Telco-112-rechrg_total_load_p3 | **53.52** | **93.71** | 2.17 | 3.24 | 3.08 | 0.92 | 5.94 | 31.25 | 38.02 |
| Telco-113-rechrg_total_load_p4 | **56.98** | **96.30** | 2.55 | 3.70 | 3.27 | 0.92 | 5.90 | 36.62 | 48.44 |
| Telco-114-rechrg_total_load_p6 | 53.61 | **86.98** | 2.26 | 3.93 | 2.27 | 0.89 | 5.83 | 51.15 | **55.28** |
| Telco-151-total_mins_p1 | 17.68 | **57.03** | 3.14 | 6.51 | 7.34 | 0.87 | 5.73 | 47.74 | **54.33** |
| Telco-152-total_mins_p2 | 18.14 | **57.31** | 2.68 | 6.53 | 5.56 | 0.86 | 5.71 | 40.07 | **50.40** |
| Telco-153-total_mins_p3 | 23.94 | **71.78** | 2.72 | 5.01 | 4.66 | 0.86 | 5.76 | 43.01 | **49.94** |
| Telco-156-total_outgoing_min_p1 | 18.39 | **56.76** | 3.03 | 7.11 | 7.67 | 0.86 | 5.65 | 49.02 | **56.88** |
| Telco-157-total_outgoing_min_p2 | 17.23 | **55.66** | 3.35 | 6.86 | 8.99 | 0.87 | 5.73 | 47.65 | **52.83** |
| Telco-158-total_outgoing_min_p3 | 23.63 | **71.65** | 2.84 | 5.36 | 5.25 | 0.88 | 5.74 | 44.89 | **49.33** |
| Telco-159-total_outgoing_min_p4 | 23.49 | **73.02** | 3.23 | 5.78 | 6.54 | 0.88 | 5.76 | 42.06 | **49.56** |
| Telco-171-total_outgoing_rev_p1 | 17.67 | **53.62** | 3.28 | 8.67 | 9.34 | 0.86 | 5.72 | 50.14 | **58.85** |
| Telco-22-chrgd_rev_p1 | 24.20 | **70.15** | 2.83 | 5.24 | 4.41 | 0.88 | 5.75 | 49.88 | **53.67** |
| Telco-33-free_mins_p1 | 18.11 | **61.74** | 2.03 | 4.35 | 3.11 | 0.87 | 5.79 | 33.76 | **38.38** |
| Telco-35-free_mins_p3 | 17.94 | **68.28** | 2.15 | 4.33 | 3.15 | 0.89 | 5.78 | 25.98 | **35.57** |
| Telco-83-offnet_rev_p1 | 30.37 | **76.45** | 2.04 | 3.62 | 2.44 | 0.87 | 5.78 | 45.68 | **52.55** |
| Telco-93-onnet_mins_p1 | 19.76 | **62.22** | 2.66 | 5.03 | 4.76 | 0.87 | 5.74 | 49.08 | **53.82** |
| Telco-95-onnet_mins_p3 | 24.15 | **73.20** | 2.38 | 4.24 | 3.41 | 0.88 | 5.76 | 43.09 | **48.81** |
| Telco-96-onnet_mins_p4 | 24.70 | **75.70** | 2.67 | 4.32 | 3.84 | 0.88 | 5.69 | 46.12 | **50.46** |

Table B.5: Throughput (GB/s) fully decompressing floats datasets. Best throughput per dataset in green, second best in yellow.

# Appendix C

# Double precision floating-point datasets results

| | ALP | G-ALP | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|
| **Average** | **19.26** | **19.51** | **18.60** | **26.31** | **27.03** | **41.66** | **41.84** | **49.74** | **55.12** |
| **Median** | **16.02** | **16.27** | **17.22** | **26.78** | **26.60** | **47.52** | **47.70** | **59.47** | **64.11** |
| arade4 | **24.94** | **25.19** | 35.71 | 43.19 | 50.43 | 59.69 | 59.88 | 64.11 | 64.11 |
| astro_mhd_f64 | 3.20 | 3.45 | **2.39** | **3.09** | 5.39 | 13.58 | 13.76 | 3.28 | 64.11 |
| basel_temp_f | **29.56** | 29.81 | **24.51** | 31.53 | 34.75 | 59.79 | 59.98 | 64.11 | 64.11 |
| basel_wind_f | 29.79 | 30.04 | **20.62** | **28.91** | 29.14 | 59.13 | 59.32 | 64.11 | 64.11 |
| bird_migration_f | **20.73** | **20.98** | 22.34 | 32.26 | 31.83 | 57.77 | 57.95 | 62.90 | 64.11 |
| bitcoin_f | 26.84 | 27.09 | **13.64** | **16.49** | 28.19 | 58.14 | 58.32 | 64.11 | 64.11 |
| bitcoin_transactions_f | **36.22** | **36.47** | 39.85 | 47.06 | 54.21 | 60.15 | 60.33 | 64.03 | 64.11 |
| city_temperature_f | **10.73** | **10.98** | 17.93 | 27.60 | 26.60 | 33.09 | 33.27 | 61.16 | 63.47 |
| cms1 | 36.23 | 36.48 | **26.02** | 33.80 | **32.32** | 53.05 | 53.23 | 49.68 | 63.88 |
| cms25 | **41.09** | **41.34** | 57.97 | 61.39 | 61.65 | 60.43 | 60.61 | 63.99 | 63.99 |
| cms9 | **12.18** | **12.43** | 15.13 | 24.82 | 20.50 | 19.04 | 19.22 | 54.72 | 51.42 |
| food_prices | 28.93 | 29.18 | **17.83** | 26.78 | **25.56** | 35.85 | 36.04 | 49.79 | 60.72 |
| gov10 | **30.63** | 30.88 | **27.85** | 38.03 | 37.65 | 41.16 | 41.34 | 59.83 | 57.91 |
| gov26 | **0.38** | 0.63 | **0.21** | 0.74 | 3.23 | 8.23 | 8.43 | 0.50 | 0.52 |
| gov30 | **7.02** | 7.27 | **4.25** | 8.92 | 8.13 | 12.87 | 13.06 | 9.22 | 8.61 |
| gov31 | **2.60** | 2.85 | **1.57** | 4.40 | 4.99 | 10.40 | 10.59 | 4.18 | 4.56 |
| gov40 | 0.99 | 1.24 | **0.40** | 1.16 | 3.50 | 8.57 | 8.77 | **0.94** | 0.99 |
| medicare1 | 39.37 | 39.62 | **31.06** | **38.59** | 38.71 | 53.87 | 54.06 | 57.74 | 64.09 |
| medicare9 | **12.07** | **12.32** | 15.37 | 24.89 | 20.73 | 19.08 | 19.25 | 55.76 | 51.45 |
| neon_air_pressure | **16.71** | 16.96 | **15.89** | 23.99 | 24.01 | 55.44 | 55.62 | 46.35 | 64.11 |
| neon_bio_temp_c | **9.96** | **10.21** | 17.22 | 27.65 | 29.08 | 45.59 | 45.77 | 62.39 | 64.11 |
| neon_dew_point_temp | **13.57** | **13.82** | 25.06 | 32.91 | 37.05 | 56.61 | 56.80 | 64.11 | 64.11 |
| neon_pm10_dust | **8.63** | 8.88 | **7.96** | 18.81 | 13.98 | 43.68 | 43.87 | 39.07 | 64.10 |
| neon_wind_dir | **16.02** | **16.27** | 25.07 | 33.38 | 36.77 | 48.62 | 48.81 | 64.11 | 64.11 |
| nyc29 | 40.37 | 40.62 | **25.39** | 34.17 | **33.76** | 53.95 | 54.14 | 54.14 | 64.11 |
| phone_gyro_f64 | 41.81 | 42.06 | **27.54** | **35.65** | 40.26 | 58.10 | 58.28 | 64.11 | 64.11 |
| spain_gas_price_f64 | **11.36** | **11.61** | 13.63 | 22.19 | 23.67 | 48.16 | 48.34 | 64.10 | 64.11 |
| ssd_hdd_benchmarks_f | **16.15** | 16.40 | **13.99** | 23.77 | 22.27 | 32.90 | 33.08 | 59.47 | 64.11 |
| stocks_de | **10.83** | 11.08 | **10.81** | 24.19 | 21.46 | 47.52 | 47.70 | 62.09 | 63.97 |
| stocks_uk | **11.29** | 11.54 | **10.85** | 22.38 | 19.84 | 33.69 | 33.87 | 51.04 | 63.22 |
| stocks_usa_c | **6.93** | **7.18** | 8.64 | 22.89 | 18.41 | 43.26 | 43.44 | 56.72 | 64.11 |

Table C.1: Bits per value for each dataset and compressor. Lower is better, 64 bits per value is the upper bound for compressing data. Best bits per value per dataset in green, second best in yellow.

| | ALP | G-ALP | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|
| **Average** | **12.40** | **9.68** | **20.63** | **7.85** | **4.21** | **2.25** | **2.23** | **8.56** | **7.62** |
| **Median** | **4.00** | **3.93** | **3.72** | **2.39** | **2.41** | **1.35** | **1.34** | **1.08** | **1.00** |
| arade4 | **2.57** | **2.54** | 1.79 | 1.48 | 1.27 | 1.07 | 1.07 | 1.00 | 1.00 |
| astro_mhd_f64 | 20.01 | 18.56 | **26.78** | **20.73** | 11.88 | 4.71 | 4.65 | 19.50 | 1.00 |
| basel_temp_f | **2.16** | 2.15 | **2.61** | 2.03 | 1.84 | 1.07 | 1.07 | 1.00 | 1.00 |
| basel_wind_f | 2.15 | 2.13 | **3.10** | **2.21** | 2.20 | 1.08 | 1.08 | 1.00 | 1.00 |
| bird_migration_f | **3.09** | **3.05** | 2.87 | 1.98 | 2.01 | 1.11 | 1.10 | 1.02 | 1.00 |
| bitcoin_f | 2.38 | 2.36 | **4.69** | **3.88** | 2.27 | 1.10 | 1.10 | 1.00 | 1.00 |
| bitcoin_transactions_f | **1.77** | **1.76** | 1.61 | 1.36 | 1.18 | 1.06 | 1.06 | 1.00 | 1.00 |
| city_temperature_f | **5.96** | **5.83** | 3.57 | 2.32 | 2.41 | 1.93 | 1.92 | 1.05 | 1.01 |
| cms1 | 1.77 | 1.75 | **2.46** | 1.89 | **1.98** | 1.21 | 1.20 | 1.29 | 1.00 |
| cms25 | **1.56** | **1.55** | 1.10 | 1.04 | 1.04 | 1.06 | 1.06 | 1.00 | 1.00 |
| cms9 | **5.25** | **5.15** | 4.23 | 2.58 | 3.12 | 3.36 | 3.33 | 1.17 | 1.24 |
| food_prices | 2.21 | 2.19 | **3.59** | 2.39 | **2.50** | 1.79 | 1.78 | 1.29 | 1.05 |
| gov10 | **2.09** | 2.07 | **2.30** | 1.68 | 1.70 | 1.56 | 1.55 | 1.07 | 1.11 |
| gov26 | **166.47** | 100.87 | **302.78** | 86.63 | 19.84 | 7.78 | 7.59 | 126.96 | 122.36 |
| gov30 | **9.12** | 8.81 | **15.05** | 7.18 | 7.87 | 4.97 | 4.90 | 6.94 | 7.43 |
| gov31 | **24.65** | 22.48 | **40.66** | 14.55 | 12.83 | 6.15 | 6.04 | 15.31 | 14.05 |
| gov40 | 64.35 | 51.43 | **159.79** | 55.31 | 18.30 | 7.47 | 7.30 | **67.82** | 64.67 |
| medicare1 | 1.63 | 1.62 | **2.06** | **1.66** | 1.65 | 1.19 | 1.18 | 1.11 | 1.00 |
| medicare9 | **5.30** | **5.19** | 4.16 | 2.57 | 3.09 | 3.35 | 3.32 | 1.15 | 1.24 |
| neon_air_pressure | **3.83** | 3.77 | **4.03** | 2.67 | 2.67 | 1.15 | 1.15 | 1.38 | 1.00 |
| neon_bio_temp_c | **6.43** | **6.27** | 3.72 | 2.31 | 2.20 | 1.40 | 1.40 | 1.03 | 1.00 |
| neon_dew_point_temp | **4.72** | **4.63** | 2.55 | 1.94 | 1.73 | 1.13 | 1.13 | 1.00 | 1.00 |
| neon_pm10_dust | **7.42** | 7.21 | **8.04** | 3.40 | 4.58 | 1.47 | 1.46 | 1.64 | 1.00 |
| neon_wind_dir | **4.00** | **3.93** | 2.55 | 1.92 | 1.74 | 1.32 | 1.31 | 1.00 | 1.00 |
| nyc29 | 1.59 | 1.58 | **2.52** | 1.87 | **1.90** | 1.19 | 1.18 | 1.18 | 1.00 |
| phone_gyro_f64 | 1.53 | 1.52 | **2.32** | **1.80** | 1.59 | 1.10 | 1.10 | 1.00 | 1.00 |
| spain_gas_price_f64 | **5.63** | **5.51** | 4.70 | 2.88 | 2.70 | 1.33 | 1.32 | 1.00 | 1.00 |
| ssd_hdd_benchmarks_f | **3.96** | 3.90 | **4.57** | 2.69 | 2.87 | 1.95 | 1.93 | 1.08 | 1.00 |
| stocks_de | **5.91** | 5.77 | **5.92** | 2.65 | 2.98 | 1.35 | 1.34 | 1.03 | 1.00 |
| stocks_uk | **5.67** | 5.55 | **5.90** | 2.86 | 3.23 | 1.90 | 1.89 | 1.25 | 1.01 |
| stocks_usa_c | **9.24** | **8.92** | 7.40 | 2.80 | 3.48 | 1.48 | 1.47 | 1.13 | 1.00 |

Table C.2: Compression ratio doubles datasets. Best compression ratio per dataset in green, second best in yellow.

| | Avg. value bit width | Avg. exceptions / vector | Avg. bits / value ALP | Avg. bits / value GALP | Compression ratio ALP | Compression ratio GALP |
|---|---|---|---|---|---|---|
| **Average** | **16.84** | **29.91** | **19.26** | **19.51** | **12.40** | **9.68** |
| **Median** | **15.93** | **8.38** | **16.02** | **16.27** | **4.00** | **3.93** |
| arade4 | 24.21 | 8.15 | 24.94 | 25.19 | 2.57 | 2.54 |
| astro_mhd_f64 | 0.00 | 39.84 | 3.20 | 3.45 | 20.01 | 18.56 |
| basel_temp_f | 27.14 | 29.85 | 29.56 | 29.81 | 2.16 | 2.15 |
| basel_wind_f | 28.32 | 17.70 | 29.79 | 30.04 | 2.15 | 2.13 |
| bird_migration_f | 20.12 | 6.77 | 20.73 | 20.98 | 3.09 | 3.05 |
| bitcoin_f | 26.10 | 8.38 | 26.84 | 27.09 | 2.38 | 2.36 |
| bitcoin_transactions_f | 30.00 | 78.46 | 36.22 | 36.47 | 1.77 | 1.76 |
| city_temperature_f | 10.64 | 0.11 | 10.73 | 10.98 | 5.96 | 5.83 |
| cms1 | 25.27 | 139.15 | 36.23 | 36.48 | 1.77 | 1.75 |
| cms25 | 40.41 | 7.72 | 41.09 | 41.34 | 1.56 | 1.55 |
| cms9 | 9.91 | 28.03 | 12.18 | 12.43 | 5.25 | 5.15 |
| food_prices | 18.88 | 127.53 | 28.93 | 29.18 | 2.21 | 2.19 |
| gov10 | 26.30 | 54.27 | 30.63 | 30.88 | 2.09 | 2.07 |
| gov26 | 0.06 | 2.99 | 0.38 | 0.63 | 166.47 | 100.87 |
| gov30 | 2.80 | 52.93 | 7.02 | 7.27 | 9.12 | 8.81 |
| gov31 | 1.88 | 8.10 | 2.60 | 2.85 | 24.65 | 22.48 |
| gov40 | 0.01 | 11.54 | 0.99 | 1.24 | 64.35 | 51.43 |
| medicare1 | 28.05 | 143.78 | 39.37 | 39.62 | 1.63 | 1.62 |
| medicare9 | 11.25 | 9.42 | 12.07 | 12.32 | 5.30 | 5.19 |
| neon_air_pressure | 15.95 | 8.66 | 16.71 | 16.96 | 3.83 | 3.77 |
| neon_bio_temp_c | 9.86 | 0.20 | 9.96 | 10.21 | 6.43 | 6.27 |
| neon_dew_point_temp | 12.88 | 7.71 | 13.57 | 13.82 | 4.72 | 4.63 |
| neon_pm10_dust | 8.22 | 4.12 | 8.63 | 8.88 | 7.42 | 7.21 |
| neon_wind_dir | 15.93 | 0.00 | 16.02 | 16.27 | 4.00 | 3.93 |
| nyc29 | 39.73 | 7.10 | 40.37 | 40.62 | 1.59 | 1.58 |
| phone_gyro_f64 | 33.80 | 101.50 | 41.81 | 42.06 | 1.53 | 1.52 |
| spain_gas_price_f64 | 9.71 | 19.99 | 11.36 | 11.61 | 5.63 | 5.51 |
| ssd_hdd_benchmarks_f | 16.05 | 0.11 | 16.15 | 16.40 | 3.96 | 3.90 |
| stocks_de | 10.57 | 2.27 | 10.83 | 11.08 | 5.91 | 5.77 |
| stocks_uk | 11.14 | 0.84 | 11.29 | 11.54 | 5.67 | 5.55 |
| stocks_usa_c | 6.84 | 0.00 | 6.93 | 7.18 | 9.24 | 8.92 |

Table C.3: Average value bit width, average number of exceptions per vector, and the bits per value and compression ratios for both ALP and G-ALP per double precision floating-point dataset. Value bit width refers to how many bits were used to encode values in the packed vectors, bits per value refers to the compressed data size in bits divided by total number of compressed values.

| | ALP | G-ALP | Thrust | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|---|
| **Average** | **102.19** | **146.35** | **37.98** | **4.83** | **7.72** | **6.42** | **0.91** | **5.64** | **33.51** | **44.84** |
| **Median** | **106.61** | **148.34** | **37.77** | **2.28** | **5.10** | **4.63** | **0.89** | **5.53** | **32.81** | **44.74** |
| arade4 | **106.61** | **148.44** | 37.50 | 2.26 | 4.61 | 2.61 | 0.87 | 5.46 | 43.54 | 44.23 |
| astro_mhd_f64 | **112.92** | **148.16** | 38.35 | 14.08 | 24.42 | 18.80 | 0.97 | 6.03 | 24.18 | 44.65 |
| basel_temp_f | **69.10** | **147.60** | 37.73 | 2.26 | 4.83 | 3.36 | 0.87 | 5.45 | 43.66 | 44.83 |
| basel_wind_f | **91.00** | **148.01** | 38.84 | 2.30 | 5.11 | 4.18 | 0.86 | 5.48 | 43.88 | 44.81 |
| bird_migration_f | **109.83** | **148.24** | 37.33 | 2.25 | 4.81 | 3.75 | 0.87 | 5.45 | 26.82 | 44.86 |
| bitcoin_f | **105.61** | **148.36** | 37.38 | 4.88 | 10.17 | 5.14 | 0.88 | 5.47 | 44.00 | 44.66 |
| bitcoin_transactions_f | **48.36** | **143.98** | 39.20 | 2.26 | 4.68 | 2.76 | 0.86 | 5.47 | 40.60 | 44.74 |
| city_temperature_f | **142.72** | **148.56** | 36.99 | 1.81 | 4.91 | 5.39 | 0.94 | 5.65 | 21.12 | 27.68 |
| cms1 | 36.35 | **129.62** | 37.67 | 2.92 | 5.94 | 5.06 | 0.87 | 5.49 | 22.17 | **44.57** |
| cms25 | **100.64** | **147.91** | 37.47 | 3.62 | 16.46 | 15.57 | 0.87 | 5.50 | 39.99 | 44.54 |
| cms9 | **79.98** | **148.11** | 37.38 | 2.24 | 5.15 | 7.71 | 0.96 | 5.80 | 19.88 | 29.01 |
| food_prices | **41.58** | **143.82** | 37.71 | 2.25 | 4.93 | 3.99 | 0.90 | 5.49 | 21.51 | 31.81 |
| gov10 | **57.16** | **146.30** | 38.39 | 1.92 | 4.29 | 2.71 | 0.88 | 5.61 | 26.22 | 34.89 |
| gov26 | **128.22** | **148.94** | 37.68 | 28.60 | 21.88 | 18.91 | 1.01 | 6.43 | 54.85 | 69.12 |
| gov30 | **67.10** | **147.92** | 37.49 | 7.93 | 9.97 | 8.85 | 0.98 | 5.90 | 44.68 | 54.17 |
| gov31 | **133.42** | **148.35** | 37.69 | 12.89 | 14.50 | 12.81 | 0.99 | 6.07 | 49.76 | 64.74 |
| gov40 | **125.35** | **148.70** | 38.43 | 22.36 | 21.24 | 15.49 | 1.01 | 6.40 | 54.37 | 68.63 |
| medicare1 | 33.71 | **130.91** | 38.34 | 2.50 | 5.45 | 4.00 | 0.88 | 5.49 | 23.10 | **44.80** |
| medicare9 | **109.66** | **148.38** | 37.95 | 2.25 | 5.05 | 8.22 | 0.96 | 5.81 | 20.07 | 29.06 |
| neon_air_pressure | **106.23** | **148.54** | 37.44 | 2.98 | 5.71 | 5.11 | 0.88 | 5.53 | 23.97 | 44.82 |
| neon_bio_temp_c | **146.79** | **148.77** | 38.68 | 1.99 | 4.81 | 3.31 | 0.90 | 5.53 | 33.99 | 44.76 |
| neon_dew_point_temp | **113.79** | **148.54** | 37.84 | 2.20 | 4.76 | 2.73 | 0.87 | 5.49 | 43.94 | 44.73 |
| neon_pm10_dust | **130.29** | **148.32** | 37.70 | 2.66 | 5.58 | 5.65 | 0.90 | 5.53 | 21.07 | 44.64 |
| neon_wind_dir | **148.69** | **148.65** | 38.17 | 2.14 | 4.95 | 3.38 | 0.86 | 5.54 | 44.13 | 44.77 |
| nyc29 | **100.63** | **147.82** | 38.44 | 2.67 | 5.17 | 3.74 | 0.89 | 5.51 | 21.70 | 44.73 |
| phone_gyro_f64 | **47.81** | **133.21** | 37.97 | 2.48 | 5.01 | 3.45 | 0.87 | 5.49 | 44.04 | 44.84 |
| spain_gas_price_f64 | **101.52** | **148.34** | 38.43 | 2.65 | 4.93 | 3.67 | 0.89 | 5.48 | 43.66 | 44.79 |
| ssd_hdd_benchmarks_f | **146.01** | **148.54** | 37.77 | 1.93 | 5.10 | 4.75 | 0.91 | 5.60 | 19.72 | 44.89 |
| stocks_de | **133.86** | **148.53** | 39.00 | 2.13 | 4.90 | 4.52 | 0.89 | 5.51 | 32.81 | 44.39 |
| stocks_uk | **143.77** | **148.61** | 38.87 | 2.09 | 5.19 | 4.63 | 0.90 | 5.58 | 24.00 | 42.01 |
| stocks_usa_c | **149.05** | **148.75** | 37.65 | 2.28 | 4.92 | 4.87 | 0.91 | 5.56 | 21.52 | 44.81 |

Table C.4: Throughput (GB/s) filtering doubles datasets. Best throughput per dataset in green, second best in yellow.

| | ALP | G-ALP | nv-zstd | nv-LZ4 | nv-Snappy | nv-Deflate | nv-GDeflate | Bitcomp | BitcompSparse |
|---|---|---|---|---|---|---|---|---|---|
| **Average** | **75.78** | **95.26** | **5.39** | **8.49** | **6.99** | **0.92** | **5.92** | **45.15** | **65.84** |
| **Median** | **78.24** | **100.32** | **2.34** | **5.31** | **4.83** | **0.90** | **5.78** | **42.34** | **63.80** |
| arade4 | **75.44** | **91.98** | 2.31 | 4.82 | 2.62 | 0.89 | 5.73 | 62.27 | 63.92 |
| astro_mhd_f64 | **91.06** | **121.62** | 15.63 | 29.24 | 21.72 | 0.99 | 6.31 | 28.78 | 63.89 |
| basel_temp_f | 51.79 | **84.93** | 2.32 | 5.02 | 3.50 | 0.88 | 5.72 | 62.10 | **64.00** |
| basel_wind_f | **65.76** | **86.79** | 2.37 | 5.29 | 4.32 | 0.88 | 5.73 | 61.66 | 63.37 |
| bird_migration_f | **78.70** | **95.34** | 2.32 | 4.99 | 3.89 | 0.89 | 5.73 | 32.65 | 63.91 |
| bitcoin_f | **73.66** | **90.26** | 5.11 | 10.99 | 5.44 | 0.89 | 5.74 | 62.15 | 63.65 |
| bitcoin_transactions_f | 36.82 | **69.54** | 2.32 | 4.86 | 2.85 | 0.88 | 5.73 | 55.37 | **63.85** |
| city_temperature_f | **102.16** | **107.02** | 1.86 | 5.14 | 5.66 | 0.95 | 5.91 | 24.48 | 34.26 |
| cms1 | 29.86 | **59.55** | 3.02 | 6.22 | 5.27 | 0.89 | 5.76 | 26.06 | **63.20** |
| cms25 | **68.71** | **79.07** | 3.76 | 18.63 | 17.63 | 0.88 | 5.72 | 54.34 | 63.39 |
| cms9 | **60.83** | **96.80** | 2.31 | 5.37 | 8.20 | 0.98 | 6.09 | 23.12 | 36.09 |
| food_prices | 33.80 | **66.03** | 2.30 | 5.19 | 4.28 | 0.92 | 5.85 | 25.08 | **40.45** |
| gov10 | 44.36 | **77.93** | 1.98 | 4.45 | 2.82 | 0.90 | 5.82 | 31.74 | **45.56** |
| gov26 | 117.51 | **131.29** | 35.67 | 25.81 | 21.95 | 1.03 | 6.81 | 86.81 | **128.23** |
| gov30 | 53.14 | **102.71** | 8.53 | 10.87 | 9.76 | 1.00 | 6.26 | 63.44 | **84.45** |
| gov31 | 110.11 | **121.33** | 14.50 | 16.34 | 14.13 | 1.01 | 6.41 | 75.64 | **112.91** |
| gov40 | 110.62 | **127.48** | 26.90 | 24.92 | 17.62 | 1.03 | 6.69 | 84.96 | **126.87** |
| medicare1 | 27.70 | **58.14** | 2.58 | 5.70 | 4.15 | 0.90 | 5.75 | 27.25 | **63.69** |
| medicare9 | **80.24** | **101.18** | 2.31 | 5.31 | 8.87 | 0.98 | 6.10 | 23.11 | 36.13 |
| neon_air_pressure | **78.24** | **98.88** | 3.08 | 6.01 | 5.33 | 0.89 | 5.75 | 28.46 | 63.97 |
| neon_bio_temp_c | **102.56** | **108.15** | 2.04 | 5.04 | 3.45 | 0.91 | 5.80 | 43.90 | 63.70 |
| neon_dew_point_temp | **81.68** | **101.92** | 2.27 | 5.00 | 2.90 | 0.88 | 5.73 | 61.72 | 64.03 |
| neon_pm10_dust | **94.27** | **108.01** | 2.75 | 5.85 | 5.96 | 0.91 | 5.81 | 24.81 | 63.25 |
| neon_wind_dir | **99.93** | **101.06** | 2.20 | 5.20 | 3.42 | 0.88 | 5.78 | 61.95 | 63.89 |
| nyc29 | **68.81** | **79.35** | 2.75 | 5.41 | 3.90 | 0.90 | 5.76 | 25.34 | 64.05 |
| phone_gyro_f64 | 37.38 | 61.62 | 2.58 | 5.26 | 3.62 | 0.89 | 5.73 | **62.13** | **63.87** |
| spain_gas_price_f64 | **74.90** | **100.32** | 2.74 | 5.19 | 3.83 | 0.91 | 5.78 | 61.22 | 63.87 |
| ssd_hdd_benchmarks_f | **95.22** | **100.44** | 1.99 | 5.34 | 4.95 | 0.93 | 5.90 | 22.93 | 63.70 |
| stocks_de | **95.02** | **105.60** | 2.20 | 5.11 | 4.76 | 0.90 | 5.78 | 42.34 | 62.85 |
| stocks_uk | **100.05** | **105.85** | 2.16 | 5.38 | 4.83 | 0.92 | 5.87 | 28.48 | 58.26 |
| stocks_usa_c | **108.77** | **112.80** | 2.34 | 5.16 | 5.05 | 0.92 | 5.82 | 25.21 | 63.80 |

Table C.5: Throughput (GB/s) fully decompressing doubles datasets. Best throughput per dataset in green, second best in yellow.