Tile-based Lightweight Integer Compression in GPU

Anil Shanbhag*
Massachusetts Institute of Technology
anil@csail.mit.edu

Bobbi W. Yogatama*
University of Wisconsin-Madison
bwyogatama@cs.wisc.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Samuel Madden
Massachusetts Institute of Technology
madden@csail.mit.edu

ABSTRACT

GPUs are increasingly used for high-performance and interactive data analytics workloads due to their capability to accelerate computation using massive parallelism. A key constraint of GPU-based data analytics today is the limited memory capacity in GPU devices.

Data compression is a powerful technique that can mitigate the capacity limitation in two ways: (1) fitting more data into GPU memory and (2) speeding up data transfer between CPU and GPU. However, compression schemes for GPU today are still limited in compression ratio and/or decompression speed. We identify two limiting factors of existing approaches. First, existing decompression solutions require multiple passes of scanning the global memory to decode layers of compression schemes, incurring significant memory traffic and hurting performance. We present the *tile-based decompression* model to decompress encoded data in a single pass over global memory and inline with query execution. Second, we develop an efficient implementation of bit-packing-based compression schemes and their optimization techniques in the context of GPU. Our evaluation shows that our schemes can achieve similar compression rates to the best state-of-the-art compression schemes in GPU (i.e., nvCOMP) while being 2.2× and 2.6× faster in decompression speed and query running time.

CCS CONCEPTS

• **Theory of computation** → **Data compression**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

KEYWORDS

GPU data compression, GPU data analytics, bit-packing

ACM Reference Format:
Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526132

1 INTRODUCTION

In the past decade, special-purpose graphics processing units (GPUs) have evolved into general-purpose computing devices, with general-purpose parallel programming models, such as CUDA [2] and OpenCL [8]. Because of GPUs' high compute power, they have seen significant adoption in deep learning and high performance computing [3]. GPUs also have great potential to accelerate memory-bound applications such as database systems, because GPUs utilize High-Bandwidth Memory (HBM), a new class of RAM that has significantly higher bandwidth compared to traditional DDR RAM used with CPUs. A single modern GPU can have up to 80 GB of HBM which is capable of delivering up to 2 TBps of memory bandwidth and 19.5 Tflops of compute compared to 100GBps of memory bandwidth and < 1 Tflops on a single CPU. This rise in memory capacity, coupled with the ability to equip a modern server with several GPUs (up to 20), means that it's possible to have hundreds of gigabytes of GPU memory on a modern server.

Several commercial systems, including Omnisci [7], Kinetica [4], and BlazingDB [1], aim to provide real-time analytics capabilities by using GPUs to store a large fraction (or all) of the working set. A key constraint in these systems is the GPU memory capacity. Currently, GPUs have at most 80 GB of memory which is used both to cache the working set and as scratch memory for query execution. Therefore, to accommodate working set larger than GPU memory capacity, sharding between CPU and GPU or between multiple GPUs is necessary [32, 36]. However, this approach will incur performance penalty due to slower communication across PCIe. Data compression can play a critical role to address this issue by achieving two goals: (1) Fit in more working set in a single GPU memory and (2) Reduce the data transfer time across PCIe by transferring the compressed data instead. Furthermore, data compression is also applicable beyond the database field as link bandwidth is often the bottleneck in many applications that use GPUs such as machine learning and image processing.

We identify two major limitations in GPU data compression solutions today:

1) Cascading Decompression
GPU-based systems [18, 30] have looked at frame-of-reference (FOR) [21, 52], delta coding (DELTA) [29], dictionary compression (DICT) [10, 52], run-length encoding (RLE) [10], and null suppression (NS) [10]. To minimize the size of the compressed data, existing systems cascade multiple compression schemes such that the output of one scheme is the input to another scheme. The database engine decompresses one layer at a time. Such a cascading decompression strategy leads to suboptimal performance as multiple GPU kernels are launched and each requires reading from and writing data back to the global memory. This causes high memory traffic which may lead to much worse performance compared to a design that does not use any compression.

In this paper, we improve decompression performance by treating a *thread block* as the basic decompression unit, that decodes one block of encoded entries. This way, we are able to cache a block of data in on-chip caches and inline multiple decoding steps into

*Both authors contributed equally (listed alphabetically)

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3526132

a single kernel, resulting in a single pass over the data. Moreover, we can even inline the decompression with query execution, resulting in a single round-trip to the global memory for the whole decompression step and query execution.

We call this *tile-based decompression model* which is inspired by Crystal [40]. *Tile-based decompression* allows us to decode at close to memory bandwidth speed, resulting in a very low decompression overhead over the previous compression work for GPU. It also eliminates the need for sophisticated compression planners used by past works [18, 30], since instead of balancing the trade-off between decompression time and compression ratio, we can simply choose the scheme with the best compression ratio — all schemes achieve similar performance. To the best of our knowledge, this is the first paper to decompress cascaded compression schemes in a single kernel pass and inline with the query execution.

2) Efficient Decompression of Bit-Packed Schemes
Besides the organization of different compression schemes, the decompression speed of individual schemes is also limited in GPU today. Prior works on CPU compression [16] have shown that bit-aligned packing compression schemes manage to achieve better compression ratio compared to byte- and word-aligned packing. Supporting efficient bit-level packing in GPU is challenging due to the SIMT programming model and the relatively limited instruction set to perform bit-level alignment, which is not a problem in CPU compression schemes due to more powerful CPU instructions.

In this paper, we design optimized bit-packing based compression schemes and their optimization techniques in the context of GPU. Specifically, we introduce three new bit-packing based compression schemes: `GPU-FOR` does bit-packing in conjunction with Frame-Of-Reference (FOR) and work well with uniform data and can handle skew; `GPU-DFOR` uses delta encoding with bit-packing and FOR, targeting sorted or semi-sorted data; `GPU-RFOR` uses RLE encoding with bit-packing and FOR, targeting data with high average run length. These schemes are designed to offer improved compression ratios while still being able to decode data in parallel across thousands of threads at close to memory bandwidth speeds. Overall, our compression schemes can be decoded 2.2× faster than previous implementations.

We integrate the tile-based decompression model and `GPU-FOR`, `GPU-DFOR`, and `GPU-RFOR` into the Crystal framework [40], an open-source highly optimized GPU data analytics engine. We encapsulate decompression into a device function that enables programmers to change a kernel operating on an uncompressed array to a compressed column with a single line of code. Our compression schemes target integer, decimal, and dictionary-encoded strings and support all query operations that run on these data types. In data analytics workloads, prior works [14, 34, 40] and commercial systems [7] typically apply dictionary encoding on top of string columns to encode them to integers.

In summary this paper makes the following contributions:

- We introduce *tile-based decompression*, a decompression strategy that allows us to decode the data in a single pass at close to memory bandwidth speed and inline with query execution.
- We present three optimized bit-packing based compression schemes (`GPU-FOR`, `GPU-DFOR`, and `GPU-RFOR`) that can be used to store data compactly on the GPU.
- We present an integration of the decompression routines into the `Crystal` framework and demonstrate ease of use.
- We present an evaluation[1] on multiple synthetic benchmarks and on the Star Schema Benchmark (SSB). On SSB, our schemes can achieve similar compression rates to the best state-of-the-art compression schemes in GPU (i.e., nvCOMP) while being 2.2× and 2.6× faster in decompression speed and query running time.

The rest of the paper is organized as follows: related work and background are discussed in Section 2. We introduce the *tile-based decompression model* in Section 3. We present the data format and the unpacking implementation on the GPU for `GPU-FOR`, `GPU-DFOR`, and `GPU-RFOR` in Section 4, 5, and 6 respectively. Section 7 discusses the integration into Crystal. Section 8 discusses the usage and choice of compression scheme, parameter, and other relevant discussion. Section 9 evaluates the performance and compression ratio of our approach against other schemes on GPU. Finally, we conclude in Section 10.

2 BACKGROUND AND RELATED WORK

In this section, we review the basics of GPU architecture and describe past approaches to data compression on GPUs and CPUs.

2.1 GPU Architecture

Performance of database operations on GPU is bound by the memory subsystem (either shared or global memory) [49]. The lowest and largest memory in the hierarchy is the *global memory*. A modern GPU can have global memory capacity of up to 80 GB with memory bandwidth of up to 2000 GBps. Each GPU has a number of compute units called *Streaming Multiprocessors (SMs)*. Each SM has a number of cores and a fixed set of registers. Each SM also has a *shared memory* (SMEM) which serves as a scratchpad that is controlled by the programmer and can be accessed by all the cores in the SM. Accesses to global memory from a SM are cached in the L2 cache (L2 cache is shared across all SMs) and optionally also in the L1 cache (L1 cache is local to each SM).

Processing on the GPU is done by a large number of threads organized into *thread blocks* (each run by one SM). Thread block size can vary from 32 to 1024 threads. Thread blocks are further divided into groups of threads called warps (usually consisting of 32 threads). The threads of a warp execute in a *Single Instruction Multiple Threads (SIMT)* model, where each thread executes the same instruction stream on different data. The device groups global memory loads and stores from threads in a single warp such that multiple loads/stores to the same cache line are combined into a single request. Maximum bandwidth can be achieved when a warp's accesses to global memory target neighboring locations.

The programming model allows users to explicitly allocate global memory and shared memory. Shared memory has an order of magnitude higher bandwidth than global memory (10 TBps vs. 900 GBps on the Nvidia V100 GPU) but has much smaller capacity (a few MB vs. multiple GB). Finally, registers are the fastest layer of the memory hierarchy. If a thread block needs more registers than available, register values spill over to global memory.

[1]The source code is available at https://github.com/anilshanbhag/gpu-compression

2.2 Compression Techniques

Lossless compression techniques have been heavily exploited in modern column-store databases for efficient query processing and can be categorized into two buckets: *lightweight* and *heavyweight*. Lightweight algorithms are mainly used in in-memory column stores while heavyweight algorithms like Huffman [27] and Lempel Ziv [51] (together with lightweight techniques) are used in disk-based column stores. In this paper we focus on lightweight techniques. We show later in Section 9 that most of the compression gains can be achieved with just lightweight techniques.

There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [21, 52], delta coding (DELTA) [29], dictionary compression (DICT) [10, 52], run-length encoding (RLE) [10], and null suppression (NS) [10].

FOR represents each value in a sequence as a difference to a given reference value. FOR is applied to a block of integers and the reference value chosen is usually the minimum value to make all values positive. FOR is good when the block of integers have similar values.

DELTA represents each value as a difference to its predecessor value. DELTA is good when the array is sorted or semi-sorted.

DICT replaces each value by its unique key in the dictionary. DICT is effective for columns with low cardinality.

RLE replaces uninterrupted sequences of occurrences of the same values (called runs) by the value and length of the sequence. Hence a sequence of values is replaced by a sequence of pairs (value, length).

NS removes leading zeros from an integer's bit representation. NS is useful when a column contains many small integers.

FOR, DELTA, DICT, and RLE work at the logical level where a sequence of values is compressed into another sequence. NS addresses the physical level of bits with the basic idea of removing leading zeros in the bit representation of small integers. There are many different NS techniques proposed which can broadly be categorized as (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned. Bit-aligned NS algorithms compress an integer to a minimal number of bits; byte-aligned NS compress an integer with a minimal number of bytes; word-aligned NS encodes as many integers as possible into 32/64-bit words. The NS algorithms also differ in their data layout. We distinguish between *horizontal layout* [31, 46] and *vertical layout* [19, 29, 31]. In the horizontal layout, the compressed representation of subsequent values is situated in subsequent memory locations. In the vertical layout, each subsequent value is stored in a separate memory word in a striping fashion. Cascading multiple compression schemes together has been done on both CPU and GPU to achieve a better compression ratio [5, 18, 29, 30], where the input of one scheme is the output of another scheme.

CPU-based compression. Wang et al. [44] presented a survey of bitmap and inverted list compression in the area of database and information retrieval. Among these schemes, VB [15, 17] is a variant of NS algorithm with variable byte-aligned packing. PFOR and its variants [48, 53] encode a block of integers such that the majority of the integers can be encoded into b-bits and store the rest of the integers at the end. Simple-N and its variants [11, 12, 50] are word-aligned compression methods with 4 status bits to represent N combinations of bitwidth and 28 data bits used to store the data. All these schemes are typically a NS algorithm cascaded with Delta/RLE since they run on sorted datasets. Li and Patel [31] proposed Bitweaving which includes Horizontal Bit-Parallel (HBP) and Vertical Bit-Parallel (VBP) storage layouts. HBP achieves better decompression performance but VBP achieves better compression rates. ByteSlice [19] improves the performance of VBP by striping in bytes instead of bits but at the cost of larger storage footprint.

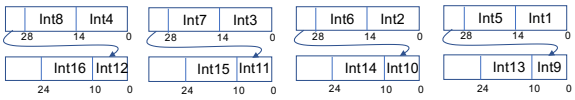


Figure 1: Bit-packing with vertical data layout

Across various compression schemes, the best performing ones are `SIMD-scan` [37, 46] which uses bit-aligned packing with a horizontal data layout and `SIMD-BP128` [29] (and its variants) which use bit-aligned packing with a vertical data layout. `SIMD-scan` stores column values in a tightly bit-packed horizontal layout, ignoring any byte boundaries. `SIMD-BP128` processes data in blocks of 128 integers at a time and stores these integers in a vertical layout using the number of bits required for the largest of them. Figure 1 illustrates the vertical layout where the first four integers Int1, Int2, Int3, Int4 start out in four different 32-bit words. Int5 is immediately adjacent to Int1, Int6 is adjacent to Int2, etc. Later in Section 4.3, we discuss the difference between our schemes and `SIMD-scan`/`SIMD-BP128`.

GPU-based compression. Fang et al. [18] and HippogriffDB [30] support the cascading of five basic lightweight techniques. They evaluated two byte-aligned schemes: NSF and NSV. NSF encodes every element in the input array with a fixed number of bytes whereas NSV allows each value to be encoded in a variable length of bytes. To generate the compression schemes, these designs use a compression planner to generate a plan for each column. Based on the column properties (i.e., sorted/unsorted, average run length, number of distinct values, etc.), the planner will generate a plan with the best compression ratio. During decompression, these designs treated each compression scheme as independent layers. Therefore, during decompression of cascaded scheme, different kernels will be called in succession to decode one compression layer at a time (see Figure 4 (left)). This leads to suboptimal decompression performance.

Mallia et al. [33] introduced two new NS algorithms (`GPU-BP` and `GPU-VByte`). `GPU-BP` encodes the data in a horizontal layout similar to `SIMD-Scan`. `GPU-VByte` decodes the input array with a variable length of bytes similar to NSV in [18]. This work, however, does not support cascading compression schemes which limits its compression rate. nvCOMP [5] is a generalized CUDA-based compression library that supports cascaded compression schemes consisting of the five basic lightweight techniques. Their bit-packing scheme does not saturate memory bandwidth. Further, nvCOMP does not support the end-to-end pipelining of multiple decompression steps with query execution. This leads to suboptimal performance during query execution. We will evaluate the performance of all the previous GPU compression schemes in Section 9.4.

2.3 Query Execution on GPUs

A flurry of recent work [20, 26, 32, 40, 49] and a number of commercial systems (e.g, Omnisci [7], Kinetica [4], and BlazingSQL [1]) use GPU(s) to accelerate query performance. These

works can be broadly categorized into two categories. The first category [1, 4, 7, 40] uses one or more GPUs to store all or significant fraction of the working set directly in GPU memory and aim to deliver interactive query response time; the main constraint is the limited GPU memory capacity. The second category [20, 26, 30, 32, 49] uses the GPU as an accelerator, where data is stored primarily in CPU and moved to the GPU at query execution time. Some of these works focused on accelerating individual database operations such as selection [42], join [24, 25, 28, 36, 38, 39, 41, 47], and sort [22, 43]. Recently, Lutz et al. [32] showed that using the hybrid system with CPU and GPU connected using the high-bandwidth NVLink can deliver significant performance improvement over a CPU-only system. For this category, the bottleneck is the amount of data moving over the interconnect. Our work makes it possible to use bit-packing to compress columns leading to smaller storage footprint in category 1 and reduces data transfer time for category 2.

A recent work, Crystal [40], presented a library of CUDA device functions that can be composed together to execute analytic queries on GPUs. Crystal adopts the idea of *tile-based execution model*, which instead of viewing each thread as an independent execution unit, views a thread block as the basic execution unit with each thread block processing a tile of entries at a time. Compared to a single thread, a single thread block can hold a significantly larger group of elements collectively in shared memory. This group of elements is called a *tile*. The key advantage of this model is that after a tile is loaded into shared memory, subsequent passes over the tile will be read directly from shared memory, avoiding multiple round-trips to the global memory. As a result, Crystal could execute analytic queries close to memory bandwidth speed. In this paper, we introduce *tile-based decompression* which adopts the idea for decompression routines (see Section 3).

3 TILE-BASED DECOMPRESSION

Cascading compression schemes achieve better compression ratio than individual compression schemes [18, 30]. However, decompressing a cascaded scheme can degrade performance as intensive decompression overburdens the GPU. The reason behind the bad performance was that past work decoded one layer of compression at a time and thus required multiple passes to decode the compressed data and execute the query. Figure 2 (left) illustrates this approach for decompression of table data consisting of delta encoding, frame of reference, and fixed length byte-aligned packing (`Delta+FOR+NSF`). To fully decode the column and execute the query, these systems have to (1) do a first pass to unpack the byte-packed data (decompress `NSF`), (2) do a second pass to add the data with the base reference (decompress `FOR`), (3) do a third pass to delta decode the data (decompress `Delta`), and finally (4) launch the query kernel on top of the the fully decoded data. Step (2) and (3) can potentially be merged together into a single step — adding the reference while unpacking the byte-packed data. Prior works [18, 30], however, separated the kernels to support more flexible compression plans. If the query requires multiple encoded columns, steps (1)–(3) will have to be repeated for each column. In this model, decompression is expensive since the intermediate data is read and written to the global memory after every kernel pass,

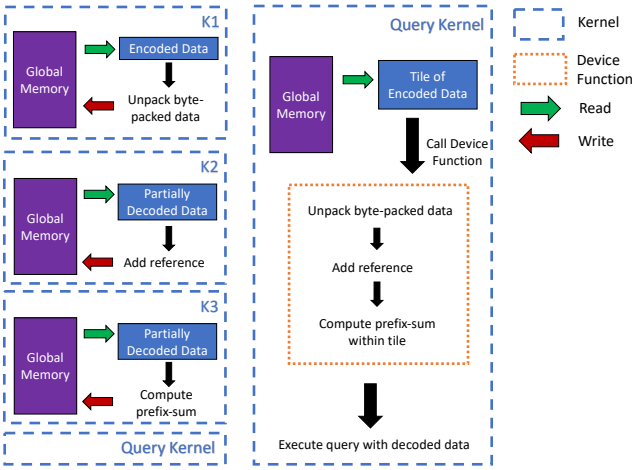


Figure 2: Decoding cascaded compression scheme (Delta+FOR+NSF) using *cascaded decompression* (left) and using *tile-based decompression* (right)

incurring significant memory traffic. We will refer to this model as the *cascading decompression model*.

In this work, we introduce the *tile-based decompression model* which is inspired by *tile-based execution model* [40]. In the tile-based decompression model, each *thread block* collectively loads a block of encoded data into shared memory, which is called a tile. Next, multiple steps of decompression are applied on the tile directly in shared memory, avoiding multiple passes to the global memory. Figure 2 (right) shows how to decompress the cascaded scheme `Delta+FOR+NSF` with *tile-based decompression*. The three decompression steps can now be encapsulated into a single function which can be called from inside the query kernel during execution. This would enable us to decompress cascaded compression schemes in a single pass over the column. Further, this decompression can be inlined with query execution. Compared to *cascading decompression*, the intermediate data transfer of *tile-based decompression* is X times less, where X is the depth of the compression layers.

Applying *tile-based decompression* requires each compression scheme in the cascade to have the following two properties:

Property 1: Tile-granularity data format
In tile-based model, the data needs to be partitioned and encoded in the granularity of tiles where each tile fits in shared memory. This allows us to decode each tile independently.

Property 2: Tile-based decompression routine
During decompression, we want to read the encoded data from global memory only once. To achieve this for cascaded schemes, we should be able to express the decompression routine as a function that takes a tile in shared memory as input and outputs a tile to shared memory.

In the next three sections, we will show that `FOR`, `Delta`, `RLE` and bit-aligned `NS` obey the above two properties. We will introduce three new bit-packing based compression schemes (`GPU-FOR`, `GPU-DFOR`, and `GPU-RFOR`) and discuss how they can be integrated with the *tile-based decompression model*. We will also discuss the integration with Crystal system in Section 7.

4 FAST BIT UNPACKING

While Section 3 has demonstrated that *tile-based decompression* could solve the cascading decompression problem in GPU. However, individual compression schemes such as bit-packing have not been sufficiently optimized on GPU. Addressing this issue will be the main focus of the next three sections.

In this section, we describe the GPU-FOR compression format, which uses bit-packing in conjunction with Frame-of-Reference (FOR) to store data compactly on the GPU and the fast bit unpacking routine used to decompress it efficiently on the GPU. GPU-FOR can be used to efficiently compress attributes of type integer, decimal, or dictionary-encoded string (i.e. sequence of integers) in a column store. At runtime, the executor decompresses data and runs the query on the decompressed data. Hence, optimizing the performance of decompression is critical for analytic workloads. In the rest of the section, we describe first the bit-packing representation we use and then the kernel implementation on the GPU. We present a series of optimizations that allow us to decode bit-packed data while saturating memory bandwidth.

4.1 Data Format

Bit-packing is a process of encoding small integers in $[0, 2^b)$ using b bits; b can be arbitrary and not just 8, 16, 32, or 64. Each number is written using a string of length b. Bit strings of fixed size b are concatenated together into a single bit string, which can span several 32-bit words. If some integer is too small to use b bits, it is padded with zeros. Compressing 32-bit integers to b bits achieves a compression ratio of $32/b$, which can be significant.

Choosing a common bit size b for an entire array would mean that the occurrence of a single large value would increase the number of bits needed to encode the values. Hence, bit-packing is generally used in conjunction with FOR encoding. In GPU-FOR, the array of values is partitioned into *blocks* of 128 integers, which is the tile that will be processed by a thread block. The range of values in the block is first found and then all the values are written in reference to the minimum value. For example, if the values in a block are integers in the range [100,130], then using a reference of 100, we can store them using 5 bits ($log_2(130+1-100)$). Each block is further divided into sequences of 32 integers called *miniblocks*. For each miniblock, we choose a bit-width based on the maximum number of bits needed to encode the largest value.

The choice of block size and miniblock size is to ensure they align on a 32-bit boundaries (shared memory access granularity). Having 32 integers in one miniblock means that for any bitwidth, the miniblock always ends on 32-bit boundary. This would allow us to use 32-bit arithmetic while decoding and make shared memory accesses efficient. The bitwidth for a miniblock can then be stored in 1 byte. Since we want to align along 32 bit boundaries, we group 4 miniblocks into a single block and store the 4 bitwidths at the start of the block using a single integer. Hence, each block now consists of 128 integers.

The bit-packed array needs to be decoded in parallel across a large number of threads. For this, we store the start index of the blocks in a separate array called block starts. Finally we store the metadata associated with the encoding: block size (i.e., the number of integers within each block), miniblock count (i.e., number of miniblocks per block), and the total count (i.e., total number of integers in the data array) in the header. Figure 3 shows a schematic of the format we use to store data.

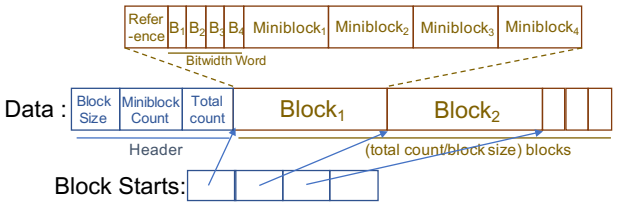


Figure 3: GPU-FOR Data Format

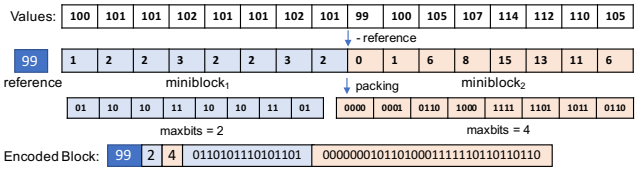


Figure 4: Example encoding with GPU-FOR

Figure 4 shows an example of encoding 16 integers into a block with 2 miniblocks. The minimum value in the block (i.e., 99) is used as the reference. We calculate the difference of the block values from the reference. Each miniblock contains 8 integers. We see that the first miniblock needs 2-bits per block while the second miniblock needs 4-bits per block. We encode each miniblock with their respective bitsizes and store the reference and bitwidths at the start of the block.

Our encoding format with horizontal data layout is similar to previous approach on CPU [37, 46]. We will discuss the key difference between our design and existing CPU designs later in Section 4.3.

4.2 Implementation

In this section, we describe a number of optimizations at the implementation level that we applied to achieve decompression at close to GPU memory bandwidth speed. To give an impression of the importance of each optimization, we end every subsection with the time taken to decode a compressed dataset of 500 million integer values drawn from a uniform distribution $U(0, 2^{16})$. The details of the experimental setup can be found in Section 9.1.

Base Algorithm: Algorithm 1 shows the pseudocode that would run in parallel on each thread (n threads are allocated for n-element dataset). Each thread block (of size 128 threads) is assigned to decode a block (of 128 elements) with each thread decoding one element in the block based on its index. Each thread starts by reading the block start pointer of the block to find where in the data array the block starts (line 1–2). Each thread then reads in the bitwidth_word, and uses it to compute the offset of its miniblock in the data array (miniblock_offset) (lines 7–10). In computing the miniblock offset, we use the fact that if entries in a miniblock are encoded with b bits, then the miniblock occupies $4b$ bytes (since there are 32 entries per miniblock). Next, we compute the offset in bits within the miniblock (line 12). Since the entries are bit-packed, they are not byte-aligned and can span byte boundaries. Using starting bit index, we calculate the starting integer index (start_intindex) of the entry (lines 13-14). We then load an 8-byte block starting at

Algorithm 1: Fast Bit Unpacking on GPU — The following code runs on each of the 128 threads within a thread block in parallel.

```
Input : int[] block_starts; int[] data; int block_id;
        int thread_id
Output: int item
1  int block_start = block_starts[block_id];
2  uint * data_block = &data[block_start];
3  int reference = data_block[0];
4  uint miniblock_id = thread_id/32;
5  uint index_into_miniblock = thread_id & (32 - 1);
6  uint bitwidth_word = data_block[1];
7  uint miniblock_offset = 0;
8  for j = 0; j < miniblock_id; j++ do
9      miniblock_offset += (bitwidth_word & 255);
10     bitwidth_word >>= 8;
11 uint bitwidth = bitwidth_word & 255;
12 uint start_bitindex = (bitwidth * index_into_miniblock);
13 uint header_offset = 2;
14 uint start_intindex = header_offset + miniblock_offset +
     start_bitindex/32;
15 uint64 element_block = data_block[start_intindex] |
     (((uint64)data_block[start_intindex + 1]) << 32);
16 start_bitindex = start_bitindex & (32-1);
17 uint element = (element_block & (((1<<bitwidth) - 1) <<
     start_bitindex)) >> start_bitindex;
18 item = reference + element;
```

`start_intindex` (`element_block`) (line 15). This block contains the entire element, we use bitshift arithmetic to extract the entry (lines 16–17). Finally, we add `reference` and return the result. The result resides in a register and is used subsequently during query execution. In Section 7, we describe in greater detail how the algorithm is used during query execution.

This algorithm takes 18 ms to decompress the dataset described at the start of the section. Reading an uncompressed dataset of 500 million 4-byte integers takes 2.4 ms. This means decompressing the dataset is 7.5× slower than reading the uncompressed data. Below we describe a number of optimizations to bridge the gap:

Optimization 1: Operating in Shared Memory

Each thread makes multiple requests to the data array which sits in global memory. Since, all the requests made by all threads within a thread block touch one data block, in this optimization, we load the entire block into shared memory once at the start of the operation. Each thread block starts by reading `block_start[BlockId]` and `block_start[BlockId+1]` to determine the boundaries of the data block to be processed and then loads it into its shared memory in a coalesced manner. All subsequent requests are made to the data block in shared memory. Recall that the shared memory is an order of magnitude faster than global memory. This optimization shifts global memory reads to shared memory reads, thereby improving performance. This optimization results in runtime reduction from 18ms to 7ms on the sample dataset.

Optimization 2: Processing Multiple Blocks

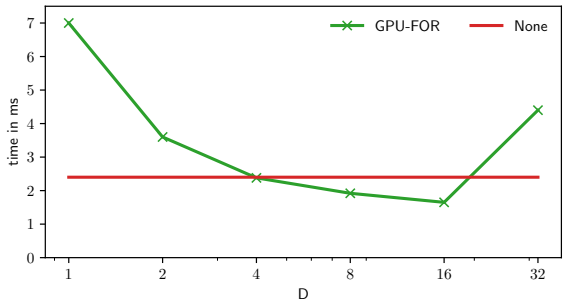


Figure 5: Decompression performance with varying number of data blocks per thread block (D)

The granularity of reads from global memory is 128 bytes [40]. This is achieved when warps (group of 32 threads) access a 4-byte integer array of size 32. In the sample dataset, if all integers end up being encoded with 16 bits, the block size is 258 bytes (2 bytes for block header + 256 bytes for miniblocks). When a thread block of size 128 reads in the data block from global memory, some warp accesses do not result in an aligned full segment being read from global memory. The same issue occurs when we access the `block_start` array, we are reading in only two values from global memory, again leading to loss of efficiency.

In this optimization, we attempt to reduce the impact of these irregular accesses to global memory by processing multiple data blocks per thread block. Each thread block is assigned $D(= 2/4/8/16/32)$ data blocks to process. At the start, each thread block reads in $D + 1$ `block_start` entries from global memory. Next they read in the data blocks `block_start[D×BlockId]` and `block_start[D×BlockId + D]` from global memory into shared memory. As a result, we have reduced the number of irregular accesses to both the `block_start` and the `data` array.

Figure 5 shows the runtime for decompression of the sample dataset with varying D. As we can see from the figure, the largest reduction comes from going from $D = 1$ to $D = 4$. Going from $D = 4$ to $D = 16$ improves the performance, however the improvement is marginal. Finally, when we go to $D = 32$ the performance deteriorates significantly. This is because the number of registers required and the shared memory requirement increases proportional to D. On an Nvidia V100 GPU, each thread can only use 65 registers and 48 bytes of shared memory per thread at full occupancy. As a result, when we go to $D = 16$, each thread requires 64 bytes of shared memory which reduces occupancy slightly. When we go to $D = 32$, each thread requires 128 bytes of shared memory which results in significant reduction in occupancy and register spilling — hence the slowdown.

When we run full SQL queries, we have to store D values per output column in registers until the end of the query. We noticed in our evaluation on the Star Schema Benchmark (discussed later in Section 9.4) that there is little difference in performance with $D = 4/8$ and choosing $D > 8$ leads to deterioration in performance. This is because each query has 3-4 output columns and choosing higher values of D leads to register spilling and reduced occupancy. Hence, we choose to simply use $D = 4$ in the rest of the paper. Note that D is a parameter and users can choose higher value of D in case they are just decoding a single column.

Optimization 3: Precomputing Miniblock Offsets

Computing the `miniblock_offset` involves a `for` loop (lines 8–11 in Algorithm 1). We can make two observations: (1) miniblock offsets are a exclusive prefix sum over the bitwidths array (2) with $D = 4$, there are only $D * 4 = 16$ unique miniblocks offsets to compute, while Algorithm 1 performs this computation on all 128 threads redundantly. In this optimization, we reduce the compute load of the algorithm by precomputing the $D * 4$ miniblock offsets once at the start and storing them in shared memory. On the first $D*4 = 16$ threads (i.e. `thread_id` $\in [0, 16)$), we task by each thread one miniblock offset/bitwidth pair to compute. Each such thread loads the corresponding bitwidth word and computes a prefix sum over it (this can be done using bitshift arithmetic). Finally we extract the relevant offset and bitwidth for the miniblock and store it in shared memory. These values are read by each thread when they need it. The optimization eliminates the for loop in lines 8–11 in Algorithm 1 and reduces the runtime from 2.39ms to 2.1ms, which is lower than the time taken to read the uncompressed data.

4.3 Discussion

GPU-FOR vs CPU Designs: As described earlier in Section 2.2, there are two variants of bit-packing based on the data layout: *horizontal* and *vertical*.

For horizontal data layout, the best performing scheme is `SIMD-scan`. While the original work focused on decoding an entire column using a single bitwidth, they could be modified to work with GPU-FOR-like data format that combines `FOR` with bit-packing. While the data format is similar, the implementations are distinct. `SIMD-scan` uses register shuffling using SIMD instructions to decode the data which does not directly work on the GPU. `GPU-FOR` decodes directly from GPU's fast shared memory and uses optimizations to minimize shared memory and global memory accesses.

For vertical data layout, the best performing schemes are `SIMD-BP128` (and its variants). `SIMD-BP128` uses SSE instructions with each SSE register holding 4 32-bit integers. The data is encoded with 4 lanes each with 32 integers allowing the data to be decoded efficiently by mapping each lane to a different vector lane of the SSE register. Each block encodes 128 integers. To ensure 16-byte alignment, `SIMD-BP128` groups 16 blocks together, storing the bitwidths used in each block at the start. This is similar to `GPU-FOR` format with each block having 16 miniblocks, with each miniblock having 128 integers and encoded with a vertical layout.

On the GPU, if we consider a SIMD lane as equivalent to a GPU thread in a warp, we can directly translate the `SIMD-BP128` style vertical storage layout to the GPU. Let's call this `GPU-SIMDBP128`. We go from having 4 lanes on the CPU to 32 lanes on the GPU (warp size is 32 threads). As a result, on a typical thread block size of 128, with each thread having 32 integers to ensure their lane terminates in 32-bit boundaries, we would need a block size of 4096 vs 128 on the CPU. We implemented `GPU-SIMDBP128` and compared the performance of `GPU-FOR` vs `GPU-SIMDBP128` on the same microbenchmark. `GPU-FOR` (with $D = 16$) takes 1.55ms compared to `GPU-SIMDBP128` which takes 4.3ms. Hence `GPU-SIMDBP128` is 2.7× slower than `GPU-FOR`.

On the GPU, vertical packing like `SIMD-BP128` is slower because the number of registers available per thread is limited. Decoding the vertical layout would require space for 32 4-byte entries and 32 registers to store output. Similar to the case when $D = 32$, this leads to reduced occupancy. If we have a query with only 3 columns needed for result computation, we would need more than 2× the registers available per thread resulting in significant register spilling. To get a sense for the performance impact, we evaluated the Star Schema Benchmark q1.1 (described later in Section 9.4) with columns encoded using GPU-FOR vs with columns encoded using `GPU-SIMDBP128`. The query uses 4 columns. The performance with `GPU-SIMDBP128` was 14× slower than with `GPU-FOR`. Another downside of using `GPU-SIMDBP128` is the large block size (4096 vs 128). Large block sizes mean that one skewed value could lead to large bitwidth for the entire block, reducing compression gains.

Bit-packing without Miniblocks: Instead of having 4 miniblocks, one could instead just have one miniblock encoded with a single bitwidth. There is no difference in terms of memory space overhead as both schemes store a bitwidth(s) as a single 4-byte integer. However, there is reduced computation as we don't have to calculate the miniblock offsets. We implemented this scheme and found the performance to be marginally better. The performance on the sample dataset improves from 2.1ms to 2ms. When we experimented further to see if it is possible to reduce runtime by reducing compute load by having a single bitwidth across blocks or using zero as reference, we could not achieve any further improvement. This leads us to believe that the performance is close to saturating bandwidth given our global memory access pattern.

5 FAST DELTA DECODING

Delta encoding (also called differential encoding) is a common approach used (typically in conjunction with other techniques) to compress sorted or partially-sorted integer/decimal arrays. Instead of storing the original array of integers (x_1, x_2, x_3...), delta encoding keeps only the difference between successive integers together with the initial integer ($x_1, \delta_2 = x_1 - x_1, \delta_3 = x_3 - x_2, ..$). Since the differences are typically much smaller than the original integers, delta encoding allows for more efficient compression. In this section, we describe the `GPU-DFOR` coding scheme that uses delta encoding in conjunction with bit-packing and frame of reference to achieve good compression ratios and can be decoded efficiently.

The sequential form of delta encoding requires just one subtraction per value ($\delta_i = x_i - x_{i-1}$). During decoding, we require one addition per value ($x_i = \delta_i + x_{i-1}$). For an array A of k elements, the prefix sum p_A is a k-element array where $p_A[j] = \sum_{i=0}^{j-1} A_j = p_A[j-1] + A_j$. Hence, the process of delta decoding is equivalent to computing the prefix sum. Efficient parallel prefix sum routines have been proposed [23] that could be used to decode delta encoded data on the GPU. A simple approach to delta encode + bit-pack the data would be to do it as two separate steps: first compute the deltas for the entire array and then bit-pack the deltas. The decoding would then be a two-step process: the first pass would use the bit unpacking routine described in Section 4.2 to decode the deltas and write it to global memory; the second pass would use the prefix sum routine to decode the data. This is the approach used by past work [18, 30] and is inefficient as it requires multiple passes over the data. Later in this section we describe how we can combine the delta decoding step and the bit unpacking step into a single pass.

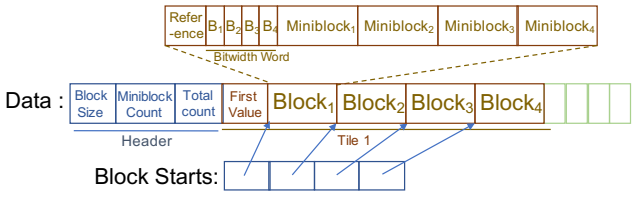


Figure 6: GPU-DFOR Data Format

5.1 Data Format

Delta encoding an entire array as $x_0, \delta_1, \delta_2 ...$ makes it hard to parallelize as decoding the n^{th} block requires the $(n-1)^{th}$ block to have been decoded already. To enable parallel decoding, we build our data format on GPU-FOR encoding scheme (Section 4.1) by partitioning the array into sets of D blocks where each block contains 128 integers and delta encoding each set of D blocks independently (where D is the number of blocks processed per thread block), as shown in Figure 6. Encoding x integers generates $x-1$ deltas. During encoding, we pad the deltas with 0 to ensure every block has 128 entries. We store the first value separately before every D^{th} block, with start pointers still pointing to the start of each block.

GPU-DFOR compresses better than GPU-FOR on sorted and semi-sorted arrays, e.g., sorted primary keys or posting lists in search workloads. Consider a dataset of $n = 500$ million integers with entries from 1 to n sorted. This dataset when compressed using GPU-DFOR uses 1.8 bits per integer vs 7.8 bits per integer used by GPU-FOR. Using it for unsorted data could lead to worse compression ratios compared to simply bit-packing the data. For example, for an array of integers drawn uniform randomly from $[0, 32)$, the deltas will be in the domain $[-31, 31]$. In this case, GPU-DFOR will likely require more bits per integer than GPU-FOR.

5.2 Implementation

With the data format described above, each tile of D blocks can be decoded independently. During decoding, we first start by loading the D block segments into shared memory and use the fast bit unpacking routine (described in Section 4.2) to decode the deltas.

After bit unpacking the deltas, we have D deltas per thread. The output data entries can be calculated using prefix sum over the deltas of all threads within the thread block. We can use block-wide prefix sum to compute the prefix sum over the deltas based on the work-efficient prefix sum algorithm proposed by Blelloch et al. [13]. For an array of n integers, the algorithm is able to compute the prefix sum of the array in parallel using $\Theta(\log n)$ steps. Interested reader can refer to [13, 40] for more details.

Although prefix sum has been used widely in libraries like Thrust [9], doing prefix sum over an entire array is expensive and involves multiple passes over data. A key observation we make is that since we do delta coding in each tile (set of D blocks) separately, prefix sum can happen entirely within a thread block in shared memory. This also allows us to fuse bit unpacking and delta decoding steps into a single kernel which allows our implementation to perform decompression in a single pass over the data blocks in global memory compared to multiple passes required by previous works [18, 30]. The bit unpacking and delta decoding share the same shared memory buffer. In our implementation, we reuse an existing block-wide prefix sum implementation from Crystal. The total resource requirement of the kernel is D 4-byte entries in shared memory and D registers to store the output per thread.

6 FAST RUN LENGTH DECODING

Run length encoding (RLE) is a common approach used to compress a data sequence with consecutive runs of values. For example, given an original array of integers $(x, x, x, x, y, y, z, z, z)$, run length encoding stores the original array as two separate arrays: the values (x, y, z) and the run length of the values $(4, 2, 3)$. For data set with high average run length, RLE can greatly reduce the total memory footprint. In this section, we describe the GPU-RFOR coding scheme that uses RLE in conjunction with bit-packing and frame of reference to achieve good compression ratios and efficient decompression time.

To enable parallel decoding, we once again build our data format on the GPU-FOR encoding scheme (Section 4.1) by partitioning the array into blocks of 512 integers and applying RLE to each block independently to create the value array and the run length array. We then apply FOR on top of both arrays separately. In addition to reference and bitwidths (see Section 4.1), we store extra metadata of the run length/values count at the beginning of each block. The two compressed representations of run length and values arrays are stored separately.

With the data format described above, each block can be decoded independently. During decoding, we first start by loading one block of compressed run length and one block of compressed values into shared memory and use the fast bit unpacking routine (described in Section 4.2) to obtain the uncompressed run length and values array for the corresponding block. After bit unpacking both columns, the output data block entries can be calculated using the four steps described in [18] for all the threads within the thread block. These steps include 2 scatter and 2 inclusive prefix sum operations. We use the same block-wide prefix sum as the one described in Section 5.

Since each block can be decoded independently, the four decoding steps of RLE can be served completely by the shared memory. This allows us to fuse bit unpacking and run length decoding steps into a single kernel which allows our implementation to perform decompression in a single pass over the data blocks in global memory. GPU-RFOR, however, requires twice more resources than GPU-DFOR since there are two input columns (value and run length) resulting in twice more blocks to be loaded into the shared memory.

7 DATABASE INTEGRATION

Given the efficient massively parallel bit-unpacking implementations described in the previous sections, we were naturally interested in its usability in a full system. As a proof of concept, we implemented the decompression routines as CUDA device functions[2] and show how they can be used with an existing GPU analytical engine. In particular, we chose Crystal [40], an open-source GPU analytics framework developed recently.

Crystal is based on the idea of a *tile-based execution model*. Previous work [40] has shown that SQL query operators and analytical queries implemented with Crystal can saturate memory bandwidth and thereby deliver an-order-of-magnitude speedup compared to

[2]Device functions are functions that can be called from kernels on the GPU

CPU-based implementations. We have implemented the decompression routines for GPU-FOR, GPU-DFOR, and GPU-RFOR as CUDA device functions LoadBitPack, LoadDBitPack, and LoadRBitPack respectively. These functions can be used in queries implemented in Crystal easily and can be used more broadly in any CUDA kernel as well. To integrate them into Crystal, the only required changes are to replace the load routines (BlockLoad) in Crystal with LoadBitPack. Therefore, the user can run the query on compressed data by just changing a single line of code. Readers interested in Crystal can refer to [40].

One key drawback of bit-packed data is that it lacks random access. Accessing any element requires loading the entire data block. As a result, when selections or joins filter data entries, we still have to read the entire column. We will show in Section 8 why this would not lead to material impact on performance.

Since the routines LoadBitPack, LoadDBitPack, and LoadRBitPack are ordinary device functions, they can be used directly in user's CUDA code in conjunction with other GPU frameworks like Thrust and Cub [9], and they can also be called directly from NVVM (a compiler internal representation based on LLVM IR designed to represent GPU compute kernels) [6].

8 DISCUSSION

In this section, we discuss certain key aspects that we haven't covered with respect to usage and choice of compression method.

Choice of Compression Scheme: The rule-of-thumb when choosing a compression scheme is to use the one that has the lowest storage footprint for each column. Therefore, data distribution plays a very important role in such cases. In general, GPU-DFOR is suitable for sorted or semi-sorted columns with high number of distinct values. GPU-RFOR is suitable for columns which have a low number of distinct values or columns with a high average run length. Other columns which do not have such properties are typically suited for GPU-FOR. Since each column can be decoded with different schemes, we will refer to this hybrid scheme as GPU-* for the remainder of the paper. We will show how GPU-* performs when we measure the end-to-end system performance in Section 9.

Hyperparameter Tuning: The number of blocks processed per thread block D is the only hyperparameter in the schemes we propose. As discussed in 4.2, we will choose $D = 4$ in our evaluation. As GPUs improve, it is likely they will have more shared memory and registers per thread, thereby allowing us to use higher values of D during query processing.

Compression Speed: In data analytics workload, data compression is generally a one-time activity that happens on the CPU side. However, in the event of updates, the data need to be recompressed and transferred again to the GPU memory to replace the old data. We measure the time required to compress the data to simulate such cases. Compressing 250 million entries of random dataset using GPU-FOR and GPU-DFOR on a 6 cores CPU machine takes approximately 1.2s and 1.3s respectively. GPU-RFOR is not suitable for this distribution and therefore takes longer to compress (2.2s).

Out-of-core Dataset: In the event that the compressed dataset does not fit in the GPU memory, the GPU as a co-processor model can be leveraged. In such case, the compressed data will be transferred from CPU to GPU before running each query. Compression still brings benefits since it reduces the total amount of data being transferred through the limited PCIe interconnect. We show the benefit of compression for out-of-core dataset in Section 9.5.

Random Access Performance: We measure and compare the performance of our compression schemes against uncompressed data under random access patterns. To simulate random access behavior, we generate a predicate bitvector to filter random 250 million data entries and sweep the selectivity from 0 to 1. For our compressed schemes (GPU-FOR, GPU-DFOR, and GPU-RFOR), we achieve good performance when $\sigma < 1/TILE\_SIZE$ since we can skip loading the entire compressed tile. When $\sigma > 1/TILE\_SIZE$, however, we will have to load the entire compressed tile from the global memory and decompress it before applying the bitvector. This result in the constant performance of 2.1 ms for GPU-FOR and GPU-DFOR.

For uncompressed data, since the granularity of access in GPU global memory is 128B, when the selectivity is above ($\sigma > 1/32$), we would have to read the entire cache line for every random access which results in reading the whole dataset. This results in the constant performance of 2.5 ms. Our performance is better since the reduction in data size reduces the total data read which often compensates for the loss of efficiency in the case of a selective filter. This shows that random access does not have a material impact on performance in a compressed setting.

9 EVALUATION

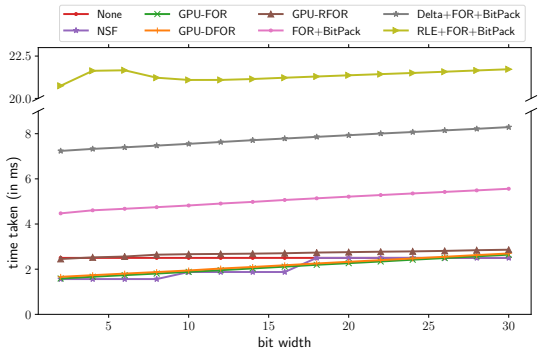
In this section, we evaluate the performance characteristics of the different compression schemes (GPU-FOR, GPU-DFOR, and GPU-RFOR) to understand (1) impact of applying *tile-based decompression*, (2) impact of data distribution, (3) performance within SQL queries, and (4) performance of our encoding schemes compared to the existing schemes from previous works.

The rest of the section is organized as follows: we discuss the experiment setup in Section 9.1. In Section 9.2 we evaluate the performance of different encoding schemes on synthetic dataset with varying bitwidths and illustrate the impact of tile-based decompression on our encoding schemes. In Section 9.3, we evaluate the impact of data distributions to different encoding schemes. In Section 9.4, we evaluate the end to end systems performance against previous works using the Star Schema Benchmark. Finally, in Section 9.5, we discuss the case when GPU is used as a coprocessor.

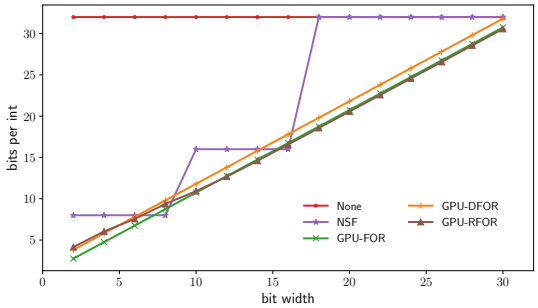
9.1 Setup

Hardware configuration: For the experiments, we use a virtual machine instance that has an Nvidia V100 GPU which is connected to Intel Xeon Platinum 8167M CPU via PCIe3. The Nvidia V100 GPU has 16 GB of HBM2 memory. The global memory read/write bandwidth is 880 GBps. The Intel Xeon Platinum CPU has 6 virtual cores and 180 GB DRAM. The bidirectional PCIe transfer bandwidth is 12.8 GBps. The system is running on Ubuntu 18.04 and the GPU instance uses CUDA 11.2.

Measurement: In our evaluation except for Section 9.5, we ensure that data is already loaded into the GPU memory before experiments start. We run each experiment 3 times and report the average measured execution time.



(a) Decompression time



(b) Compression rate

Figure 7: Performance with Varying Bitwidths

9.2 Performance with Varying Bitwidths

In this section, we test the performance of our proposed compression algorithms (GPU-FOR, GPU-DFOR, and GPU-RFOR) against existing encoding schemes on workloads with varying bitwidths. We will compare the performance against the following schemes:

- None: Data is stored as 4-byte integers uncompressed.
- NSF: Null suppression with fixed length encoding. The entire array is encoded as 1, 2, or 4 byte entries depending on the maximum number of bits needed for any integer in the column.
- FOR+BitPack: Using GPU-FOR encoding scheme but adopting *cascading decompression* routine. Hence, it will decode the schemes in two kernel passes — the first pass will decode bit-packing and the second pass will decode frame of reference.
- Delta+FOR+BitPack: Using GPU-DFOR encoding scheme but adopting *cascading decompression* routine. Hence, it will decode the schemes in three kernel passes — the first two passes to decode FOR+BitPack followed by a prefix sum to decode Delta.
- RLE+FOR+BitPack: Using GPU-RFOR encoding scheme but adopting *cascading decompression* routine. It decodes the schemes in eight kernel passes — the first four passes to decode FOR+BitPack for both the run length and values columns and the last four passes to decode RLE following the steps described in [18].

For this comparison, we use a synthetic dataset with varying bitwidth. We generate 15 unsorted datasets each with 250 million entries, such that all data elements in the i-th dataset have exactly i effective bits, i.e., the value range is $[0, 2^i)$ for i = 2,4,..,30. Within each range, the values are uniformly distributed.

Figure 7a shows the decompression time of these eight schemes for varying bitwidth. In this figure, we are (1) reading the compressed data from the global memory, (2) decoding it, and (3) writing back the uncompressed data to the global memory. Note that (3) is constant across bit widths while (1) will incur higher global memory operation (read) as the bitwidth increases which would translate directly to slightly increasing decompression time for all schemes. In all measurements, GPU-FOR, GPU-DFOR, and GPU-RFOR perform significantly better compared to their correspondence without *tile-based decompression*. GPU-FOR achieves 2.6× better performance compared to FOR+BitPack. GPU-DFOR achieves 4× better performance compared to Delta+FOR+BitPack. GPU-RFOR achieves 8× better performance compared to RLE+FOR+BitPack. This shows the benefit of applying *tile-based decompression* on our schemes.

The performance of NSF is a staircase pattern where the runtime is based on whether the entry size is 1, 2, or 4 byte. Both None and NSF saturate memory bandwidth. The performance of GPU-FOR is slightly worse than NSF with the worst case gap of 15% achieved at bitwidth 7. The gap is due to slightly larger data size and irregular access pattern associated with accessing the block_starts array used to find the block offsets in the data array. The decompression performance of GPU-DFOR is comparable but slightly worse to GPU-FOR as it has a slightly larger storage footprint. The decompression performance of GPU-RFOR is worse than GPU-FOR and GPU-DFOR. Apart from the fact that the random dataset is not suitable for this compression scheme, decoding GPU-RFOR also requires twice as much resource in terms of GPU registers and shared memory capacity and operations compared to GPU-DFOR.

Figure 7b shows the compression rate of five compression algorithms. The bit-packing schemes achieve the finest possible granularity and thus can perfectly adapt to any bitwidth. Consequently, the compression rate is a linear function of the bitwidth. The overhead for GPU-FOR is 0.75 bit per int (1 block start word + 1 reference word + 1 bitwidth word per block of 128 integer entries). The overhead for GPU-DFOR is 0.81 bit per int (0.75 + 1 first value word per $D = 4$ blocks). As the data is not sorted, the deltas vary in the range $[0, 2^i)$ and require one additional bit; our experiments in Section 9.3 show the benefit of GPU-DFOR on sorted data. Since the dataset generates random values, GPU-RFOR is not suitable for this dataset. Nevertheless, the compression rate is still a linear function of the bitwidth since the bit-packing scheme is applied on top of the RLE columns. The overhead for GPU-RFOR is slightly less than GPU-FOR since this scheme uses 512 integer entries per block, resulting in less compression metadata (the block start word and the reference word is stored every 512 entries).

Overall, this experiment shows that (1) *tile-based decompression* can significantly improve the performance compared to *cascaded decompression model*, and (2) our bit-packing schemes manage to achieve very good compression rates with almost no overhead on decompression speed.

9.3 Performance Across Data Distributions

In this section, we test the robustness of various compression schemes by evaluating their performance using various data distributions. In addition to the schemes we compare against in the

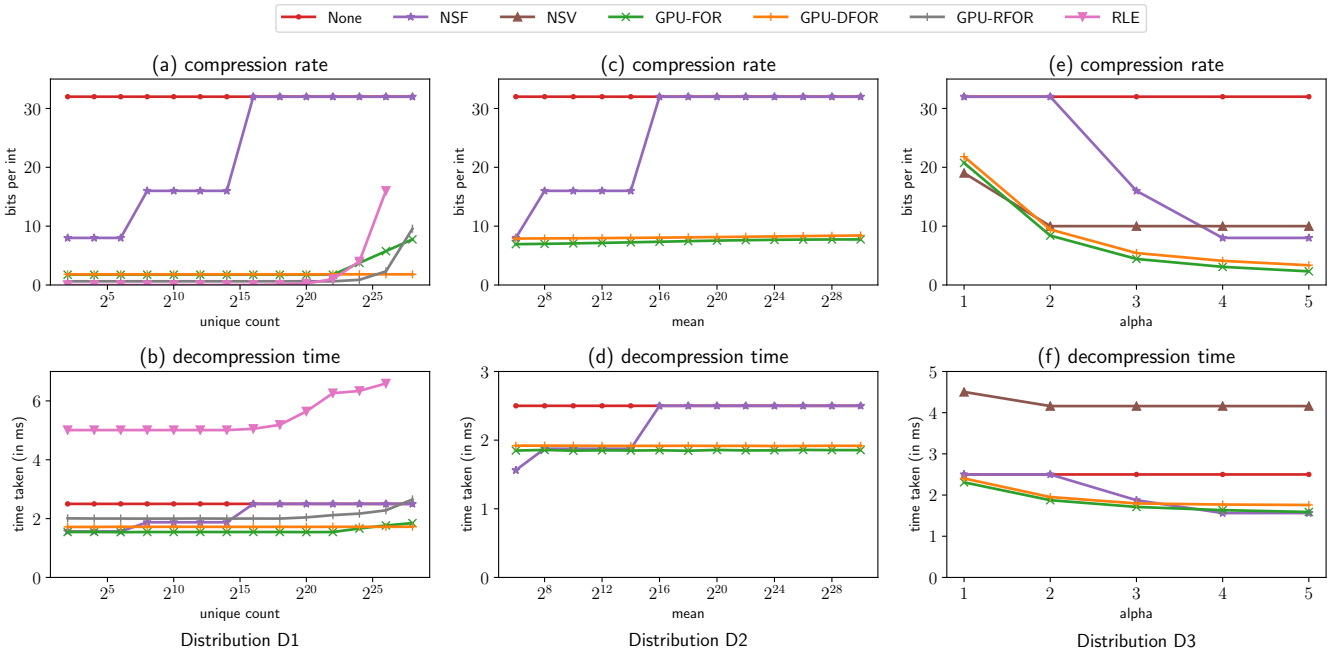


Figure 8: Comparison of compression schemes on different data distributions

previous section, we also compare to the performance of two more compression schemes:

- RLE: Represents runs of the same value as a pair: (value, run-length). Values and run lengths are stored in two separate columns.
- NSV: Represents each value with a variable number of bytes (1,2,3 or 4). In a separate array it maintains the number of bytes used using 2 bits per value. This scheme is good for handling skew.

In this experiment, we generate synthetic workloads with the size of 250 million entries with the following three different distributions.

- D1: a sorted array where we vary the number of unique values from 4 to 2^{28}. Typically a table is sorted based on one column, which D1 is designed to resemble.
- D2: a normal distribution with a standard deviation of 20 and mean varying from 64 to 2^{30}.
- D3: a Zipfian distribution with the exponent *alpha* characterizing the distribution varying from 1 to 5 (1 is least skewed, 5 is most skewed). D3 resembles dictionary encodings of tweets or text corpora where distribution of words follows Zipf's law. For this distribution, we also compare against NSV.

We will not evaluate RLE/GPU-RFOR on D2, D3 as they are not suitable for these data distributions.

The results for D1 can be found in Fig. 8 (a-b). The bit-aligned algorithms GPU-FOR, GPU-DFOR, and GPU-RFOR achieve better compression ratios compared to None and NSF due to use of FOR. As the number of unique values increases beyond 2^{22}, the block of 128 integers is likely to have different values. As the dataset is sorted, GPU-DFOR can encode such cases with fewer bits compared to GPU-FOR and GPU-RFOR. In the extreme case, when the number of unique values equals 2^{28} i.e., each value is unique and the array is sorted, GPU-DFOR encodes the data with just 1.8 bits per int vs 7.8 bits per int used by GPU-FOR and 8 bits per int used by GPU-RFOR. The performance of GPU-DFOR (Fig. 8 (b)) is bounded by shared memory bandwidth which results in a performance gap in comparison to GPU-FOR.

GPU-RFOR and RLE achieve a very good compression ratio when the number of distinct values is less than 2^{22}. Beyond that, the compression ratio of these two schemes worsened with GPU-RFOR still better than RLE due to the use of FOR. GPU-RFOR is also 2.5× faster than RLE due to the use of tile-based decompression (Fig. 8(b)). Decompressing RLE is a 4-step process as explained in Section 6, which even after optimizations is similar to GPU-FOR making 4 passes over an array of size n. Across the entire range, we can see that GPU-RFOR is a better choice when the number of distinct values is low while GPU-DFOR is a better choice when the number of distinct values is high in a sorted dataset.

For D2 (Fig. 8 (c-d)), we can make the same general observations. When using GPU-FOR/GPU-DFOR, each block's entries generally lie within 3 standard deviations of the mean and occasional occurrence of a value outside this range does not move the compression rate significantly. For mean greater than 2^{16}, the bit-aligned schemes achieve 3× reduction in storage footprint compared to the other schemes and showcases the use of FOR.

For D3 (Fig. 8 (e-f)), we see that the bit-aligned schemes can adapt to change in skew and achieve both better compression rate and lower runtime compared to NSF and NSV. NSV is better at adapting to skew compared to NSF, however its performance is significantly worse compared to all the other schemes. Decoding NSF suffers from the same issues that affect RLE, it requires multiple steps that lead to multiple reads and writes, the decoding can't be inline with query execution and it requires buffer space for intermediates.

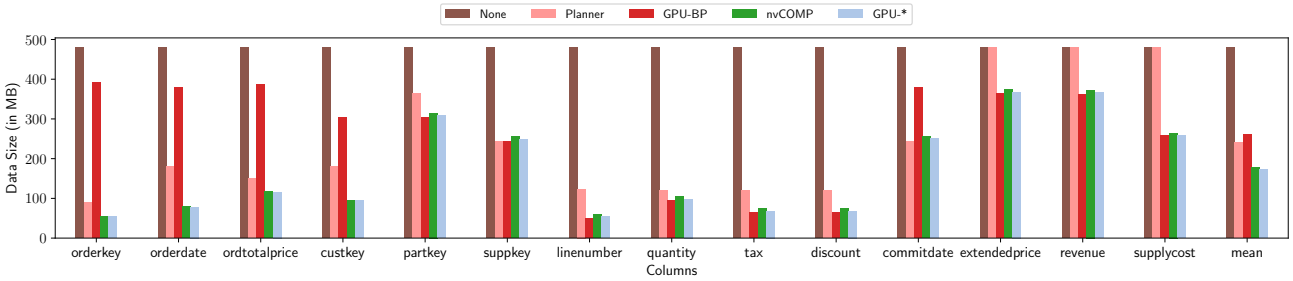
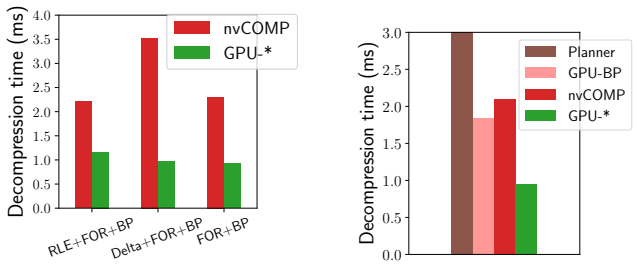


Figure 9: Compression Waterfall for Star Schema Benchmark columns



(a) GPU-* vs nvCOMP **(b) GPU-* vs Existing systems**

Figure 10: Average Decompression Performance across SSB columns

Overall, this experiment shows that compared to existing schemes, our bit-packing schemes are robust under various data distributions.

9.4 Performance on SSB

For the full query evaluation, we use the Star Schema Benchmark (SSB) [35] which has been widely used in various data analytics research studies [20, 30, 45, 49]. SSB is a simplified version of the more popular TPC-H benchmark. It has one fact table *lineorder* and four dimension tables *date, supplier, customer,* and *part* which are organized as a star schema. There are a total of 13 queries in the benchmark, divided into 4 query flights. Query flight 1 (q1.1, q1.2, and q1.3) are selection queries whereas query flights 2, 3, and 4 are join queries. In our experiments we run the benchmark with a scale factor of 20 which will generate a fact table with 120 million tuples. To enable efficient query execution in GPU, we dictionary encode the string columns into integers prior to data loading and the queries run directly on dictionary-encoded values. The total dataset size is around 13GB.

In this experiment, we compare the performance of our compression schemes with four existing systems:

- **Planner:** We use the compression planner from [18] to generate compression plans for all the column. Note that this work does not support bit-packing-based schemes. For decompression, this work uses *cascading decompression* as explained in Section 3.
- **GPU-BP:** Mallia et al. [33] introduced two bit-packing schemes: GPU-BP and GPU-VByte. We compare against GPU-BP in this work since it has superior compression ratio and decompression performance. This scheme, however, only consists of a single compression layer without the use of FOR, Delta, or RLE. It also lacks optimization techniques which we introduced in Section 4.2.
- **nvCOMP:** We compare against the cascaded compression scheme in nvCOMP. Despite having various schemes such as bit-packing, delta, and RLE, nvCOMP does not support end-to-end pipelining of multiple decompression steps with query execution.
- **OmniSci:** A commercial GPU database system. OmniSci only uses dictionary encoding to encode the string column and therefore has the same compression scheme as None.

Figure 9 shows the column sizes after the compression for each system. GPU-* is the hybrid scheme that chooses between GPU-FOR, GPU-DFOR, and GPU-RFOR as explained in Section 8. Each column in SSB has different properties. For example, lo_orderkey is a sorted column with high average run length (similar to D1 in Section 9.3); lo_orderdate, lo_ordtotalprice, and lo_custkey are unsorted but also has high average run length. On average, GPU-* manages to reduce the total memory footprint by 2.8× compared to no compression (None).

Comparing to GPU-BP, GPU-* achieves 50% better compression rates. This is because GPU-BP only supports a single bit-packing layer without frame-of-reference. Therefore, GPU-BP has poor performance for columns with RLE pattern such as lo_orderkey, lo_orderdate, lo_ordtotalprice, and lo_custkey. GPU-BP also performs poorly on date columns such as lo_commitdate due to absence of frame-of-reference in GPU-BP.

Comparing to Planner, GPU-* achieves 40% better compression rates and outperforms Planner across all columns. Planner has poor performance on columns with large random integers such as lo_extendedprice, lo_revenue, and lo_supplycost. These columns can only be compressed using bit-packing schemes which are not supported by Planner. For columns with certain distribution (lo_orderkey, lo_orderdate, lo_ordtotalprice, and lo_custkey), Planner performs well as it is distribution-aware. GPU-*, however, still outperforms Planner on every column.

nvCOMP and GPU-* achieves similar compression rates across all columns while GPU-* is slightly superior by 2%. This is due to the fact that both of these schemes support the same set of compression schemes. It utilizes RLE based scheme for columns with consecutive runs of values and FOR based scheme for columns with large random integers. The 2% gain for GPU-* comes from our more compact data format which requires less metadata and therefore could store the compressed data more efficiently. Despite supporting the same compression schemes, we will show in the next experiment how GPU-* is superior in decompression performance and query running time compared to nvCOMP.

Figure 10a shows a detailed one-on-one decompression time comparison between different schemes in nvCOMP and GPU-*. It can

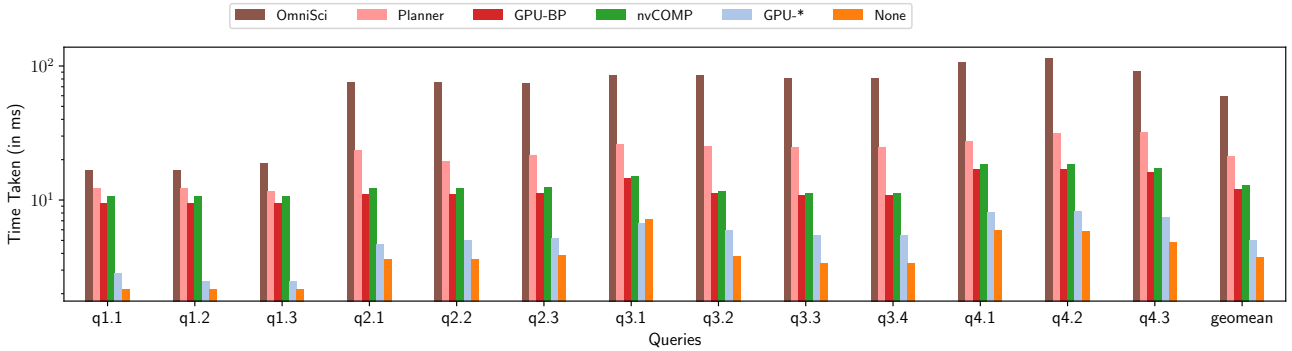


Figure 11: Performance on Star Schema Benchmark Queries

be seen from this figure that despite supporting the same set of compression schemes, GPU-* outperform nvCOMP in each scheme. GPU-FOR outperform nvCOMP(FOR+BitPacking) by 2.4×, GPU-DFOR outperform nvCOMP(Delta+FOR+BitPacking) by 3.5×, and GPU-RFOR outperform nvCOMP(RLE+FOR+BitPacking) by 2×.

Figure 10b shows the geomean of decompression performance across all columns for each system. For nvCOMP and GPU-*, we choose the scheme which results in the best performance thereby resulting in a smaller performance gap compared to Figure 10a. Overall, GPU-* outperforms Planner, GPU-BP, and nvCOMP by 5.5×, 2×, and 2.2× respectively. The performance gap is due to the use of tile-based decompression and our optimization techniques presented in Section 4.2.

Figure 11 shows the end-to-end query performance across all systems. For the runtime comparison, we compare the performance of Crystal without any encoding (None) against Crystal with the decompression routines of GPU-* and nvCOMP. We also compare our system with Planner, GPU-BP, and OmniSci.

Comparing GPU-* with None, None is 1.35× faster. This is to be expected since there are overhead of decompressing the columns, especially the lo_orderdate and lo_custkey columns which utilizes GPU-RFOR. These two columns are used in almost every query in SSB. Comparing GPU-* with OmniSci, GPU-* is 12× faster. This result is consistent with result from prior works [40]. This is because the query execution engine in OmniSci does not support the tile-based execution model. Comparing GPU-* with other compression schemes (Planner, GPU-BP, and nvCOMP), GPU-* outperform these schemes by 4×, 2.4×, and 2.6× respectively. Not only decompressing individual columns by GPU-* is faster, all these schemes cannot decompress the columns inline with the query execution. Therefore, these schemes will have to decompress each individual column one by one before executing each query, which results in a much higher total query runtime.

Overall, our experiment shows that GPU-* can achieve similar or better compression rates to the best state-of-the-art compression schemes in GPU (i.e., nvCOMP) while being 2.2× and 2.6× faster in decompression speed and query running time. For GPU-based DBMS systems, using GPU-* results in significantly lower storage footprint with minimal performance degradation.

9.5 GPU as a Coprocessor

Many systems use GPU strictly as a coprocessor [20, 26, 32, 49]. These systems move data from CPU to GPU across an interconnect like PCIe or NVLink when processing every query. The compression ratio of each compression scheme would play a big role as runtime is bound by the time taken to ship data over the interconnect (transfer time). We have shown that GPU-FOR/GPU-DFOR/GPU-RFOR achieve the best compression rate across a variety of data distributions and using them would reduce the amount of data moving across the slow interconnect thereby reducing transfer time. To evaluate this, we ran one SSB query from each flight (q1.1, q2.1, q3.1, q4.1) with data initially stored on the CPU. The encoded data will then be shipped to GPU over 12GB/s bidirectional PCIe. GPU will then decode the data and execute the query. In this experiment, we will compare the query performance of GPU-* against uncompressed data (None). Figure 12 shows that after applying compression, the query runtime is 2.3× faster. This shows that our compression scheme is useful when the working set is sharded across CPU-GPU or potentially multiple GPUs.

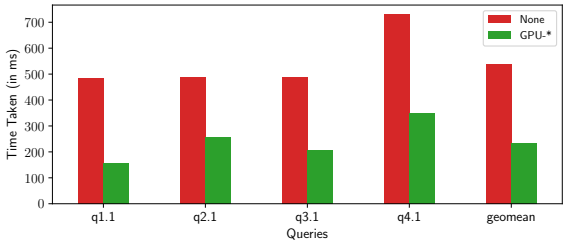


Figure 12: Benefit of data compression in GPU as coprocessor model

10 CONCLUSION

This paper advances the state-of-the-art for GPU data compression by introducing (1) *tile-based decompression* which allows a database to decompress a cascade of compression schemes in a single pass and (2) GPU-FOR, GPU-DFOR, and GPU-RFOR bit-packing based compression schemes and efficient massively parallel bit unpacking routines for them. Our evaluation shows that our schemes can achieve similar compression rates to the best state-of-the-art compression schemes in GPU while being 2.2× and 2.6× faster in decompression speed and query running time.

ACKNOWLEDGEMENT

This work was supported (in part) by the National Science Foundation PPoSS-2028818, Oracle External Research Office, and Oracle Cloud credits.

REFERENCES

[1] [n. d.]. BlazingSQL. https://blazingsql.com.
[2] [n. d.]. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[3] [n. d.]. GPU-Accelerated Supercomputers Change the Balance of Power on the TOP500. https://bit.ly/2UcBInt.
[4] [n. d.]. Kinetica. https://kinetica.com/.
[5] [n. d.]. NVComp. https://github.com/NVIDIA/nvcomp.
[6] [n. d.]. NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/.
[7] [n. d.]. OmniSci. https://omnisci.com.
[8] [n. d.]. Opencl. https://www.khronos.org/opencl/.
[9] [n. d.]. Thrust. https://thrust.github.io/.
[10] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 671–682.
[11] Vo Ngoc Anh and Alistair Moffat. 2004. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval* 8 (2004), 151–166.
[12] Vo Ngoc Anh and Alistair Moffat. 2010. Index Compression Using 64-Bit Words. *Softw. Pract. Exper.* 40, 2 (feb 2010), 131–147.
[13] Guy E Blelloch. 1989. Scans as primitive parallel operations. *IEEE Transactions on computers* 38, 11 (1989), 1526–1538.
[14] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.
[15] D. Cutting and J. Pedersen. 1989. Optimization for Dynamic Inverted Index Maintenance *(SIGIR '90)*. Association for Computing Machinery, New York, NY, USA, 405–411. https://doi.org/10.1145/96749.98245
[16] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses).. In *EDBT*. 72–83.
[17] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining*. New York, NY, USA, 1–1. http://doi.acm.org/10.1145/1498759.1498761
[18] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 670–680.
[19] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 31–46.
[20] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1603–1618.
[21] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*. IEEE, 370–379.
[22] Naga Govindaraju et al. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*.
[23] Mark Harris, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU gems* 3, 39 (2007), 851–876.
[24] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *SIGMOD*.
[25] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB* (2013).
[26] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* (2013).
[27] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.
[28] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *DaMoN*.
[29] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.
[30] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.
[31] Yinan Li and Jignesh M Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 289–300.
[32] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705
[33] Antonio Mallia, Michał Siedlaczek, Torsten Suel, and Mohamed Zahran. 2019. GPU-Accelerated Decoding of Integer Lists *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 2193–2196. https://doi.org/10.1145/3357384.3358067
[34] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.
[35] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252.
[36] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. *MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254
[37] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (Melbourne, VIC, Australia) *(DaMoN'15)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/2771937.2771943
[38] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 708–720. https://doi.org/10.14778/3436905.3436927
[39] Ran Rui and Yi-Cheng Tu. 2017. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM, 17.
[40] Anil Shanbhag, Xiangyao Yu, and Samuel Madden. 2020. A Study of the Fundamental Performance Charecteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data*. ACM.
[41] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. *Hardware-conscious Hash-Joins on GPUs*. Technical Report.
[42] Evangelia A Sitaridi and Kenneth A Ross. 2013. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. ACM, 4.
[43] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *SIGMOD*. ACM.
[44] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 993–1008. https://doi.org/10.1145/3035918.3064007
[45] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment* 7, 11 (2014), 1011–1022.
[46] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment* 2, 1 (2009), 385–394.
[47] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational joins on GPUs: A closer look. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2663–2673.
[48] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Proceedings of the 18th International Conference on World Wide Web* (Madrid, Spain) *(WWW '09)*. Association for Computing Machinery, New York, NY, USA, 401–410. https://doi.org/10.1145/1526709.1526764
[49] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *PVLDB* (2013).
[50] Jiangong Zhang, Xiaohui Long, and Torsten Suel. 2008. Performance of Compressed Inverted List Caching in Search Engines. In *Proceedings of the 17th International Conference on World Wide Web* (Beijing, China) *(WWW '08)*. Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1367497.1367550
[51] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.
[52] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 59–59.
[53] M. Zukowski, S. Heman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*. 59–59. https://doi.org/10.1109/ICDE.2006.150