Database Compression on GPUs

ABSTRACT

GPUs are increasingly used for high-performance and interactive data analytics workloads. While GPUs provide massive parallelism to accelerate computation, a key constraint in using GPUs is the limited memory capacity. In this paper, we investigate bit-packing based compression schemes for storing data on GPU. We present efficient decompression routines and a series of optimizations for GPU that allow us to decode using a single pass over the data, inline with query execution and at memory bandwidth speed. Our evaluations on many microbenchmarks and on the Star Schema Benchmark, a popular analytical workload, show that our approach yields significant storage savings with very little impact on query performance. These results show that our algorithms make it practical to use bit-packing for compression on GPU for the first time.

1 INTRODUCTION

In the past decade, special-purpose graphics processing units (GPUs) have evolved into general-purpose computing devices, with the advent of general-purpose parallel programming models, such as CUDA [3] and OpenCL [8]. Because of GPUs' high compute power, they have seen significant adoption in deep learning and high performance computing [4]. GPUs also have significant 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 32 GB of HBM which is capable of delivering up to 1.2 TBps of memory bandwidth and 16 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. This is sufficient for many analytical tasks; for example, one machine could host several weeks of a large online retailer's (with say 100M sales per day) sales data (with 100 bytes of data per sale) in GPU memory, the on-time flight performance of all commercial airline flights in the last few decades, or the time, location, and (dictionary encoded) hash tags used in every of the several billion tweets sent over the past few days.

Several commercial systems, including Omnisci [7], Kinetica [5], and BlazingDB [2], 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 32 GB of memory which is used both to cache the working set and as scratch memory for query execution. GPU memory is 6× more expensive compared to CPU RAM [26] and going outside a single GPU's memory incurs a penalty. Therefore, data compression is critical. Currently, GPU-based systems use simple compression schemes like fixed-width dictionary encoding and run-length encoding (RLE) [7, 29] similar to CPU-based in-memory analytics systems, and decompressing on-the-fly during query execution. To the best of our knowledge, no GPU database today uses bit-packed schemes which have been shown to achieve the best compression ratios [14] on the CPU but are non-trivial to decode in parallel across thousands of threads.

In this paper, we introduce two new efficient compression schemes for GPUs: `GPU-FOR` which does bit-packing in conjunction with Frame-Of-Reference (FOR) and `GPU-DFOR` which uses delta encoding with bit-packing and FOR. Both these schemes are designed to offer improved compression ratios while still being able to decode them in parallel across thousands of threads at close to memory bandwidth speeds. `GPU-FOR` partitions the data into blocks, in each block encoding integers with the minimum bit size needed to represent a value in the block. It works well with uniform data and can handle skew. `GPU-DFOR` first delta-encodes a block of integers before using `GPU-FOR`. It is suited for sorted and semi-sorted columns.

Past works [15, 23] have looked at delta encoding, FOR, and variable length byte-aligned packing (NSV). These works found that achieving the minimum space cost by using a combination of compression schemes (e.g. delta+NSV) can degrade performance as intensive decompression overburdens the GPU. Hence previous work deemed these schemes GPU-unfriendly. The reason for bad performance was that these works treated threads on the GPU as independent execution units and hence required multiple passes to decode the compressed data. For example: to use a column encoded using delta encoded variable length byte-aligned packing in a query, these systems would first run a prefix-sum primitive to unpack the variable-length byte-packed data and write it to global memory, then do a second pass prefix-sum primitive to delta decode the data which is written to global memory, and finally the query kernel would read from global memory the unpacked column — the intermediate data is read/written to global memory multiple times. In our work, we treat a *thread block* as the basic execution unit and each thread block collectively decodes one block of encoded entries. By treating the thread block as the basic unit, we are able to cache a block of data in on-chip caches and inline the multiple steps involved in decoding into a single kernel, resulting in a single pass over the data. We present a series of optimizations that enable us to decode at close to memory bandwidth speed. The performance of our schemes simplifies the choice of a compression scheme to encode a column — we choose the scheme with the smallest storage footprint. It eliminates the need for sophisticated compression planners used by past works to choose the right compression scheme.

To show that our compression schemes perform well and and significantly reduce the storage footprint of GPU-based systems, we present an integration of `GPU-FOR` and `GPU-DFOR` into the Crystal framework [26]. We encapsulate the 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. In the end, we find that our compression schemes can reduce the storage footprint by up to 10× on certain data distributions; on the Star Schema Benchmark, the proposed scheme achieves

50% reduction in storage compared to no compression and 37% compared to existing GPU compression schemes with almost no impact on performance.

In summary we make the following contributions:

- We present two bit-packing based compression schemes `GPU-FOR` and `GPU-DFOR` that can be used to store data compactly on the GPU.
- We develop a series of optimizations that allow us to decode the encoded data on-the-fly at close to memory bandwidth speed.
- We present an integration of `GPU-FOR` and `GPU-DFOR` into the `Crystal` framework and demonstrate ease of use.
- We present an evaluation on multiple synthetic benchmarks and on the Star Schema Benchmark (SSB). On SSB our schemes can achieve significant (37% on SSB) space savings and just 4% loss in performance compared to existing methods.

The rest of the paper is organized as follows: related work and background are discussed in Section 2. We present the data format and the unpacking implementation on the GPU for `GPU-FOR` and `GPU-DFOR` in Section 3 and Section 4, respectively. In Section 5, we discuss the database integration. In Section 6, we evaluate the performance and compression ratio of binary packing against other schemes on the GPU. Finally, we conclude in Section 7.

2 BACKGROUND

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

2.1 GPU Architecture

Many database operations executed on the GPU are performance bound by the memory subsystem (either shared or global memory) [29]. In order to characterize the performance of different algorithms on the GPU, it is, thus, critical to properly understand its memory hierarchy.

Figure 1 shows a simplified hierarchy of a modern GPU. The lowest and largest memory in the hierarchy is the *global memory*. A modern GPU can have global memory capacity of up to 32 GB with memory bandwidth of up to 1200 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 an 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 result in neighboring locations being accessed.

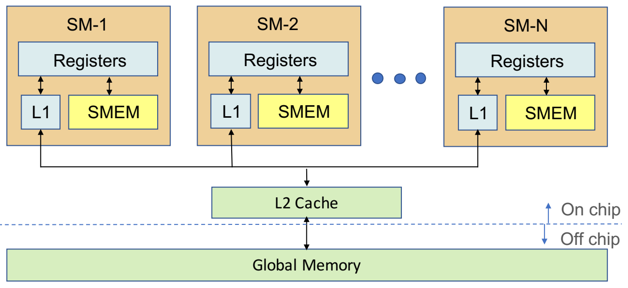


Figure 1: GPU Memory Hierarchy

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.

2.2 Compression Techniques

Compression techniques are heavily exploited in modern column-store databases for efficient query processing. These databases store relational data in a decomposition storage model (DSM) [13] where a n-attribute relation is replaced by n arrays, one for each attribute. Since each attribute is stored separately as a sequence of values, we can use lossless compression techniques to store them compactly. Based on the compute intensity of decompression, lossless compression techniques are categorized into two buckets: *lightweight* and *heavyweight*. Lightweight algorithms are mainly used in in-memory column stores while heavyweight algorithms like Huffman [21] and Lempel Ziv [30] (together with lightweight techniques) are used in disk-based column stores. In this paper we focus on lightweight techniques. We show later in Section 6 that most of the compression gains are achieved with just lightweight techniques for our workload.

There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [18, 31], delta coding (DELTA) [22], dictionary compression (DICT) [11, 31], run-length encoding (RLE) [11], and null suppression (NS) [11].

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 used 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.

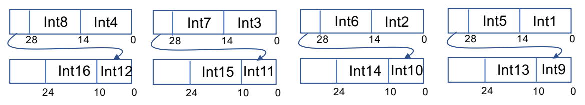


Figure 2: Bit packing with vertical data layout

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, and word-aligned NS encode as many integers as possible into 32/64-bit words. The NS algorithms also differ in their data layout. We distinguish between *horizontal layout* and *vertical layout*. 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.

Researchers have proposed a number of NS algorithms for compressing columns in main memory database management systems (DBMS) on CPUs. SIMD-Scan [28] stores column values in a tightly bit-packed horizontal layout, ignoring any byte boundaries and uses SIMD instructions to scan a column of entries. For example, to store a column of 11 bits in memory, the first value is put in the 1st to 11th bits whereas the second value is put in the 12th to 22nd bits and so on. Such a bit-packed layout incurs overhead to unpack the data before processing and does not saturate memory bandwidth. In the example above, several SIMD instructions have to be spent to align eight 11-bit values with the eight 32-bit banks of a register. Li and Patel [24] proposed the Horizontal Bit-Parallel (HBP) and Vertical Bit-Parallel (VBP) storage layouts. HBP is a word-aligned layout that packs as many entries as possible into the same word. In the example, 5 11-bit entries would be packed in 64-bit word and the remaining 9 bits wasted. Due to padding, HBP does not achieve compact storage. In VBP, if a processor uses S-bit words, it groups S entries of k bits each into k processor words such that the ith processor word contains the ith bit of each entry. Reconstructing/looking-up a value under the VBP layout is expensive though. That is because the bits of a value are spread across k words. ByteSlice [16] improves on VBP. It groups S/8 entries to: (1) an S-bit memory word contains bytes from S/8 different values; (2) bytes of a k-bit entry are spread across $\lfloor k/8 \rfloor$ words. ByteSlice stripes by byte, hence while scan is faster than VBP, the storage footprint is also larger than VBP.

The best performing NS scheme that also achieves good compression ratios is `SIMD-BP128` [22] (and its variants). 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 2 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. Each lane has 32 integers. The used bit width is stored in a single byte, whereby 16 of these bit widths are followed by 16 compressed blocks. SIMD-BP128 achieves better compression than Li and Patel [24] and ByteSlice, and can be decoded at memory bandwidth speed.

In Section 3.3, we discuss why directly translating SIMD-BP128 to the GPU leads to bad performance. GPU-FOR uses a horizontal layout similar to SIMD-Scan, however the decoding algorithm is novel and contains optimizations tailored to the GPU architecture. GPU-FOR achieves a better compression ratio than HBP as it does not use padding. We think VBP is not well suited for the GPU architecture as it is not easily vectorizable. In our evaluation, we compare against byte-aligned null suppression which achieves the same compression as ByteSlice.

2.3 Query Execution on GPUs

With the slowing of Moore's Law, CPU performance has stagnated. In recent years, researchers have begun to explore heterogeneous computing as a way to overcome the scaling problems of CPUs and to continue to deliver interactive performance for database applications. Ocelot [20] provides a hybrid analytical engine as an extension to MonetDB. YDB [29] is a GPU-based data warehousing engine. HippogriffDB [23] used GPUs for large scale data warehousing where data resides on SSDs. More recently, HorseQC [17] proposes pipelined data transfer between CPU and GPU to improve query runtime. All these works have focused on using GPU as a coprocessor, where data is stored primarily on the CPU side and moved to the GPU at query execution time. Recent work [26] has shown that GPU as a coprocessor is slower than just running queries on the CPU, instead a better model is to store the working set directly on the GPU memory. The memory capacity of GPUs has increased significantly over the years, today a GPU can have up to 32GB of High Bandwidth Memory (HBM), which is likely to further increase as HBM technology improves. It is possible to attach up to 20 GPUs to a single socket CPU allowing the user to aggregate enough memory to store large datasets. Commercial systems like Omnisci [7], Kinetica [5], and BlazingDB [2] use this philosophy and aim to provide real-time analytical capabilities by using GPUs to store large parts of the working set. This paper focuses on implementing compression schemes efficiently on the GPU so that more data can be cached on the GPU with minimal performance degradation. The compression schemes are beneficial to systems that use GPU as a coprocessor as well as they help reduce the data transfer time between CPU and GPU (see Section 6.5).

Researchers have looked at data compression for GPUs in the past. Yuan et al. [29] studied effect of RLE and DICT compression on query execution. Fang et al. [15] extended the work and studied a larger set of compression schemes like DICT, FOR, RLE, Null Suppression with Fixed Length (NSF), and NS with Variable Length (NSV). In NSF, all values are encoded with the number of bits being multiple of 8 to ensure output values are byte-aligned. As GPU is byte-addressable, they claimed that this achieves good decompression performance. NSV uses a variable number of bytes per entry. For each value, it stores the number of bytes used to store value (1, 2, 3, or 4) followed by bytes of the output value. Jing et al. [23] also studied compression, however they reused the implementation of Fang et al.. A key issue with past work is that they treated a cascade of compression schemes as independent layers. For example, data compressed with DELTA + FOR + NSF would run the DELTA decoding kernel, followed by FOR kernel and NSF kernel, finally using the uncompressed column in the actual query execution. Each kernel would read and write the entire column to global memory. As a result, cascades of compression schemes would achieve lower performance while likely having better compression ratios introducing a cost-benefit problem. Hence, both works proposed compression planners that based on a cost-model decided which cascades to use to minimize space cost while also ensuring query performance is not impacted significantly.

Compared to past work, this paper focuses on building efficient decompression routines that can decode (variable length) *bit-aligned* null suppression schemes. Past works have looked only at byte-aligned schemes (NSF/NSV) and didn't evaluate bit-aligned schemes that we describe in this paper which we show achieve better compression ratios. We also show our compression schemes (which are a cascade of basic compression schemes) can be decoded in a single pass over the data and inline during query execution at close to memory bandwidth speed. This eliminates the need for complicated compression planners. Commercial systems currently only implement fixed length dictionary encoding and would also benefit from our work.

3 FAST BIT UNPACKING

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 query time, the query executor will need to decompress data and run the query on the decompressed data. Hence, optimizing the performance of decompression is critical for analytic workloads. In contrast, data is only compressed once as it is loaded into the GPU, so optimizing the performance of compression is not as important. In the rest of the section, we first describe the bit-packed representation we use and then describe the kernel implementation on the GPU. We present a series of optimizations which allow us to decode bit-packed data while saturating memory bandwidth.

3.1 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.

In bit-packing, a sequence of values is encoded with fixed bit size b. 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*. We use blocks of 128 integers. 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

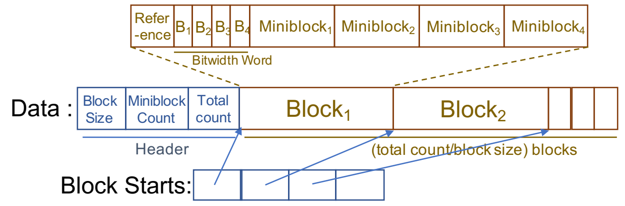


Figure 3: GPU-FOR Data Format

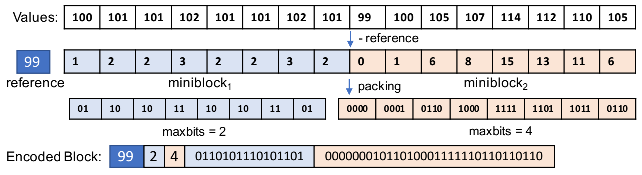


Figure 4: Example encoding with GPU-FOR

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. Each bitwidth can be stored in 1 byte. We store the bitwidths of 4 miniblocks at the start of the block using a single integer. The choice of the size of the miniblock and the reason for storing bitwidths together is to ensure they align on 32-bit boundaries. This allows us to use 32-bit arithmetic while decoding and makes shared memory accesses (which are aligned to 32-bit boundaries) efficient.

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 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.

The key difference between GPU-FOR and state-of-the-art bit-packing algorithms for CPUs like SIMD-BP128 is that GPU-FOR uses horizontal data layout to store the entries while SIMD-BP128 uses vertical data layout. We will discuss later in Section 3.3 the reason for this choice.

3.2 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. These optimizations are

Algorithm 1: Fast Bit Unpacking on GPU — The following code runs on each of the 128 threads within a threadblock 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      $miniblock\_offset \mathrel{+}= (bitwidth\_word \;\&\; 255)$;
10     $bitwidth\_word \gg= 8$;
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 $element\_block = data\_block[start\_intindex] \mid$
     $(((\text{uint64})data\_block[start\_intindex + 1]) \ll 32)$;
16 $start\_bitindex = start\_bitindex \;\&\; (32\text{-}1)$;
17 uint $element = (element\_block \;\&\; (((1 \ll bitwidth) - 1) \ll$
     $start\_bitindex)) \gg start\_bitindex$;
18 $item = reference + element$;
```

inspired by similar optimizations for other algorithms. However, to the best of our knowledge, none have been applied in the context of parallel decompression of bit-packed data. 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 6.1.

Base Algorithm: Algorithm 1 shows the pseudocode that would run in parallel on each thread (n threads are allocated for n-element dataset). Each threadblock (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`, 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 b 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 `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 5, 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. We use a number of optimizations detailed below 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 threadblock touch one data block, in this optimization, we load the entire block into shared memory once at the start of the operation. Each threadblock 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

The granularity of reads from global memory is 128 bytes [26]. Maximum bandwidth is achieved when warps' (groups of 32 threads) accesses to global memory result in neighboring locations being accessed. The best case scenario is when 32 threads access a 4-byte integer array of size 32, resulting in a perfect 128 byte access. 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 threadblock 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 threadblock. Each threadblock is assigned $D(= 2/4/8/16/32)$ data blocks to process. At the start, each threadblock 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 result of the decompression is stored in registers. While increasing D reduces the number of irregular accesses, the number of registers required and the shared memory requirement increases proportional to D. Each thread on the GPU has limited amount of registers and shared memory available. On an Nvidia V100 GPU, each thread can use 65 registers and

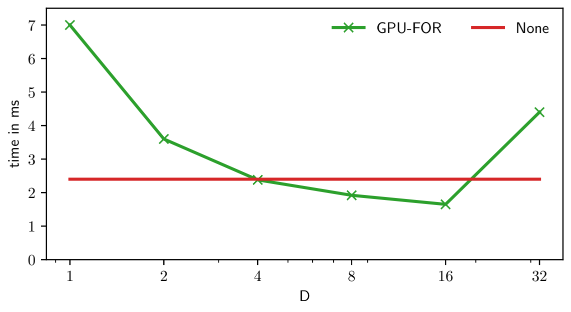


Figure 5: Decompression performance with varying number of data blocks per thread block (D)

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 6.4) 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 (for example, if the bitwidths used by 4 miniblocks within a block are 7, 8, 9, and 10, the miniblock offsets are thus 0, 7, 15, and 24); (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. Algorithm 2 shows the pseudocode for the optimization. It runs on the first $D * 4 = 16$ threads (i.e. `thread_id` $\in [0, 16)$). All the array prefixed by `s_` are in shared memory. We start by assigning each thread one miniblock offset/bitwidth pair to compute (lines 1-2). Each such thread loads the corresponding bitwidth word (line 3) and computes a prefix sum over it using bitshift arithmetic (line 4). Finally we extract the relevant offset and bitwidth for the miniblock and store it in shared memory (lines 5-8). 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.

Algorithm 2: Precomputing Miniblock Offset — The following code runs on each of the first $4 \times D$ threads within a threadblock.

```
Input  : int[] $s\_block\_starts$; int[] $s\_data$;
         int $thread\_id$
Output : int[] $s\_offsets$; int[] $s\_bitwidths$;
1 int $block\_index = thread\_id$ / 4;
2 int $miniblock\_index = thread\_id$
3 uint $bitwidth\_word = s\_data[s\_block\_starts[block\_index] - s\_block\_starts[0] + 1]$;
4 uint $miniblock\_offsets = (bitwidth\_word \ll 8) + (bitwidth\_words \ll 16) + (bitwidth\_word \ll 24)$;
5 uint $miniblock\_offset = (miniblock\_offsets \gg (miniblock\_index \ll 3))$ & 255;
6 uint $bitwidth = (bitwidth\_word \gg (miniblock\_index \ll 3))$ & 255;
7 $s\_offsets[thread\_id] = miniblock\_offset$;
8 $s\_bitwidths[thread\_id] = bitwidth$;
```

3.3 Discussion

GPU-FOR vs SIMD-BP128: As described earlier in Section 2.2, there are two variants of bit-packing based on the data layout: *horizontal* and *vertical*. On the CPU, the fastest bit-packing/unpacking routine is SIMD-BP128 [22]. SIMD-BP128 stores integers in a vertical layout. It 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 lanes as equivalent to a GPU threads 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 2.7x slower than GPU-FOR.

On the GPU, vertical packing like in 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. Furthermore, 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 6.4) with columns encoded using GPU-FOR vs with columns encoded using GPU-SIMDBP128. The query uses 4 columns. The performance with GPU-SIMDBP128 was 14x 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.

A CPU has low compute to bandwidth ratio and each CPU core has a large L1 cache. This leads to bitpacking schemes with vertical layout like SIMD-BP128 (which has lower compute intensity but higher storage requirement) perform better than schemes with a horizontal layout like SIMD-Scan [28]. On the GPU, the compute to bandwidth ratio is higher and each GPU thread has limited resources. This results in bitpacking with horizontal layout like GPU-FOR performing better than using vertical layout on the GPU.

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.

4 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 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 decoding delta encoded data is equivalent to computing the prefix sum. Efficient parallel prefix sum routines have been proposed [19] 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 3.2 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 [15, 23] 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.

Note that delta encoding is used only for sorted or partially-sorted data e.g.: to encode the primary key and secondary keys in

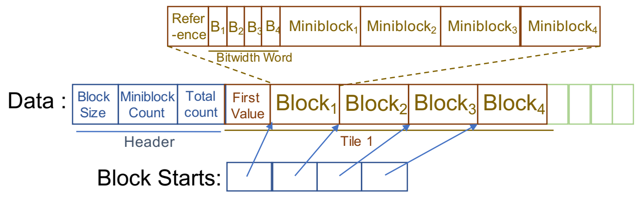


Figure 6: GPU-DFOR Data Format

databases, to encode posting lists in search workloads. Using it for unsorted data could lead to worse compression ratios compared to simply bit-packing the data. To illustrate this, consider a block of integers drawn uniform randomly from $[0, 32)$. The integers can be bitpacked with 5 bits. However the deltas will be in the domain $[-31, 31]$ and would require 6 bits per integer.

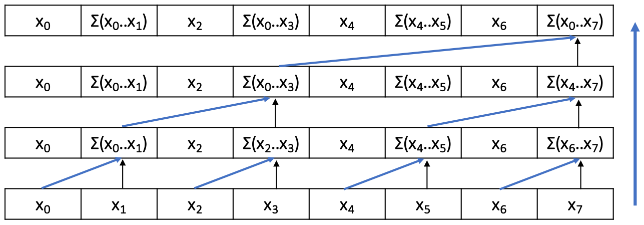
4.1 Data Format

Delta encoding the entire array as $x_0, \delta_1, \delta_2 ...$ makes it hard to decode in parallel as decoding the n^{th} block requires the $(n-1)^{th}$ block to have been decoded already. To enable parallel decoding, we build on GPU-FOR encoding scheme (Section 3.1) 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 threadblock). Figure 6 shows the data format. Encoding x integers generates $x - 1$ deltas. Hence 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.

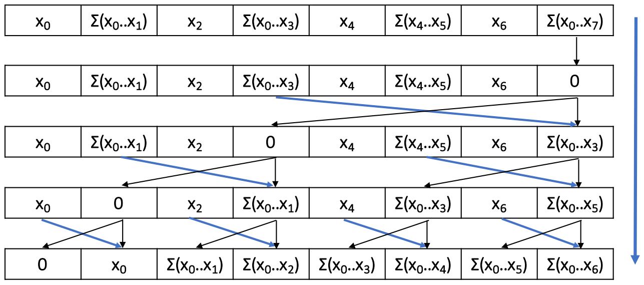
4.2 Implementation

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 3.2) 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 threadblock. 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. [12]. 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. We start with loading the computed deltas into shared memory to create a contiguous array of deltas for all D blocks. All operations in the algorithm are done in place on the array in shared memory. The algorithm consists of two phases: the up-sweep phase and the down-sweep phase. Each phase consists of a series of steps where each step is a set of additions in parallel across threads. The additions when visualised form a tree pattern as shown in Figure 7. In the up-sweep phase, we traverse the tree from leaves to root computing partial sums at the internal nodes of the tree. Figure 7a illustrates the up-sweep phase on an array with 8 elements. There are 3 steps. In each step, we do a set of additions in parallel across threads and synchronize the threads after each step. In the down-sweep phase, starting with the result of the up-sweep phase, we traverse back down the tree from the root, using the partial sums from the reduce


x0 | Σ(x0..x1) | x2 | Σ(x0..x3) | x4 | Σ(x4..x5) | x6 | Σ(x0..x7)
x0 | Σ(x0..x1) | x2 | Σ(x0..x3) | x4 | Σ(x4..x5) | x6 | Σ(x4..x7)
x0 | Σ(x0..x1) | x2 | Σ(x2..x3) | x4 | Σ(x4..x5) | x6 | Σ(x6..x7)
x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7

(a) Up-Sweep Phase


x0 | Σ(x0..x1) | x2 | Σ(x0..x3) | x4 | Σ(x4..x5) | x6 | Σ(x0..x7)
x0 | Σ(x0..x1) | x2 | Σ(x0..x3) | x4 | Σ(x4..x5) | x6 | 0
x0 | Σ(x0..x1) | x2 | 0 | x4 | Σ(x4..x5) | x6 | Σ(x0..x3)
x0 | 0 | x2 | Σ(x0..x1) | x4 | Σ(x0..x3) | x6 | Σ(x0..x5)
0 | x0 | Σ(x0..x1) | Σ(x0..x2) | Σ(x0..x3) | Σ(x0..x4) | Σ(x0..x5) | Σ(x0..x6)

(b) Down-Sweep phase

Figure 7: Illustration of Prefix Sum Algorithm

phase to build the prefix sum result in place on the array. Figure 7b shows the down-sweep phase for the example. We start by inserting a zero at the last entry (root of the tree). On each step, each node at the current level passes its own value to its left child, and the sum of its value and the former value of its left child to its right child. In the end, each thread reads D entries back into registers and returns it as a result which will be used in the rest of the query. There are a number of optimizations done to achieve good performance (e.g., using a technique called padding to break shared memory bank conflicts) that we do not touch upon. Interested reader can refer to [19] for more details.

Although prefix-sum has been used widely in libraries like Thrust [10], a key observation we make is that it is sufficient to do delta coding in each set of D blocks separately. This allows us to get away with doing prefix sum entirely within a threadblock in shared memory. Doing prefix sum over an entire array is much more expensive and involves multiple passes over data. It 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 [15, 23]. The bit unpacking and delta decoding share the same shared memory buffer. The total resource requirement of the kernel is D 4-byte entries in shared memory and D registers to store the output per thread.

Compared to decoding `GPU-FOR`, decoding `GPU-DFOR` involves significantly more operations in shared memory. When used to decompress the example dataset described earlier in Section 3, the above algorithm takes 4.45ms. This is approximately 2× slower than decompression of the dataset encoded with `GPU-FOR` while the compressed dataset size is only 6% larger. This is because the decompression of `GPU-DFOR` is bound by the shared memory bandwidth. `GPU-DFOR` does better than `GPU-FOR` on sorted and semi-sorted datasets. Consider a dataset of $n = 500$ million integers with entries

```
// Implements SELECT x FROM R WHERE y > v
// NT => NUM_THREADS
// IPT => ITEMS_PER_THREAD
template<int NT, int IPT>
__global__ void Q(int* x, int* y, int* out, int v, int* counter){
int tile_size = get_tile_size();
int offset = get_tile_offset();
__shared__ union Buffer {
  int col[NT * IPT];
  int out[NT * IPT];
} buffer;
int items[IPT];
int bitmap[IPT];
int indices[IPT];

BlockLoadInt<NT, IPT>(y+offset,items,buffer.col,tile_size);
BlockPredIntGT<NT, IPT>(items,buffer.col,cutoff,bitmap);
BlockBitmapLoadInt<NT, IPT>(x+offset,items,bitmap,tile_size);
BlockScan<NT, IPT>(bitmap,indices,buffer.col,
  num_selections,tile_size);

if(threadIdx.x == 0)
  o_off = atomic_update(counter,num_selections);

BlockShuffleInt<NT, IPT>(items,indices,buffer.out);
BlockStoreInt<NT, IPT>(buffer.out,out + o_off,num_selections);
}
```

Figure 8: Query Q0 Kernel Implemented with `Crystal`

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`. The runtime of `GPU-DFOR` is still 2× slower than `GPU-FOR` as it still does the same number of operations in shared memory. However, when used in a larger kernel, `GPU-DFOR` is faster than `GPU-FOR` as shared memory will likely not be the bottleneck in the larger kernel. We will discuss the performance characteristics in greater detail in Section 6.3.

5 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[1] and show how they can be used with an existing GPU analytical engine. In particular, we chose `Crystal` [26], an open-source GPU analytics framework developed recently.

`Crystal` is a library of templated CUDA device functions that implement the full set of primitives necessary for executing typical select-project-join-aggregation (SPJA) analytical queries. Crystal is based on the idea of a *tile-based execution model*. In such a model, instead of viewing each thread as an independent execution unit, a thread block is viewed as the basic execution unit with each thread block processing a tile of entries at a time. A tile is simply a collection of $NT \times IPT$ elements where NT is the number of threads in a threadblock and IPT is the number of items per thread. Previous work [26] has shown that SQL query operators and SPJA queries implemented with Crystal can saturate memory bandwidth and thereby deliver an-order-of-magnitude speedup compared to CPU-based implementations.

The pseudo code in Figure 8 shows how the following example selection query is implemented in `Crystal`.

```
Q0: SELECT x FROM R WHERE y > v;
```

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

The pseudo code uses the following block functions: BlockLoad loads a tile of data from global memory into the thread block (line 16). BlockPred applies the predicate to the tile and generates a bitmap (line 17). BlockBitmapLoad selects data from the tile using the bitmap (line 18). BlockScan implements hierarchical parallel prefix-sum within the tile (line 19). The atomic update in line 23 determines the offset to which to write the matched results in the output array. BlockShuffle reorders the selected items into a contiguous array (line 25). Finally, BlockStore stores data into the output array (line 26). The modular nature of Crystal allows users to write high performance kernel code easily, reduces boilerplate code and makes it easy to use non-trivial functions.

We have implemented the decompression routines for GPU-FOR and GPU-DFOR as CUDA device functions LoadBitPack and LoadDBitPack 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 in Figure 8 with LoadBitPack. Below are the changes involved to make the kernel operate on bit-packed data:

```
...
LoadBitPack<NT, IPT>(y.col, y.block_start,items,buffer.col,tile_size);
...
LoadBitPack<NT, IPT>(y.col, y.block_start,items,buffer.col,tile_size);
...
```

In this case IPT is the same as D, the number of blocks processed per threadblock. As can be seen from the example, the LoadBitPack device function encapsulates all the complexity and hides it from the end user. The user can run the query on compressed data by just changing a single line of code.

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. As we show in the next section, this does not lead to material impact on performance because: 1) granularity of access from global memory is 128B, as a result random accesses are less beneficial in the first place and 2) analytics queries are mostly scan oriented, hence reduction in data size reduces the total data read which often compensates for loss of efficiency in case of a selective filter. Note that this would not work well for OLTP workloads which are characterized by point accesses. GPUs' are in general not suited and not used for OLTP workloads.

Since the routines LoadBitPack and LoadDBitPack are ordinary device functions, they can be used directly in user's CUDA code in conjunction with other GPU frameworks like Thrust [10] and they can also be called directly from NVVM (a compiler internal representation based on LLVM IR designed to represent GPU compute kernels) [6].

6 EVALUATION

In this section, we compare the performance of the 4 main compression schemes used to store columns on the GPU:

- None: Data is stored as 4-byte integers.
- 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.
- GPU-FOR: Using bit-packing to compress the data following the algorithm discussed in Section 3.
- GPU-DFOR: Using delta encoding and bit-packing to compress the data following the algorithm discussed in Section 4.

In addition, on two microbenchmarks we also compare 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 used to handle skew.

GPU-FOR and GPU-DFOR are novel to this work and we use the decompression routines discussed previously to decode them. The rest of the compression schemes are from past works [7, 15, 23] and we use improved versions of decompression routines proposed by them. For each compression scheme, we report three different metrics:

Compression Rate: The average number of bits required for each integer after compression is applied (bits/int).

Aggregation Time: The time to load the compressed data from global memory, apply the decompression, sum the values, and store the total as 8-byte entry in global memory. We measure the raw throughput of the decompression algorithm in isolation.

Decompression Time: The time required to load the compressed data from global memory, apply the decompression algorithm, and store the uncompressed data to global memory. This runtime is more reflective of the performance of the decompression algorithm when it is part of a larger kernel.

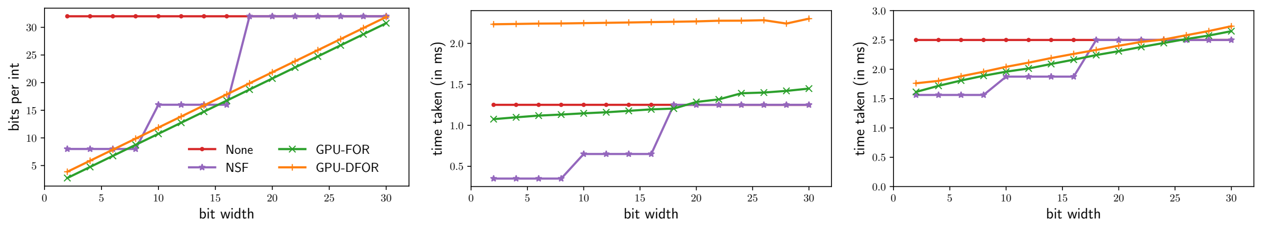
The rest of the section is organized as follows: we discuss the setup in Section 6.1. In Section 6.2 we evaluate the performance of the algorithms on synthetic dataset with varying bitwidths. In Section 6.3, we evaluate the impact of different data distributions. In Section 6.4, we evaluate the impact on performance on the Star Schema Benchmark. In Section 6.5, we discuss the case when GPU is used as a coprocessor. Finally, in Section 6.6, we discuss miscellaneous topics.

6.1 Setup

For the experiments, we use a virtual machine instance that has an Nvidia V100 GPU which is connected to the CPU via PCIe3. The Nvidia V100 GPU has 32 GB of HBM2 memory. The global memory read/write bandwidth is 880 GBps. The bidirectional PCIe transfer bandwidth is 12.8 GBps. The system is running on Ubuntu 16.04 and the GPU instance uses CUDA 10.0. In our evaluation, 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.

6.2 Performance with Varying Bitwidths

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 $[2^{i-1}, 2^{i})$ for i = 2,4,..,30. Within these ranges, the values are uniformly distributed.



(a) Compression Rate **(b) Aggregation Time** **(c) Decompression Time**

Figure 9: Performance of the different compression algorithms on uniform data with varying bit widths

Figure 9 (a-c) shows the results for the four compression algorithms. The bit-packing schemes achieve the finest possible granularity and thus can perfectly adapt to any bit width. Consequently, the compression rate is a linear function of the bit-width. 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 $[-2^i, 2^i)$ and require one additional bit; our experiments below show the benefit of GPU-DFOR on sorted data.

Figure 9b shows the aggregation time for the different 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. None and NSF saturate memory bandwidth. The performance of GPU-FOR is close to that of operating on uncompressed data. GPU-FOR does a significant amount of computation per entry which is roughly constant across varying bitwidths, compared to NSF which does a single copy followed by aggregation. Hence, in isolation GPU-FOR's performance does not vary much across bitwidths. GPU-FOR achieves bandwidth of 700 GBps which is close to the max memory bandwidth of 880 GBps. The performance of GPU-DFOR is bound by shared memory bandwidth. Hence, in isolation, GPU-DFOR performs worse than the other schemes for this experiment.

In Figure 9c, we can see that the decompression performance of the bit-packed schemes looks better as the compute load remains the same while we do one additional global memory operation (store) per integer. The performance of NSF is again a staircase pattern. GPU-FOR does slightly worse than NSF, however the worst case gap is 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 performance of GPU-DFOR is comparable to GPU-FOR in this case. In general, when GPU-DFOR is used as part of a larger kernel, shared memory may not be a bottleneck. Hence the additional shared memory passes done in GPU-DFOR to decode the deltas may not impact the actual performance of the kernel.

6.3 Dependence on Data distributions

To test the robustness of the compression schemes, we test their performance using three distributions. For each distribution, we maintain the array size n fixed at 250 million entries. The distributions are as follows:

- 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. For this distribution, we also compare against RLE.
- 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.

The results for D1 can be found in Fig. 10 (a-c). The bit-aligned algorithms GPU-FOR and GPU-DFOR 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. In the extreme case, when 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. The performance of GPU-FOR and GPU-DFOR (Fig. 10 (b-c)) is bound by shared memory bandwidth which results in a gap in performance being smaller than the gap in compression rate in comparison to the other two schemes. RLE achieves better compression rates compared to the bit-aligned algorithms when the number of distinct values is less than 2^{22}, beyond that GPU-DFOR does better. The key issue with RLE is that it is 3× slower to decompress compared to the bit-aligned schemes (Fig. 10(c)). Decompressing RLE is a 4-step process, which even after optimizations is similar to GPU-FOR making 4 passes over an array of size n. RLE decoding cannot be inlined with query execution and requires an additional memory buffer array of size n for storing intermediates. Interested readers can refer to [15] for the decompression algorithm. GPU-DFOR achieves better performance across the entire range compared to RLE and is competitive in terms of compression rate — hence it is a better choice.

For D2 (Fig. 10 (d-f)), we can make the same general observations. When using GPU-FOR/GPU-DFOR, each block's entries generally lies 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. 10 (g-i)), we see that the bit-aligned schemes can adapt to change in skew and achieve both better compression rate

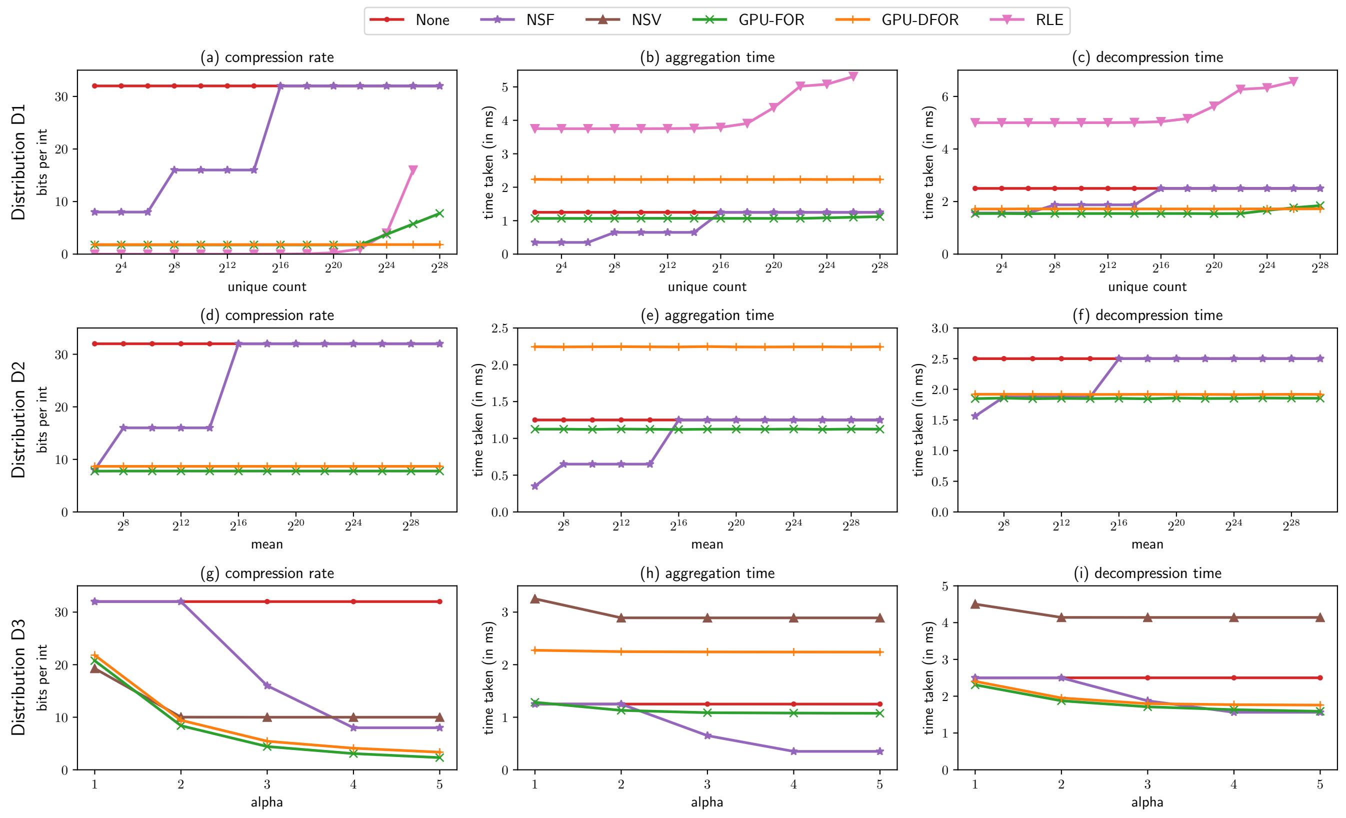


Figure 10: Comparison of compression schemes on different data distributions

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. The bit-aligned schemes are superior to NSV across all metrics.

6.4 Performance on SSB

For the full query evaluation, we use the Star Schema Benchmark (SSB) [25] which has been widely used in various data analytics research studies [17, 23, 27, 29]. 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, part* which are organized as a star schema. There are a total of 13 queries in the benchmark, divided into 4 query flights. In our experiments we run the benchmark with a scale factor of 20 which will generate the fact table with 120 million tuples. The total dataset size is around 13GB.

Figure 11 shows the column sizes after compression using the different encodings. Between GPU-FOR, GPU-DFOR, and NSF, GPU-FOR achieves the lowest storage footprint for all columns except lo_orderkey on which GPU-DFOR does better. As discussed before, GPU-DFOR does better on sorted columns. In total, using GPU-FOR achieves a 37% reduction in data size compared to NSF and 52% reduction compared to None. In additional to the schemes above, we also added in GPU-FOR+GZ, which represents using GPU-FOR (or GPU-DFOR whichever is smaller) followed by gzip. A similar scheme (bit-packing + FOR + gzip) is used by default to encode all integer columns in Apache Parquet [1, 9], a common columnar storage format used in disk-based column stores. We can see from the figure that GPU-FOR achieves most of the compression gains achievable with GPU-FOR+GZ leading to only an additional 14% reduction in storage space compared to GPU-FOR. Gzip works well when there are repetitions in the data. Hence, columns like orderkey benefit from gzip as after applying the frame of reference operation of GPU-FOR; there are repeated strings in the binary data that can be efficiently compressed by gzip.

For the runtime comparison, we compare the performance of Crystal with None and NSF encoding against Crystal with the decompression routines for GPU-FOR. We also compare against OmniSci, a commercial GPU-based OLAP DBMS. Figure 12 shows the runtime comparison of different compression schemes. OmniSci does the worst as it does not use the tile-based execution model and instead operates each thread independently. Crystal-based schemes perform significantly better. Previous work [26] has shown that queries implemented with Crystal achieve a theoretical lower bound for query runtime derived from the fact that memory bandwidth is saturated. Among all the data encoding schemes, NSF achieves the best performance. However, the gap between NSF and GPU-FOR is $<$ 10% for queries q2.1 .. q4.3. This is because these queries have multiple joins and the runtime is dominated by random accesses into hash tables.

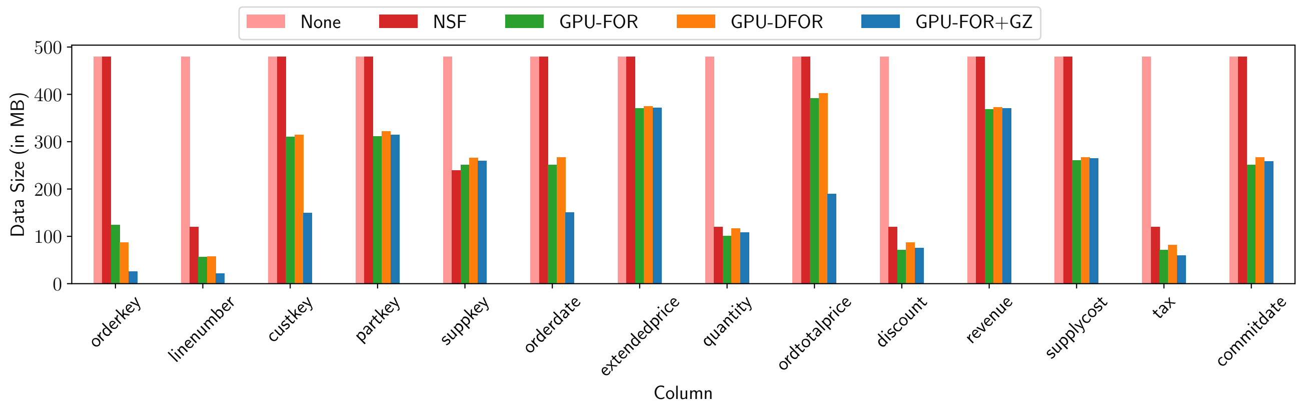


Figure 11: Compression Waterfall for Star Schema Benchmark columns

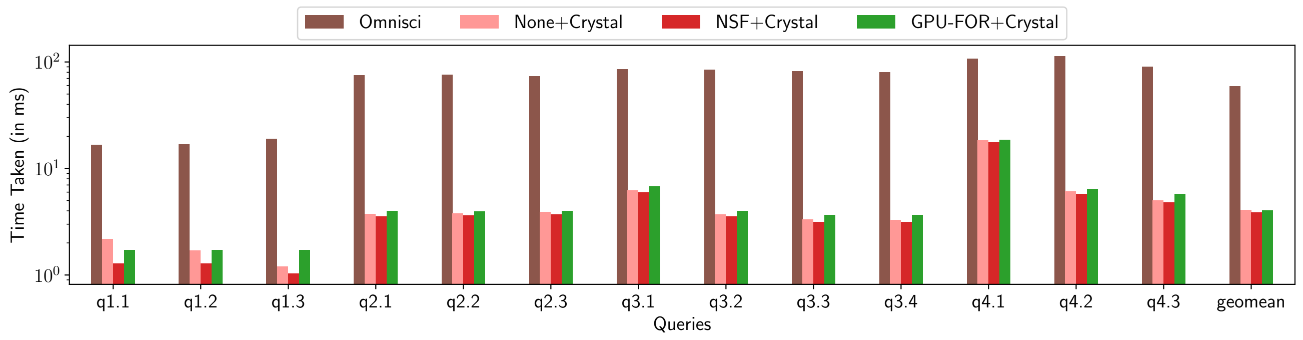


Figure 12: Performance on Star Schema Benchmark Queries

Queries q1.1, q1.2, q1.3 are selection queries and have no joins. These queries follow the same template with q1.3 more selective than q1.2 which is more selective than q1.1. The runtime of GPU-FOR is a constant across queries as it does not support random accesses of entries. Both NSF and None are able to skip cache lines when the query is very selective. For these queries, NSF does best. However, the gap between the NSF and GPU-FOR on an absolute scale is less than 1 ms. Comparing the geometric means of runtime across the entire workload, GPU-FOR is 4% slower than NSF while having 37% smaller storage footprint.

6.5 GPU as a Coprocessor

Many systems use the GPU strictly as a coprocessor [17, 20, 29]. These systems move data from CPU to GPU across an interconnect like PCIe when processing every query. In this setting, the compression schemes discussed in this paper are equally beneficial as the runtime is bound the time taken to ship data over the interconnect (transfer time). GPU-FOR/GPU-DFOR achieve the best compression rate across a variety of data distributions and using them would reduce the amount of data moving across the slow PCIe bus thereby reducing transfer time. To evaluate this, we ran the SSB q1.1 with data stored on the CPU and the columns encoded using GPU-FOR and compared against using NSF and None. The query runtime with None, NSF, and GPU-FOR are 154ms, 96ms, and 63ms respectively. The query runtime is bounded by the time taken for data transfer over PCIe. Query runtime with GPU-FOR is 34% lower than with NSF.

6.6 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: As shown in the SSB benchmark and in the microbenchmarks, the performance of GPU-FOR and GPU-DFOR is competitive with NSF when used as part of a larger kernel. As a result, the rule of thumb is to simply use the compression scheme that has the lowest storage footprint for each column independently. This means using GPU-FOR on all columns except sorted / semi-sorted columns like the primary and secondary sort keys for which GPU-DFOR would generate a more compact representation.
Hyperparameter Tuning: The number of blocks processed per threadblock D is the only hyperparameter in the schemes we propose. We have conservatively choosen $D = 4$ 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: Data compression is a one-time activity that happens on the CPU side. GPU-FOR and GPU-DFOR compression can be done efficiently on the CPU. On a machine where maximum memory bandwidth achievable on a single core is 25GBps, GPU-FOR and GPU-DFOR compression algorithms achieve bandwidth of 16GBps and 14GBps respectively.

7 CONCLUSION

GPU-based analytical systems have demonstrated significant speedups over main memory databases. The key constraint in these systems is the limited GPU memory capacity. This paper presents two efficient massively parallel implementations of bit unpacking routines: GPU-FOR and GPU-DFOR. Together these schemes achieve a 37% reduction in storage footprint on SSB compared to existing schemes with almost no impact on performance. These results show that our algorithms make it practical to use bit-packing for compression on GPU for the first time.

REFERENCES

[1] Apache Parquet. https://parquet.apache.org/.
[2] BlazingDB. https://blazingdb.com.
[3] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[4] GPU-Accelerated Supercomputers Change the Balance of Power on the TOP500. https://bit.ly/2UcBInt.
[5] Kinetica. https://kinetica.com/.
[6] NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/.
[7] OmniSci. https://omnisci.com.
[8] Opencl. https://www.khronos.org/opencl/.
[9] Parquet Encoding Format. https://github.com/apache/parquet-format/blob/master/Encodings.md.
[10] Thrust. https://thrust.github.io/.
[11] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.
[12] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.
[13] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Acm Sigmod Record*, volume 14, pages 268–279. ACM, 1985.
[14] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.
[15] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.
[16] Z. Feng, E. Lo, B. Kao, and W. Xu. 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*, pages 31–46, 2015.
[17] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1603–1618. ACM, 2018.
[18] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998.
[19] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.
[20] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 2013.
[21] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[22] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.
[23] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.
[24] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.
[25] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.
[26] A. Shanbhag, X. Yu, and S. Madden. 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, 2020.
[27] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.
[28] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.
[29] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *PVLDB*, 2013.
[30] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.
[31] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006.