Increasing the Efficiency of GPU Bitmap Index Query Processing

Brandon Tran[1], Brennan Schaffner[1], Jason Sawin[1], Joseph M. Myre[1], and David Chiu[2]

[1] Department of Computer and Information Sciences
University of St. Thomas, St. Paul, MN, USA
jason.sawin@stthomas.edu
[2] Department of Mathematics and Computer Science,
University of Puget Sound, Tacoma, WA, USA

Abstract. Once exotic, computational accelerators are now commonly available in many computing systems. Graphics processing units (GPUs) are perhaps the most frequently encountered computational accelerators. Recent work has shown that GPUs are beneficial when analyzing massive data sets. Specifically related to this study, it has been demonstrated that GPUs can significantly reduce the query processing time of database bitmap index queries. Bitmap indices are typically used for large, read-only data sets and are often compressed using some form of hybrid run-length compression.
In this paper, we present three GPU algorithm enhancement strategies for executing queries of bitmap indices compressed using Word Aligned Hybrid compression: 1) data structure reuse 2) metadata creation with various type alignment and 3) a preallocated memory pool. The data structure reuse greatly reduces the number of costly memory system calls. The use of metadata exploits the immutable nature of bitmaps to pre-calculate and store necessary intermediate processing results. This metadata reduces the number of required query-time processing steps. Preallocating a memory pool can reduce or entirely remove the overhead of memory operations during query processing. Our empirical study showed that performing a combination of these strategies can achieve 33× to 113× speedup over the unenhanced implementation.

Keywords: Bitmap Indices · Big Data · Query Processing · GPU.

1 Introduction

Modern companies rely on big data to drive their business decisions [9, 11, 18]. A prime example of the new corporate reliance on data is Starbucks, which uses big data to determine where to open stores, target customer recommendations, and menu updates [17]. The coffee company even uses weather data to adjust its digital advertisement copy [5]. To meet this need, companies are collecting astounding amounts of data. The shipping company UPS stores over 16 petabytes of data to meet their business needs [9]. Of course, large repositories of data are only useful if they can be analyzed in a timely and efficient manner. In this paper we present techniques that take advantage of synergies between hardware and software to speed up the analysis of data.

Indexing is one of the commonly-used software techniques to aid in the efficient retrieval of data. A bitmap index is a binary matrix that approximates the underlying data. They are regularly used to increase query-processing efficiency in data warehouses and scientific data. It has been shown that bitmap indices are efficient for some of the most common query types: point, range, joins, and aggregate queries. They can also perform better than other indexing schemes like B-trees [26]. One of the main advantages of bitmap indices is that they can be queried using hardware-enabled bitwise operators. Additionally, there is a significant body of work that explores methods of compressing sparse bitmap indices [6, 8, 10, 12, 22, 24]. The focus of most compression work is on various forms of hybrid run-length encoding schemes. These schemes not only achieve substantial compression, but the compressed indices they generate can be queried directly, bypassing the overhead of decompression. One such commonly used compression scheme is Word Aligned Hybrid (WAH) [24]. To improve query processing the WAH scheme compresses data to align with CPU word size.

One of the oft-cited shortcomings of bitmap indices is their static nature. Once a bitmap is compressed, there is no easy method to update or delete tuples in the index. For this reason, bitmap indices are most commonly used for read-only data sets. However, the immutable nature of bitmaps can be exploited to increase the efficiency of query algorithms. Specifically, as bitmap indices are not often updated, it is relatively cheap to build and maintain metadata that can be used to aid in query processing. Additionally, static data structures can be preallocated to reduce query processing overhead.

Meanwhile, recent work has shown how graphics processing units (GPUs) can exploit data-level parallelism inherent in bitmap indices to significantly reduce query processing time. GPUs are massively-parallel computational accelerators that are now standard augmentations to many computing systems. Previously, Andrezejewski and Wrembel [1] proposed GPU-WAH, a system that processes WAH compressed bitmap indices on the GPU. To fully realize the data parallel potential inherent in bitmaps, GPU-WAH must first decompress the bitmap. Nelson *et al.* extended GPU-WAH so that it could process range queries [19]. Nelson *et al.* demonstrated that tailoring the range query algorithm to the unique GPU memory architecture can produce significant improvements (an average speedup of 1.48× over the naive GPU approach and 30.22× over a parallel CPU algorithm).

In this paper, we explore techniques that use metadata, data structure reuse and preallocation tailored to speed up processing WAH range queries on GPUs.

The specific contributions of this paper are:

- We describe how reusing data structures in GPU-WAH decompression algorithm can reduce the number of synchronized memory calls by over 50%.

– We present a tiered investigation of ways to incorporate precompiled metadata into the processing of WAH queries on the GPU. Each successive tier reduces the amount of work performed by the GPU-WAH decompression algorithm but increases the memory overhead. Additionally, we explore how data type selection can align our algorithms to the architecture of the GPU.
– We present a technique that exploits the static nature of bitmap indices to create a fixed size memory pool. The pool is used to avoid all synchronous dynamic memory allocation at query time.
– We present an empirical study of our proposed enhancements to the GPU-WAH decompression algorithm applied to both real and synthetic data sets. Our experimental results show that an implementation using both metadata and a static memory pool can achieve an average speedup of 75.43× over an unenhanced version of GPU-WAH.

The remainder of the paper is organized as follows. In Section 2, we provide an overview of bitmap indices and WAH compression. Section 3 describes a procedure for executing WAH range queries on the GPU. Section 4 describes our enhancement strategies. We present our methodology in Section 5, our results in Section 6 and discuss the results in Section 7. We briefly describe related works a in Section 8. We conclude and present future work in Section 9.

2 Bitmap Indices and WAH Compression

In this section, we describe the creation of bitmap indices and the WAH compression algorithm. A bitmap index is created by discretizing a relation's attribute values into bins that represent distinct values or value-ranges. Table 1 shows a

Table 1. Example relation (left) and a corresponding bitmap (right).

Stocks	
Symbol	**Price**
GE	11.27
WFC	54.46
M	15.32
DIS	151.58
V	184.51
CVX	117.13

Symbol Bins						Price Bins			
s_0	s_1	s_2	s_3	s_4	s_5	p_0	p_1	p_2	p_3
1	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	1	0	0
0	0	1	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	1	0	0	1	0

relation and a corresponding bitmap index. The table above shows a possible bitmap for the **Stocks** relation to its left. The s_i columns in the bitmap are the bins used to represent the **Symbol** attribute. As stock symbols are distinct values, each value is assigned a bin (*e.g.*, s_0 represents the value `GE`, s_1 represents `WFC`, and so on). The p_j bins represent ranges of values into which **Price** values can fall. p_0 represents the range $[0, 50)$, p_1 denotes $[50, 100)$, p_2 is $[100, 150)$, and p_3 represents $[150, \infty)$.

Consider the first tuple in the **Stocks** relation (Table 1). This tuple's **Symbol** value is `GE`, and thus in the bitmap a 1 is placed in s_0 and all other s bins are set to 0. The **Price** value is 11.27. This value falls into the $[0, 50)$ range, so a 1 is assigned to the p_0 bin, and all other p bins get 0.

The binary representation of a bitmap index means that hardware primitive bitwise operations can be used to process queries. For example, consider the following query: `SELECT * FROM Stocks WHERE Price>60;`. This query can be processed by solving $p_2 \vee p_2 \vee p_3 = res$. Only the rows in res that contain a 1 corresponds to a tuple that should be retrieved from disk for further processing.

$$\overbrace{\begin{array}{c}000000000000000001011000000100000000001000111010010000000000101\\000000000000000000000000000000000000000000000000000000000000000\\000000000000000000000000000000000000000000000000000000000000000\\000000000000000000000000000000000000000000000000000000000000000\end{array}}^{Original\ bit\ vector\ in\ 63\ bit\ chunks}$$

(a) Uncompressed bit vector (252 bits)

$$\overbrace{0|000000000000000001011000000100000000001000111010010000000000101}^{64\ bit\ WAH\ literal\ atom}$$

$$\overbrace{1|0|00000000000000000000000000000000000000000000000000000000000011}^{64\ bit\ WAH\ fill\ atom}$$

(b) WAH compressed bit vector

Fig. 1. An example of WAH Compression.

WAH compression operates on stand-alone bitmap bins (which are also referred to as *bit vectors*). An example WAH compression of 252-bits is shown in Figures 1(a) and 1(b). Assuming a 64-bit architecture, WAH clusters a bit vector into consecutive *(system word length)*-1 (or 63) bit "chunks." In Figure 1(a) the first chunk is heterogeneous and the remaining 3 chunks are homogeneous.

WAH then encodes each chunk into system word sized (64-bit) atoms. Heterogeneous chunks are encoded into *literal atoms* of the form (*flag, lit*), where the most-significant-bit (MSB), or *flag*, is set to zero to indicate a *literal*. The remaining 63-bits record the original heterogeneous chunk from the bit vector. The first chunk in Figure 1(a) is heterogeneous and encoded into a *literal* atom.

Homogeneous chunks are encoded as *fill* atoms of the form (*flag, val, len*), where the MSB (*flag*) is set to 1 to indicate a *fill* and the second-MSB (*val*) records the value of the homogeneous sequence of bits. The remaining 62-bits (*len*) record the run length of identical chunks in the original bit vector. The last three chunks in Figure 1(a) are homogeneous and are encoded into a *fill* atom, where the *val* bit is set to 0 and the *len* field is set to 3 (as there are three consecutive repetitions of the homogeneous chunk).

3 GPU Processing of WAH Range Queries

WAH compressed bitmaps can be queried directly without the need for decompression. It has been shown that the system word alignment used by WAH can lead to faster querying than other compression schemes [23]. However, this approach is tailored to the CPU. Previous work [19] has shown that GPUs can process range queries even faster.

Figure 2 illustrates the execution steps used in [19] to process a range query on the GPU. Initially, the compressed bit vectors are stored on the GPU. When the GPU receives a query, the required bit vectors are sent to the **Decompressor**. The decompressed columns are then sent to the **Query Engine** where the query is processed, and the result is sent to the CPU.

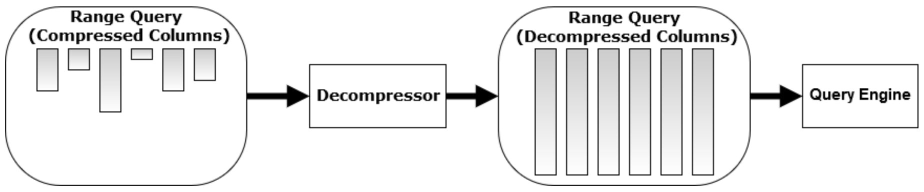


Fig. 2. Main components used to process WAH range queries on a GPU.

Using NVIDIA's compute unified device architecture (CUDA), Nelson et al. [19] presented three parallel reduction-based methods for the query engine: column-oriented access (COA), row-oriented access (ROA), and a hybrid approach. COA performs the reduction on columns, and ROA performs the reduction across single rows. In the hybrid approach, GPU threads are grouped into blocks, and thread blocks are tiled into grids to cover the query data. The blocks then perform a reduction on their data. This approach makes the most efficient use of the GPU memory system. Specifically, it utilizes both the coalesced memory accesses of COA and the use of shared memory for processing along rows of ROA; the hybrid was found to be the fastest method in their experimental study. For the remainder of the paper, we will only be considering the hybrid approach for the query engine though our improvements would benefit all approaches.

The work of this paper focuses on the decompressor component of the above approach. Algorithm 1 presents a procedure for the decompressor unit. It was designed by Andrezejewski and Wrembel [1] and modified in [19] to decompress multiple columns in parallel. The input to the algorithm is a compressed bit vector, *CompData*, the size of the compressed data, *CSize*, and the size of the decompressed data, *DSize*. The output is the corresponding decompressed bit vector, *DecompData*. The algorithm itself comprises five stages; the stages execute sequentially, but the work within stages is processed in parallel.

`Stage 1` (lines 2 - 9) generates an array *DecompSizes* which has the same number of elements as *CompData*. At the end of `Stage 1`, each element in *DecompSizes* will hold the number of words being represented by the atom with the same index in *CompData*. This is accomplished by creating a thread for each atom in *CompData*. If an atom is a literal, its thread assigns 1 to the appropriate index in *DecompSizes* (line 5). If the atom is a fill, the thread assigns the number of words compressed by the atom (line 7).

Algorithm 1 Parallel decompression of compressed data

```
1: procedure DECOMP($Compressed\_BitVector\ CompData$, $CSize$, $DSize$)
2:     ************* STAGE 1 *************
3:     for $i \leftarrow 0$ to $CSize - 1$ in parallel do
4:         if $CompData(i)_{63} = 0b$ then
5:             $DecompSizes[i] \leftarrow 1$
6:         else
7:             $DecompSizes[i] \leftarrow$ the value of $len$ encoded on bits $CompData(i)_{0\to 61}$
8:         end if
9:     end for
10:    ************* STAGE 2 *************
11:    $StartingPoints \leftarrow$ exclusive scan on the array $DecompSizes$
12:    ************* STAGE 3 *************
13:    $EndPoints$ is an array of size $DSize$ filled with zeroes
14:    for $i \leftarrow 1$ to $CSize - 1$ in parallel do
15:        $EndPoints[StartingPoints[i] - 1] \leftarrow 1$
16:    end for
17:    ************* STAGE 4 *************
18:    $WordIndex \leftarrow$ exclusive scan on the array $EndPoints$
19:    ************* STAGE 5 *************
20:    for $i \leftarrow 0$ to $DSize - 1$ in parallel do
21:        $tempWord \leftarrow CompData[WordIndex[i]]$
22:        if $tempWord_{63} = 0b$ then
23:            $DecompData[i] \leftarrow tempWord$
24:        else
25:            if $tempWord_{62} = 0b$ then
26:                $DecompData[i] \leftarrow 0_{64}$
27:            else
28:                $DecompData[i] \leftarrow 0_1 + 1_{63}$
29:            end if
30:        end if
31:    end for
32:    return $DecompData$        ▷ contains a decompressed bit vector of $CompData$
33: end procedure
```

`Stage 2` (line 11) executes an exclusive scan (parallel element summations) on *DecompSizes* storing the results in *StartingPoints* . $StartingPoints[i]$ contains the total number of decompressed words compressed into $CompData[0]$ to $CompData[i-1]$, inclusive. $StartingPoints[i] * 63$ is the number of the bitmap row first represented in $CompData[i]$.

`Stage 3` (lines 13 - 16) creates an array of zeros, *EndPoints*. The length of *EndPoints* equals the number of words in the decompressed data. A 1 is assigned to *EndPoints* at the location of $StartingPoints[i] - 1$ for $i < |StartingPoints|$. In essence, each 1 in *EndPoints* represents where a heterogeneous chunk was found in the decompressed data by the WAH compression algorithm. Note that each element of *StartingPoints* can be processed in parallel.

`Stage 4` (line 18) performs an exclusive scan over $EndPoints$ storing the result in $WordIndex$. $WordIndex[i]$ provides the index to the atom in $CompData$ that contains the information for the ith decompressed word.

`Stage 5` (lines 20 - 31) contains the final for-loop, which represents a parallel processing of every element of $WordIndex$. For each element in $WordIndex$, the associated atom is retrieved from $CompData$, and its type is checked. If $CompData[WordIndex[i]]$ is a WAH literal atom (MSB is a zero), then it is placed directly into $DecompData[i]$. Otherwise, $CompData[WordIndex[i]]$ must be a fill atom. If it is a fill of zeroes (second MSB is a zero), then 64 zeroes are assigned into $DecompData[i]$. If it is a fill of ones, a word consisting of 1 zero (to account for the flag bit) and 63 ones is assigned to $DecompData[i]$. The resulting $DecompData$ is the fully decompressed bitmap.

4 Memory Use Strategies

We explored memory-focused strategies to accelerate GPU query processing: 1) data structure reuse, 2) metadata storage, and 3) employing a preallocated memory pool. Descriptions of each strategy are provided below.

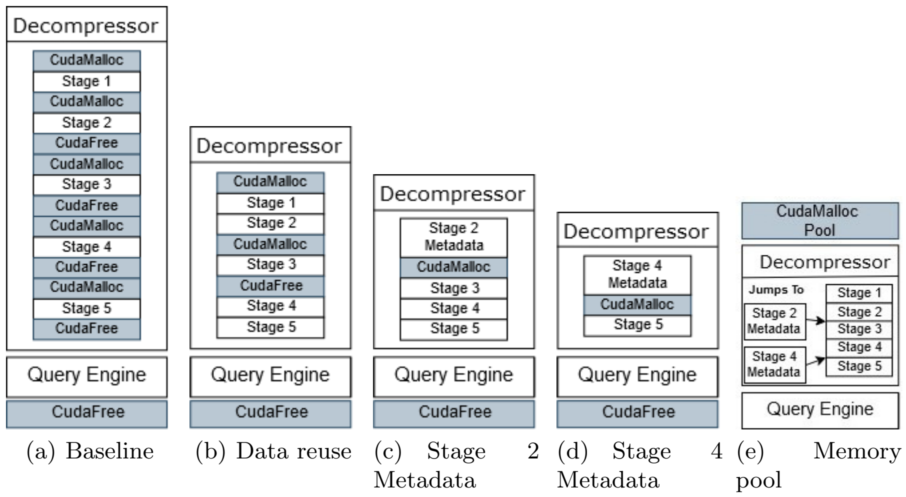


Fig. 3. Various implementations of a GPU-WAH system specialized for range queries.

Figure 3(a) depicts the steps required for our baseline implementation of Algorithm 1. As shown, this implementation requires five `cudaMalloc()` calls and four `cudaFree()` calls in the decompressor and an additional `cudaFree()` after the query engine has finished. Each `cudaMalloc()` is allocating an array needed in the following algorithmic stage. The CUDA library only supports synchronous memory allocation and deallocation. Synchronous memory operations combined with data dependencies in Algorithm 1 make memory operations a limiting factor for decompression.

Data structure reuse- We can reduce the number of CUDA memory calls by reusing data structures. The arrays created in `Stage 1` and `Stage 2` of Algorithm 1, *DecompSizes* and *StartingPoints*, are both the length of the compressed data. By performing an in-place exclusive scan on *DecompSizes*, meaning the results of the scan are saved back to *DecompSizes*, we no longer need to create *StartingPoints*. Similarly, we can perform an in-place scan on *EndPoints* in `Stage 4`. Moreover, we can reuse *EndPoints* for *DecompData*. After the data is read from *EndPoints* (line 21), the results of the writes in line 26 and line 28 can be written back to *EndPoints* without loss of data. Figure 3(b) shows the steps of implementation with data structure reuse. As shown, it only requires two calls to `cudaMalloc()`, one before `Stage 1` and another before `Stage 2`. It requires a call to `cudaFree()` after `Stage 3` is finished with *DecompSizes* and the final `cudaFree()` after the query engine has finished with the decompressed data saved in *EndPoints*. By careful reuse of data structures, we reduce the number of CUDA memory calls from 10 to 4.

Storing Metadata- Further memory management and even some processing stages can be skipped by pre-generating intermediate results of the decompression algorithm (Algorithm 1) and storing them as metadata. For example, the only information from `Stage 1` and `Stage 2` used in the remainder of the algorithm is stored in *StartingPoints*. By generating *StartingPoints* prior to query-time and storing the results as `Stage 2 metadata` both `Stage 1` and 2 of Algorithm 1 can be skipped. Figure 3(c) depicts an extension of our *data structures reuse* system enhanced with `Stage 2 metadata`. At query-time, the metadata is stored statically in memory on the GPU, so there is no need to allocate memory for *StartingPoints*. By injecting the stored information, the decompression algorithm can be started at `Stage 3`. As shown, this approach still requires a call to `cudaMalloc()` to create the array that will eventually hold the decompressed bit vector. That memory will need to be freed after the query has been processed.

Using a metadata approach, it is possible to skip all but the final stage of the decompression algorithm. The only information that flows from `Stage 4` to `Stage 5` is stored in $WordIndex$ which can be pre-computed and stored. Figure 3(d) shows a system that uses `Stage 4 metadata`. Notice that it skips `Stages 1-4`. However, it still requires a memory allocation for `Stage 5` as the data structure reuse system saved the final decompressed data in the original $WordIndex$ array. Now $WordIndex$ is stored as metadata and overwriting it would slow the performance of subsequent queries as they would no longer have access to stored information. The `cudaMalloc()` call in 3(d) is allocating memory for a structure that will hold the fully decompressed data. This memory will need to be freed after the query is completed.

Any speedup realized by our metadata approaches is achieved at the cost of a larger memory footprint. To reduce the space requirements of our implementation, we explore the effects of using 32-bit and 64-bit integer types to store `Stage 2` and `Stage 4 metadata`. Our version of the decompression algorithm expected the WAH compression to be aligned with a 64-bit CPU system word size. However, `Stage 2 metadata` contains the total number of decompressed words compressed from *CompData*[0] to *CompData*[$i-1$], for some non-zero index i. The largest possible element is equal to the number of system words comprising the decompressed bit vector. Hence, for decompressed bitmaps containing less than $(2^{32}-1)\times 64$ rows `Stage 2 metadata` can be a 32-bit datatype. Essentially, this type-size reduction would make the `Stage 2 metadata` half the size of the compressed bitmap.

For each decompressed word w in a bit vector, `Stage 4 metadata` stores an index into *CompData* where w is represented in compressed format. In essence, `Stage 4 metadata` maps decompressed words to their compressed representations. As long as the compressed bit vector does not contain more than $(2^{32}-1)$ atoms, a 32-bit data type can be used for `Stage 4 metadata`. This type reduction makes `Stage 4 metadata` half the size of the decompressed data. Note that storing `Stage 4 metadata` using 64-bit integer types would require the same memory footprint as the fully decompressed bitmap. In this case, it would be advantageous to store just the decompress bitmap and circumvent the entire decompression routine.

Memory Pool- A common approach to avoid the overhead of `cudaMalloc()` and `cudaFree()` is to create a preallocated static memory pool (e.g., [14,21,25]). We create a memory pool tailored to the bitmap that is stored on the GPU. A hashing function maps thread-ids to positions in preallocated arrays. The arrays are sized to accommodate a decompressed bit vector of the bitmap stored on the GPU. Threads lock their portion of the array during processing. The array is released back to the pool at the end of query processing. All available GPU memory that is not being used to store the bitmap and metadata is dedicated to the memory pool. This design will lead to a query failure if the memory requirements are too large. This limitation motivates future work that will explore methods for distributing massive indices across multiple GPUs.

Figure 3(e) shows the design of our fully enhanced GPU-WAH range query system. The use of a memory pool removes the need to invoke CUDA memory calls. As shown, the memory pool can be used in conjunction with both of our metadata strategies to circumvent stages of the decompression algorithm. It can also be used as a standalone solution.

5 Experiments

In this section, we describe the configuration of our testing environment and the process that was used to generate our results. All testing was executed on a machine running Ubuntu 16.04.5 LTS, equipped with dual 8-core Intel Xeon E5-2609 v4 CPUs (each at 1.70 GHz) and 322 GB of RAM. The CPU side of the system was written in C++ and compiled with GCC v5.4.0. The GPU components were developed using CUDA v9.0.176 and run on an NVIDIA GeForce GTX 1080 with 8 GB of memory.

We used the following data sets for evaluation. They are representative of the type of read-only applications (*e.g.*, scientific) that benefit from bitmap indexing.

- `BPA` – contains measurements reported from 20 synchrophasors (measures magnitude and phase of AC waveform) deployed by Bonneville Power Administration over the Pacific Northwest power grid [4]. Data from each synchrophasors was collected over approximately one month. The data arrived at a rate of 60 measurements per second and was discretized into 1367 bins. We use a $7,273,800$ row subset of the measured data.
- `linkage` – contains anonymous records from the Epidemiological Cancer Registry regarding the German state of North Rhine-Westphalia [20]. The data set contains $5,749,132$ rows and 12 attributes. The 12 attributes were discretized into 130 bins.
- `kddcup` – contains data obtained from the 1999 Knowledge Discovery and Data Mining competition. These data describe network flow traffic. The set contains $4,898,431$ rows and 42 attributes [15]. Continuous attributes were discretized into 25 bins using Lloyd's Algorithm [16], resulting in 475 bins.
- `Zipf` – contains data generated using a Zipf distribution. This is the only synthetic data set on which we tested. A Zipf distribution represents a clustered approach to discretization, which can capture the skew of dense data in a bitmap. With the Zipf distribution generator, the probability of each bit being assigned to 1 is: $P(k, n, skew) = (1/k^{skew})/\sum_{i=1}^{n}(1/i^{skew})$ where n is the number of bins determined by cardinality, k is their rank (bin number: 1 to n), and the parameter $skew$ characterizes the exponential skew of the distribution. Increasing $skew$ increases the likelihood of assigning 1s to bins with lower rank (lower values of k) and decreases the likelihood of assigning 1s to bins with higher rank. We set $n = 10$ and $skew = 2$ for 10 attributes, which generated a data set containing 100 bins (*i.e.,* ten attributes discretized into ten bins each) and 32 million rows. This is the same synthetic data set used in Nelson et al. [19].

We tested multiple configurations of additional enhancement strategies for query execution. These configurations are comprised of three classes of options:

1. Data structure reuse
2. Metadata: None, `32-bit Stage 2`, `64-bit Stage 2`, `32-bit Stage 4`, and fully decompressed columns.
3. Memory pool usage: used or unused.

We tested all valid combinations of these options on each of the four data sets. Due to the mutually exclusive nature of the metadata storage options, this results in 10 augmented configurations plus the baseline approach.

All tests used range queries of sizes 64 columns. To obtain representative execution times for each query configuration we repeated each test 6 times. The execution time of the first test is discarded to remove transient effects, and the arithmetic mean of the remaining 5 execution times is recorded. We used the average to calculate our performance comparison metric, $speedup = t_{base}/t$, where t_{base} is the execution time of the baseline for comparison and t is the execution time of the test of interest. The baseline we used for all speedup calculations was the implementation of the decompression algorithm from [19] (a slightly modified version of the algorithm presented in [1]).

6 Results

Here we present the results obtained from the experiments described in the previous section. We first discuss the impact of memory requirements. We then present results for data structure reuse, metadata, data type size, and memory pool strategies that were described in Section 4.

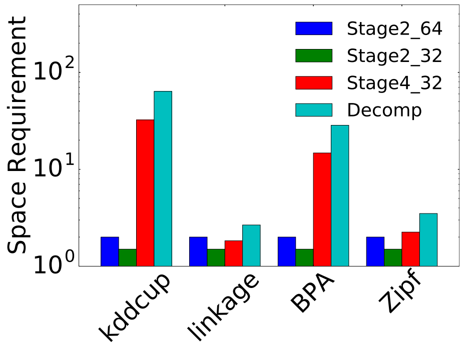


Fig. 4. Metadata memory space requirements relative to the baseline approach storing only compressed bitmaps. Note, the vertical axis is logarithmic.

The performance provided by some of the techniques in this paper comes at the cost of additional memory costs, which are shown in Figure 4. Relative to standard storage requirements, the storage requirements when using 32-bit `Stage 2 metadata`, 64-bit `Stage 2 metadata`, and 32-bit `Stage 4 metadata`, are 1.5×, 2×, and an average of 12.8×, respectively.

The speedup provided by reuse of data structures to eliminate memory operations is shown in Figure 5. Eliminating the overhead of many memory operations enhanced performance by a maximum of 8.53× and 5.43×, on average. The kddcup and BPA data sets exhibited greater speedup than the linkage and Zipf datasets due to the relative compressibility of these data sets and its effect on the decompression routine.

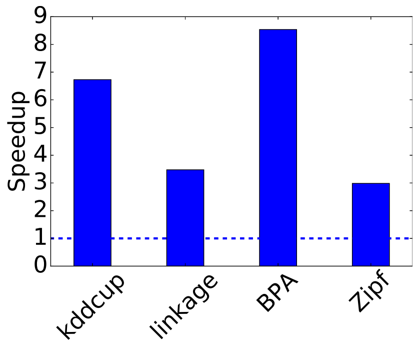
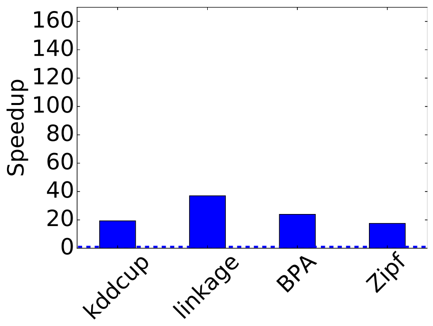


Fig. 5. Speedup provided by data structure reuse.

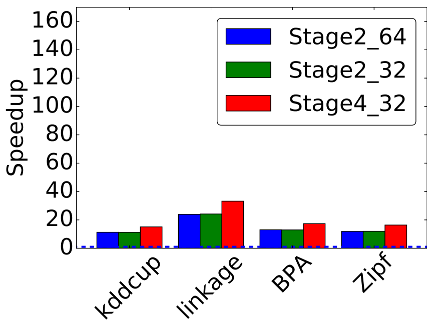
Performance enhancement provided by the use of a memory pool is shown in Figure 6(a). This enhancement consistently provided an average of 24.4× speedup across all databases and a maximum speedup of 37.0×.

Incorporating metadata also provided consistent results as can been seen in Figure 6(b). Using `Stage 2` metadata provided an average of 15.1× speedup. `Stage 4` metadata is more beneficial with an average of 20.5× speedup.

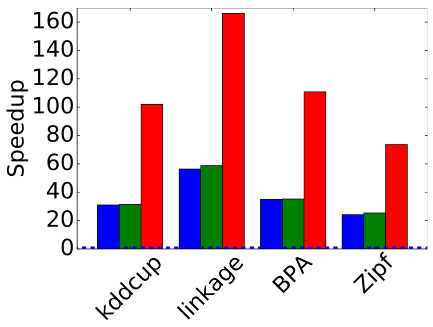
Varying data type size yielded negligible performance enhancement. When a memory pool was not used, as shown in Figure 6(b), there was no observable performance difference between 32-bit and 64-bit data types. On average, their separation was less than 0.234× speedup. When a memory pool was used, as shown in Figure 6(c), there was a performance boost when using 32-bit data types with an average improvement of 9.24× over 64-bit types.



(a) Memory pool



(b) Metadata strategies and data type sizes



(c) Metadata strategies, data type sizes, and a memory pool

Fig. 6. Shown are performance results for a) memory pool usage, b) different metadata strategies and data type sizes, and c) different metadata strategies, data type sizes, and a memory pool. The dashed horizontal line indicates a speedup of 1. Figures b) and c) share a legend.

Using a combination of metadata, data type size, and memory pool techniques produced the greatest performance benefit, as seen in Figure 6(c). Across all databases, using `Stage 2 metadata` and a memory pool provided an average 37.7× speedup and a maximum of 58.8× speedup. Using `Stage 4 metadata` and a memory pool provided an average 113× speedup and a maximum of 166× speedup.

7 Discussion of Results

The performance provided by data reuse is dependant on the compressibility of the data set. Data sets with greater compressibility exhibit stronger performance relative to those with less compressibility. This is because data sets with less compressibility incur more global memory accesses on the GPU.

Storing the results of the first exclusive scan as 32-bit metadata instead of 64-bit not only saved storage space but also provided faster execution times (23.4% faster, on average). On NVIDIA GPUs, 32-bit integer operations are faster than 64-bit because the integer ALUs are natively 32-bits. 64-bit operations are performed using combinations of 32-bit operations.

When combining metadata and memory pool strategies, the attained speedup was greater than the sum of the speedup of each individual strategy. When only using metadata, the final stage can not begin until the necessary memory is allocated. When only using a memory pool, the final stage can not begin until the subsequent stage is completed. Combining the methods removes both bottlenecks and allows `Stage 5` to execute almost immediately.

Although it has the highest storage cost, using fully decompressed columns as metadata reduces execution time because the decompression routine is completely avoided. Figure 7 shows the performance enhancement provided by using fully decompressed columns as "metadata". This option is only reasonable for small databases or GPUs with large storage space. This strategy provided a maximum of 699× speedup and an average of 411× speedup.

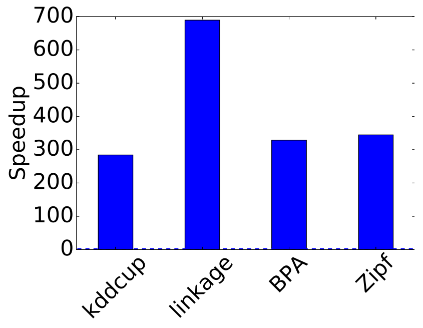


Fig. 7. Speedup provided by using decompressed bit vectors as "metadata".

Figure 8 shows execution profiles when using (a) data structure reuse, (b) 32-bit `Stage 2 metadata` without a memory pool, (c) 32-bit `Stage 4 metadata` without a memory pool, (d) only a memory pool, and 32-bit `Stage 2` and 32-bit `Stage 4 metadata` with a memory pool in (e) and (f), respectively.

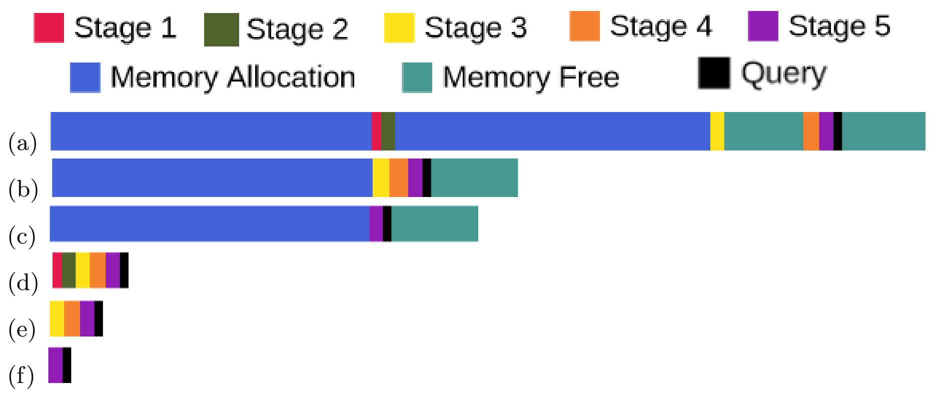


Fig. 8. A representative (albeit approximate) view of execution profiles of six identical query executions on the *linkage* database with varying enhancement strategy configurations. Query execution progresses from left to right. Longer bars correspond to longer execution times. While execution profiles for other databases exhibited slight variations, their interpretations remained consistent.

Data structure reuse (shown in Figure 8(a)) eliminated three of five allocation/free pairs providing an average of 5.43× speedup. Profiles using `Stage 2` and `Stage 4` metadata are shown in Figures 8(b) and 8(c), respectively. Both provide a noticeable reduction in execution time as each eliminate a memory allocation and free pair. The major cost of memory operations remains a dominant factor so the difference between `Stage 2` and `Stage 4` metadata use is limited.

The profile when using only a memory pool is shown in Figure 8(d). The memory pool removes the overhead of memory operations providing a greater reduction in execution time than pure metadata strategies. Strategies combining a memory pool with `Stage 2` or `Stage 4` metadata are shown in Figures 8(e) and 8(f), respectively. These combination strategies provide the benefits of both strategies: short-circuiting to a mid-point of the decompression routine and removing the overhead of GPU memory operations.

8 Related Work

Our work focuses on efficient GPU decompression and querying of WAH compressed bitmaps. There are many other hybrid run-length compression schemes designed specifically for bitmap indices. One of the earliest was Byte-aligned Bitmap Compression (BBC) [3]. The smaller alignment can achieve better compression but at the expense of query speed [23]. Other compression schemes have employed variable alignment length [8, 13]. These approaches try to balance the trade-offs between compressing shorter runs and increasing query processing time. Others use word alignment but embed metadata in fill atoms that improve compression or query speed [7, 10, 12, 22]. These techniques were developed for execution on the CPU, though they could be ported to the GPU by altering the decompressor component of the GPU system described above. We believe that many, if not all, of these compression techniques on the GPU would benefit from a variation of our metadata and memory pool enhancements.

To the best of our knowledge, we are first to use metadata to efficiently decompress WAH bitmap indices on the GPU. However, other work has explored the benefits of memory pools on GPU in a variety of applications. For example, Hou et al. [14] used a specialized memory pool to create kd-trees in the GPU. Their approach allowed them to process larger scenes on the GPU than previous work. Wang et al. [21] used a preallocated memory pool to reduce the overhead of large tensor allocations/deallocations. Their approach produced speedups of 1.12× to 1.77× over the use of `cudaMalloc()` and `cudaFree()`. The work of Simin et al. [25] is similar to our work in that they use a memory pool to increase the query processing of R-trees on GPU's. We were unable to find any work that used a GPU memory pool specifically designed for use with bitmap indices.

As mentioned above, our work extends the works of Andrzejewski and Wrembel [1, 2] and Nelson et al. [19]. Andrzejewski and Wrembel introduced WAH and PLWAH [10] compression and decompression algorithms for GPUs as well as techniques to apply bitwise operations to pairs of bit vectors. Their decompression algorithm details a parallel approach for a decompressing a single WAH or PLWAH compressed bit vector. Nelson et al. modified Andrzejewski and Wrembel's decompression algorithm to apply it to multiple bit vectors in parallel. They then presented multiple algorithms for executing bitmap range queries on the GPU. Our experimental study used their most efficient range query implementation. As the work in this paper improves the efficiency of WAH bitmap decompression on the GPU, it represents a significant enhancement to approaches presented by Andrzejewski and Wrembel's and Nelson et al.

9 Conclusion and Future Works

In this paper, we present multiple techniques for accelerating WAH range queries on GPUs: data structure reuse, storing metadata, and incorporating a memory pool. These methods focus on reducing memory operations or removing repeated decompression work. These techniques take advantage of the static nature of bitmap indexing schemes and the inherent parallelism of range queries.

We conducted an empirical study comparing these acceleration strategies to a baseline GPU implementation. The results of our study showed that the data reuse, metadata, and memory pool strategies provided average speedups of 5.4×, 17.8×, and 24.4×, respectively. Combining these techniques provided an average of 75.4× speedup. We also found that storing the entire bitmaps as accessible metadata on the GPU resulted in an average speedup of 411× by eliminating the need for decompression altogether. This option is only feasible for configurations with small databases or GPUs with large storage space.

In future work, comparing energy consumption of the above approaches may prove interesting. We would also like to investigate executing WAH queries using multiple GPUs. Using multiple GPUs would provide additional parallelism and storage capabilities. Furthermore, since WAH compression is designed for CPU style processing, future studies could investigate new compression schemes that are potentially better fit for the GPU architecture.

References

1. Andrzejewski, W., Wrembel, R.: GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In: International Conference on Database and Expert Systems Applications. pp. 315–329. Springer (2010)
2. Andrzejewski, W., Wrembel, R.: GPU-PLWAH: GPU-based implementation of the PLWAH algorithm for compressing bitmaps. Control and cybernetics **40**, 627–650 (2011)
3. Antoshenkov, G.: Byte-aligned bitmap compression. In: Proceedings DCC'95 Data Compression Conference. p. 476. IEEE (1995)
4. Bonneville power administration, http://www.bpa.gov
5. Bradlow, E., Gangwar, M., Kopalle, P., Voleti, S.: The role of big data and predictive analytics in retailing. Journal of Retailing **93**, 79–95 (March 2017)
6. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps. Softw. Pract. Exper. **46**(5), 709–719 (May 2016)
7. Colantonio, A., Di Pietro, R.: Concise: Compressed 'n' composable integer set. Information Processing Letters **110**(16), 644–650 (2010)
8. Corrales, F., Chiu, D., Sawin, J.: Variable length compression for bitmap indices. In: Database and Expert Systems Applications. pp. 381–395 (2011)
9. Davenport, T., Dyche, J.: Big data in big companies. Tech. rep., International Institute for Analytics (2013)

10. Deliège, F., Pedersen, T.B.: Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps. In: International Conference on Extending Database Technology. pp. 228–239. EDBT '10 (2010)
11. Erevelles, S., Fukawa, N., Swaynea, L.: Big data consumer analytics and the transformation of marketing. Journal of Business Research **69**, 897–904 (February 2016)
12. Fusco, F., Stoecklin, M.P., Vlachos, M.: Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. VLDB **3**(2), 1382–1393 (2010)
13. Guzun, G., Canahuate, G., Chiu, D., Sawin, J.: A tunable compression framework for bitmap indices. In: 2014 IEEE 30th International Conference on Data Engineering. pp. 484–495. IEEE (2014)
14. Hou, Q., Sun, X., Zhou, K., Lauterbach, C., Manocha, D.: Memory-scalable gpu spatial hierarchy construction. IEEE Transactions on Visualization and Computer Graphics **17**(4), 466–474 (April 2011)
15. Lichman, M.: UCI machine learning repository (2013), http://archive.ics.uci.edu/ml
16. Lloyd, S.: Least squares quantization in pcm. IEEE transactions on information theory **28**(2), 129–137 (1982)
17. Marr, B.: Starbucks: Using big data, analytics and artificial intelligence to boost performance. Forbes (May 2018), https://www.forbes.com/sites/bernardmarr/2018/05/28/starbucks-using-big-data-analytics-and-artificial-intelligence-to-boost-performance/#5784902e65cd
18. McAfee, A., Brynjolfsson, E.: Big data: The management revolution. Harvard Business Review pp. 61–68 (October 2012)
19. Nelson, M., Sorenson, Z., Myre, J., Sawin, J., Chiu, D.: GPU Acceleration of Range Queries over Large Data Sets. In: Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Application, and Technologies (BDCAT'19). pp. 11–20 (2019)
20. Sariyar, M., Borg, A., Pommerening, K.: Controlling false match rates in record linkage using extreme value theory. Journal of Biomedical Informatics **44**(4), 648–654 (2011)
21. Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S.L., Xu, Z., Kraska, T.: Superneurons: Dynamic gpu memory management for training deep neural networks. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 41–53 (2018)
22. Wu, K., Otoo, E.J., Shoshani, A., Nordberg, H.: Notes on design and implementation of compressed bit vectors. Tech. Rep. LBNL/PUB-3161, Lawrence Berkeley National Laboratory (2001)
23. Wu, K., Otoo, E.J., Shoshani, A.: Compressing bitmap indexes for faster search operations. In: Proceedings 14th International Conference on Scientific and Statistical Database Management. pp. 99–108. IEEE (2002)
24. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. **31**(1), 1–38 (2006)
25. You, S., Zhang, J., Gruenwald, L.: Parallel spatial query processing on gpus using r-trees. In: Proceedings of the 2Nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data. pp. 23–31 (2013)
26. Zaker, M., Phon-Amnuaisuk, S., Haw, S.C.: An adequate design for large data warehouse systems: Bitmap index versus b-tree index. International Journal of Computers and Communications **2**, 39–46 (2008)