Data Science and Engineering (2021) 6:209–228
https://doi.org/10.1007/s41019-020-00148-8

Exploring Means to Enhance the Efficiency of GPU Bitmap Index Query Processing

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

Received: 9 June 2020 / Revised: 17 October 2020 / Accepted: 10 November 2020 / Published online: 30 November 2020

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 32.4× to 98.7× speedup over the current state-of-the-art implementation. Our study also showed that by using our enhancements, a common gaming GPU can achieve a 15.0× speedup over a more expensive high-end CPU.

Keywords Bitmap indices · Big data · Query processing · GPU

Abbreviations

CPU	Central processing unit
CUDA	Compute unified device architecture
GPU	Graphics processing unit
MSB	Most significant bit
WAH	Word-aligned hybrid
COA	Column oriented access
ROA	Row oriented access
ALU	Arithmetic logic unit
BBC	Byte-aligned bitmap compression
PLWAH	Position list word-aligned hybrid
FAST	Fast architecture sensitive tree
GB	Gigabyte
EWAH	Enhanced word-aligned hybrid
RAM	Random access memory
KDD	Knowledge discovery and data mining
BPA	Bonneville power administration

✉ Jason Sawin
jason.sawin@stthomas.edu

[1] Department of Computer and Information Sciences, University of St. Thomas, 2115 Summit Ave, Mail Number OSS 402, St. Paul, MN 55105, USA

[2] Department of Mathematics and Computer Science, University of Puget Sound, Tacoma, WA, USA

1 Introduction

Modern companies rely on big data to drive their business decisions [14, 16, 31]. 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 [30]. The coffee company even uses weather data to adjust its digital advertisement copy [6]. 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 [14]. 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 [49]. One of the main advantages of bitmap indices is that they can be queried using fast bitwise operations. Additionally, there is a significant body of work that explores methods of compressing sparse bitmap indices [7, 12, 15, 17, 46, 47]. 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 commonly used compression scheme is word aligned hybrid (WAH) [46]. To improve query processing, WAH 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 rarely 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.

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 [33, 34]. 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 baseline GPU approach and 30.22× over a parallel CPU algorithm). Currently, Nelson et al. presents the only other work using GPU's to process WAH bitmap range queries.

In this paper, we explore techniques that use metadata, data structure reuse, and preallocation tailored to speed up the processing of WAH range queries on GPUs. The major contributions of this paper are:

- We present a novel tiered restructuring of the current state-of-art WAH decompression algorithm for GPUs. Our algorithm uses pre-compiled metadata to circumvent stages of the decompression process. Each tier represents a memory/time trade-off which allows for a tailored application of our algorithm.
- We present several memory strategies that exploit the static nature of bitmaps. These include recycling data structures and using a pre-allocated memory pool. We also demonstrate how data-type selection aligns our algorithms to the GPU architecture.
- We present a novel reduction-based method for processing WAH range queries in parallel on a CPU.
- 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 our enhancement strategies provide an average and a maximum speedups of 75.43× and 98.7×, respectively, over the current state-of-the-art of GPU-WAH range query processing algorithms.
- We compare the querying performance of our proposed enhancements to GPU-WAH to our parallel CPU WAH implementation. Our experimental results show that with our enhancements on relatively inexpensive GPUs are able to achieve an average 3.07× and a maximum of 15.0× speedup over a high-end CPU.

The remainder of the paper is organized as follows. In Sect. 2, we provide an overview of bitmap indices and WAH compression. Section 3 provides a high-level overview of GPU architecture. Section 4 describes procedures for executing WAH range queries on the CPU and GPU. Section 5 describes our enhancement strategies. We present our methodology in Sect. 6, our results in Sect. 7, and discuss the results in Sect. 8. We briefly describe related works in Sect. 9. We conclude and present future work in Sect. 10.

2 Bitmap Indices and WAH Compression

In this section, we describe how bitmap indices are created. We also present how WAH can compress such bitmap indices and the algorithms for querying WAH compressed bitmap indices.

2.1 Bitmap reation

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 relation and a corresponding bitmap index. The right most table 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, ∞).

After the bins have been established, each tuple in the relation is processed. For example, 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. This binning process is performed on all the tuples in **Stocks** to create the shown bitmap shown to the right of the relation.

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 $p_1 \vee p_2 \vee p_3 = res$. Only the rows in *res* that contain a 1 corresponds to a tuple that should be retrieved from disk for further processing.

2.2 WAH Compression and Querying

One of the predominate compression algorithms for bitmap indices is word aligned hybrid (WAH). WAH compression operates on stand-alone bitmap bins (also referred to as *bit vectors*). Figure 1a presents an example bit vector consisting of 252-bits (shown in chunks). Figure 1b shows that same bit vector compressed using WAH.

Assuming a 64-bit architecture, WAH clusters a bit vector into consecutive *(system word length)*−1 (or 63-) bit "chunks." In Fig. 1a the first chunk is heterogeneous and the remaining 3 chunks are homogeneous. Each chunk is then encoded into system word sized (64-bit) atoms. Heterogeneous chunks are encoded as *literal atoms* of the form (*flag, lit*). The most-significant-bit (MSB), or *flag*, is zero to indicate a *literal*. The remaining 63-bits (*lit*) record the original heterogeneous chunk from the bit vector. Since the first chunk is heterogeneous it is 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 Fig. 1a 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).

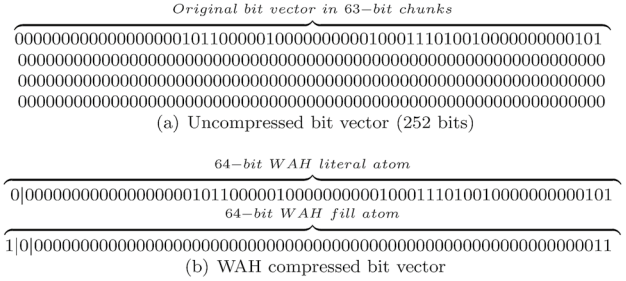


Fig. 1 An example bit vector represented with WAH compression

Table 1 Example relation (stocks) and a corresponding bitmap (symbol and price bins)

Stocks		Symbol bins						Price bins			
Symbol	Price	s_0	s_1	s_2	s_3	s_4	s_5	p_0	p_1	p_2	p_3
GE	11.27	1	0	0	0	0	0	1	0	0	0
WFC	54.46	0	1	0	0	0	0	0	1	0	0
M	15.32	0	0	1	0	0	0	1	0	0	0
DIS	151.58	0	0	0	1	0	0	0	0	0	1
V	184.51	0	0	0	0	1	0	0	0	0	1
CVX	117.13	0	0	0	0	0	1	0	0	1	0

Algorithm 1 WAH point query [46]: ∘ is a logical bit wise operator

```
1:  procedure POINT QUERY(Compressed_BitVector A,Compressed_BitVector B)
2:      Z ← empty bit vector
3:      currentAtom_A ← A.popAtom()
4:      currentAtom_B ← B.popAtom()
5:      while A and B are not empty do
6:          if currentAtom_A.isExhausted() then
7:              currentAtom_A ← A.popAtom()
8:          end if
9:          if currentAtom_B.isExhausted() then
10:             currentAtom_B ← B.popAtom()
11:         end if
12:         if currentAtom_A.isFill and currentAtom_B.isFill then
13:             nLen ← min(currentAtom_A.len, currentAtom_B.len)
14:             nVal ← currentAtom_A.val ∘ currentAtom_B.val
15:             Z.pushNewFillAtom(nval, nLen)
16:             currentAtom_A.reduce(nLen)
17:             currentAtom_B.reduce(nLen)
18:         else
19:             nLit ← currentAtom_A.getLitVal() ∘ currentAtom_B.getLitVal()
20:             Z.pushNewLitAtom(nLit)
21:             currentAtom_A.reduce(1)
22:             currentAtom_B.reduce(1)
23:         end if
24:     end while
25:     return Z                          ▷ Contains compressed result of A ∘ B
26: end procedure
```

Algorithm 1 [46] shows how two bit vectors compressed using WAH can be queried directly without the need for decompression. As shown, the algorithm takes two WAH compressed bit vectors, *A* and *B*, as input. The ∘ operator at lines 14 and 19 represents any hardware supported bitwise logical operator (e.g., *AND*, *OR*, *XOR*, etc.) The procedure returns *Z* a compressed bit vector that contains the result of $A \circ B$. In the algorithm, *A* and *B* are treated as stacks of WAH atoms. It first pops an atom off each "stack" (lines 3 and 4). The algorithm continues to process atoms as long as both stacks are not empty (line 5). During each iteration, it first checks if the current operand atoms have been fully processed, or exhausted. If the atom has been exhausted, the next atom from the appropriate bit vector is fetched (lines 6-11).

At line 12, `Point Query` determines the type of atom pairing is currently being processed. If both atoms are fills, a new resulting fill atom is pushed onto the result stack. The *val* bit of the new fill atom is the result of ∘ being applied to the two *val* bits of the operand atoms. The *len* of the resulting atom is *nLen*, which is the minimum value of the two *len* values of the operand atoms. The *reduce* operation shown in lines 16 and 17, is a bookkeeping procedure that records that *nLen* words have been processed in both the operand atoms. Note that in the `Fill/Fill`, pairing the atom with the minimum *len* value will be fully exhausted and will require replacing during the next iteration. If one of the operand atoms is a literal (lines 18-22), a new literal atom is added to *Z*. The *lit* value of the new atom is the result of applying ∘ to the *lit* values of the two operand atoms. If one of the atoms is a fill, the *getLitVal*() method returns a single literal atom representation of the fill (i.e., either 63 zeros or 63 ones). Then, each operand atom is reduced by 1 to indicate a single word has been processed. This reduction will exhaust any literal atoms. After all the atoms have been processed, *Z* is returned.

3 Graphics Processing Units

As shown above, a significant advantage of WAH is that compressed bitmaps can be queried directly without being decompressed first. It has been shown that the use of system word alignment by WAH can provide enhanced querying performance over other compression schemes [45]. This approach is well-suited for CPU implementations. While querying compressed columns is possible using graphics processing units (GPUs), the parallelism present in GPUs is unlikely to be fully utilized. This is due to low data alignment and a significant amount of branching instructions. Unlike CPUs, GPUs do not have branch predictors and branch instructions can induce a phenomenon called

warp divergence (described later in this section). It has been shown that GPUs can efficiently query decompressed bit vectors as this produces a high degree of data level parallelism and reduces the amount of branch instructions encountered [1, 2, 33].

With NVIDIA's compute unified device architecture (CUDA) programming platform for GPUs, tens of thousands of threads can be organized into 1-, 2-, or 3-dimensional Cartesian structures. Cartesian layouts naturally maps to many computational problems. With CUDA, these structures hierarchically comprise thread grids, thread blocks, and threads. Threads are executed in groups (conventionally known as cooperative thread arrays or warps) of 32, *ergo*, thread blocks are typically composed of $32m$ threads, where m is a positive integer.

The memory hierarchy for CUDA capable NVIDIA GPUs is closely linked to the organization of threads. The CUDA memory hierarchy is composed of global, shared, and local memory. Global memory is accessible to all threads. Thread blocks have private access to their own low-latency shared memory ($\sim 100\times$ less than global memory latency) [13]. Each thread also has access to its own private local memory.

For a CUDA capable NVIDIA GPU to fully realize high-bandwidth global memory transfers, it is critical to *coalesce* global-memory accesses. Coalesced global memory accesses occur when two criteria are fulfilled: 1) the accessed memory addresses are sequential and 2) the accessed memory addresses span the addresses $32n$ to $32n + 31$, for some integer, n. Coalescing global memory accesses allow the GPU to batch memory transactions in order to reduce the total number of memory transfers.

The NVIDIA CUDA GPU functional units for integer arithmetic are 32-bits at their core [35]. This incurs a performance penalty when performing 64-bit integer arithmetic as 64-bit integer arithmetic is emulated using 32-bit types. Modern 64-bit CPUs do not suffer this penalty.

A core challenge to ensuring high computational throughput on CUDA capable GPUs is warp divergence. The phenomenon of warp divergence occurs when threads within the same warp resolve a branching instruction (commonly resulting from loops or if-else statements) differently. At an architectural level, CUDA GPUs require all threads within a warp to execute the same sequence of instructions. When warp divergence occurs, a CUDA GPU will execute the multiple instruction sequences present in the warp serially. This serial execution of thread subsets within a warp can significantly reduce performance as computational throughput is reduced. For example, two possible branch outcomes exist when an if-else statement is encountered. Warp divergence occurs when a warp fails to evaluate the condition uniformly. Some subset of the warp will execute the true path and the complementary subset will execute the false path. The execution of these two branch outcomes occurs serially. The total execution time required for the serial execution of the two outcomes is then $t_T + t_F + t_\epsilon$, where t_T is the execution time of the true path, t_F is the execution time of the false path, and t_ϵ is the overhead necessary to orchestrate the serial execution. To accomplish the serial execution necessary to resolve warp divergence, the CUDA runtime environment "deactivates" subsets of the threads within the divergent warp. The active subset of threads executes instructions from a single path. Once all paths are executed, the threads within the warp continue executing uniformly.

4 Architectural Approaches to WAH Range Query Processing

4.1 CPU Processing of WAH Range Queries

We implement a parallel approach for WAH range query processing using multi-core CPUs. The majority of multi-core CPUs do not exhibit the same degree of parallelism that

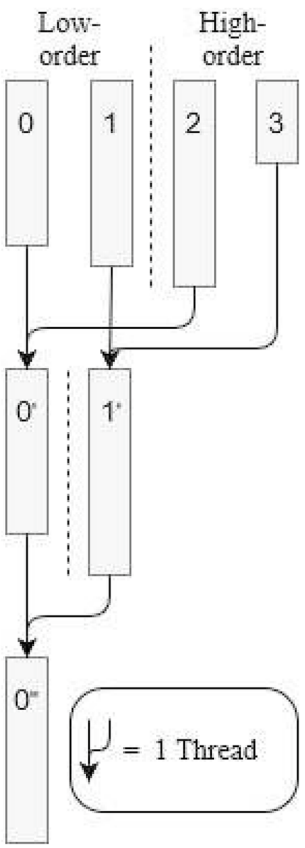


Fig. 2 Processing a range query using a reduction

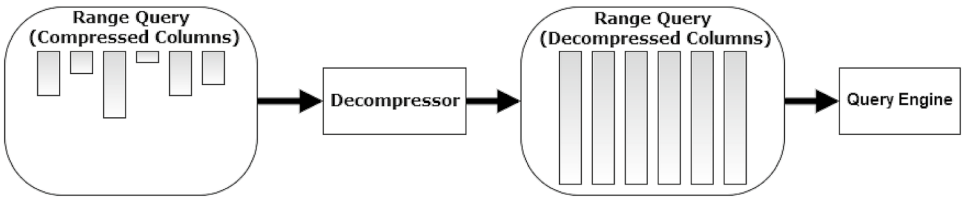


Fig. 3 Main components used to process WAH range queries on a GPU

is present in GPUs. As such, our CPU approach is restricted to a parallel reduction. This approach executes up to p parallel point queries (using OpenMP on a p-core CPU) on paired compressed bit vectors for any given reduction level. If more than p bit vector pairs exist in a given reduction level, the CPU must iterate through the remaining pairs until all of the bit vectors for that level have been processed. Once the final reduction level (consisting of a single bit vector pair) has been processed, the range query result is obtained.

Figure 2 shows this process for a range query of four bit vectors. Each thread reads from two operand bit vectors and writes to one result bit vector. Executing a range query using a reduction requires $log_2(n)$ reduction levels, where n is the number of bit vectors being processed. This is due to the consecutive halving nature of the reduction operation. The query shown in Fig. 2 requires two reduction levels. In the first level, two independent pairs of bit vectors can be processed in parallel. The final level consists of processing a single pair of bit vectors to produce the result of the query.

4.2 GPU Processing of WAH Range Queries

Figure 3 illustrates the execution steps used in [33] 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 in parallel, and the result is sent to the CPU. Without decompression, the WAH query process has very low data level parallelism and the query engine would not be able to take advantage the the parallelism inherent in the GPU.

Algorithm 2 Ideal hybrid query processing

```
1: procedure IDEALHYBRID(Cols,q)
2:     ▷ Cols is a collection of decompressed bit vectors
3:     ▷ q is the number of tiles in the y-dimension
4:     m ← |Cols|                          ▷ the number of bit vectors in the query
5:     n ← |Cols_0|                        ▷ the number of words in a bit vector
6:     for rb ← 0 to n/q − 1 in parallel do
7:         for t ← rb ∗ n/q to (rb + 1) ∗ n/q − 1 in parallel do
8:             s ← m/2
9:             while s ≥ 1 do
10:                for c ← 0 to s − 1 in parallel do
11:                    c1 ← Cols_c
12:                    c2 ← Cols_{c+s}
13:                    c1_t ← c1_t ∨ c2_t
14:                end for
15:                s ← s/2
16:            end while
17:        end for
18:    end for
19:    return Cols_0
20: end procedure
```

Using NVIDIA's CUDA, Nelson et al. [33, 34] presented four parallel reduc-tion-based methods for the query engine: column-oriented access (COA), row-oriented access (ROA), and a (standard and ideal) hybrid approach. COA performs the reduction on columns, and ROA performs the reduction across single rows. In the hybrid approaches, GPU threads are grouped into blocks, and thread blocks are tiled into grids to cover the query data. The blocks then perform two rounds of

reduction on their data. The ideal hybrid approach is formed when queries are sufficiently small to complete the query in a single round of reduction.

The ideal hybrid 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. The ideal hybrid approach is shown in Algorithm 2 [33] and is applicable to queries of up to 1024 bit vectors. Thread blocks are tiled to maximize the parallel processing of input bit vectors. Each tile, which represents one thread block, has a width that spans all columns and an adjustable length (inversely proportional to the width) so that each thread block can use the maximum of 1024 threads. Every thread block is partitioned and executed in parallel where each thread (also in parallel) performs $log_2(b)$ levels of reduction, where b is the number of bit vectors spanned by the thread block, storing the result in the lower order column. The final result of the range query, which resides in the first column, is returned. For the remainder of the paper, we will only be considering the ideal hybrid approach for the query engine though our improvements would benefit all approaches.

Algorithm 3 Parallel decompression of compressed data

```
1: procedure DECOMP(CompData,CSize,DSize)
2:     ▷ CompData: WAH compressed bit vector
3:     ▷ CSize: Size of CompData in atoms
4:     ▷ DSize: Size of decompressed data in words
5:     ********** STAGE 1 **********
6:     MemoryAllocate(DecompSizes)                          ▷ an array of size CSize
7:     for i ← 0 to CSize − 1 in parallel do
8:         if CompData(i)_63 = 0b then
9:             DecompSizes[i] ← 1
10:        else
11:            DecompSizes[i] ← the value of len encoded on bits CompData(i)_0→61
12:        end if
13:    end for
14:    ********** STAGE 2 **********
15:    MemoryAllocate(StartingPoints)                       ▷ an array of size CSize
16:    StartingPoints ← exclusive scan on the array DecompSizes
17:    MemoryFree(DecompSizes)
18:    ********** STAGE 3 **********
19:    MemoryAllocate(EndPoints)          ▷ an array of size DSize filled with zeroes
20:    for i ← 1 to CSize − 1 in parallel do
21:        EndPoints[StartingPoints[i] − 1] ← 1
22:    end for
23:    MemoryFree(StartingPoints)
24:    ********** STAGE 4 **********
25:    MemoryAllocate(WordIndex)                            ▷ an array of size DSize
26:    WordIndex ← exclusive scan on the array EndPoints
27:    MemoryFree(EndPoints)
28:    ********** STAGE 5 **********
29:    MemoryAllocate(DecompData)                           ▷ an array of size DSize
30:    for i ← 0 to DSize − 1 in parallel do
31:        tempWord ← CompData[WordIndex[i]]
32:        if tempWord_63 = 0b then
33:            DecompData[i] ← tempWord
34:        else
35:            if tempWord_62 = 0b then
36:                DecompData[i] ← 0_64
37:            else
38:                DecompData[i] ← 0_1 + 1_63
39:            end if
40:        end if
41:    end for
42:    MemoryFree(WordIndex)
43:    return DecompData          ▷ contains a decompressed bit vector of CompData
44: end procedure
```

The work of this paper focuses on the decompressor component of the above approach. Algorithm 3 presents a procedure for the decompressor unit. It was designed by Andrezejewski and Wrembel [1] and modified in [33] 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 5–13) 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 9). If the atom is a fill, the thread assigns the number of words compressed by the atom (line 11).

`Stage 2` (lines 14–17) executes an exclusive scan (parallel element summations) on *DecompSizes* storing the results in *StartingPoints*. The element, *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 18–22) 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* is processed in parallel.

`Stage 4` (lines 24–27) 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 28–42) 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.

4.3 Cost Analysis of GPU WAH Range Queries

We analytically found the operation count for all of *Decomp*'s stages (algorithm 3) using n for the number of WAH atoms in the compressed column, m for the number of system words in a decompressed column, and w for the WAH atom length in bits. The work factor given for each stage regards apparent work, not total work. In other words, we are concerned with the work that cannot be hidden by parallelism. For example, consider a loop that executes 50,000 iterations. Due to the serial nature of the loop, each operation in its body would need to be counted 50,000 times. However, if each iteration of the same loop could be executed concurrently using 50,000 threads the entirety of the loop could be completed in the time of one iteration. Our analysis treats multiple simultaneous (parallel) executions of an operation as a single event. We considers both arithmetic operations and memory operations. Arithmetic operations include additions, subtractions, conditional checks, etc., and memory operations include assignments, array accesses, allocations, and frees.

The first operation of `Stage 1` is the allocation (via `cudaMalloc()`) of $n \times w$ bits of memory for the array *DecompSizes*. All iterations of the following loop are executed in parallel, as are all other remaining loops shown in the algorithm. Getting the 63-rd bit of *CompData* and checking its value at line 8 takes one memory operation and one arithmetic operation. For *Decomp*, the worst-case configuration for a compressed column of n atoms is for all of them to be fills. This configuration would lead to the conditional statement only executing line 11. This path through `Stage 1` is the most costly, as there are two memory operations to decode the value of *len* and assign it to *DecompSizes*. Therefore, in the worst-case, `Stage 1` takes a total of 1 arithmetic operation and 4 memory operations, including an allocation of size $n \times w$.

`Stage 2` (lines 14 to 17) first allocates $n \times w$ bits of memory for the array *StartingPoints*. This new array stores the result of an exclusive scan of *DecompSizes* at line 16. *DecompSizes* has the same length as the compressed column, n; with the parallelism attained by the GPUs, the exclusive scan has a total memory and arithmetic operation count of $O(log_2(n))$, each. Then, since *DecompSizes* is no longer needed, `cudaFree()` is called to free its $n \times w$ bits of memory.

In `Stage 3`, a memory allocation of $m \times w$ bits is required for *Endpoints*. Note that this requires a call to both `cudaMalloc()` and `cudaMemset()` to initialize the array to zeroes. There is also a single arithmetic operation for subtraction and 2 memory operations to cover the *EndPoints* array access and assignment. Following the final use of *StartingPoints*, its $n \times w$ bits of memory is freed.

`Stage 4` (lines 24 to 27) requires a memory allocation of $m \times w$ bits for *WordIndex*, which holds the result of the exclusive scan on *EndPoints*. Similar to `Stage 2`, the parallel exclusive scan has a work factor of $O(log_2(m))$ memory and arithmetic operations each. Then, the $m \times w$ bits of memory allocated for *EndPoints* is freed.

`Stage 5` first allocates $m \times w$ bits of memory to store the resulting decompressed column. Then line 31 requires 2 memory operations to access and store an atom from the input column. For the remainder of the stage, the worst-case occurs when both conditionals (line 32 and 35) fail. This happens when the atom is a fill of 1's. The conditional checks execute a memory access and a boolean evaluation each, followed by an addition and assignment. Ending this stage, a `cudaFree()` releases the $w \times m$ bits allocated for *WordIndex*. In total, `Stage 5` uses 7 memory and 3 arithmetic operations. After the query is made, a final `cudaFree()` is called on the decompressed column memory.

The worst-case scenario for *Decomp* (Algorithm 3) is dominated by memory operations. These encompass 5 memory allocations and frees, $10 + O(log_2(n)) + O(log_2(m))$ other memory operations which include global reads and writes, and $5 + O(log_2(n)) + O(log_2(m))$ arithmetic operations. There is a significant body of work that has shown that conventional memory allocation is inefficient on CUDA GPUs [19, 25, 38, 44]. The inefficiency of these operations is compounded because they act as a blocking operation for all threads within a CUDA stream, a hindrance to parallelism. The next biggest expense comes from the exclusive scans. Despite our use of a high-performance implementation [32], the exclusive scans still incur numerous accesses to global memory, which each take hundreds of cycles [35]. Our implementation strategies detailed in Sect. 5 focus on reducing the need for the costly memory operations. Reusing data structures removes the need for calls to `cudaMalloc()` (and respective `cudaFree()` function calls) for *StartingPoints*, *WordIndex*, and *DecompData* which reduces the number of memory allocations and frees to 2 each. Storing pre-calculated metadata eliminates the need for additional `cudaMalloc()`s, `cudaFree()`s, and exclusive scans. Utilizing a memory pool avoids the final `cudaFree()`. Overall, the memory operation count is reduced to a single `cudaMalloc()` plus $O(1)$ assignments and accesses.

It is important to note that the operation counts are idealized. In reality, the parallelism is reduced significantly by warp divergence when threads in the same warp take different execution paths because of conditional variation. The parallelism is further reduced by architectural limits such as limited number of threads and registers. There is also limited memory throughput due to concurrent memory access and memory bus bandwidth.

4.4 Potential Sources and Impacts of Warp Divergence

As described in Sect. 3, warp divergence can have a detrimental effect on execution time. There are six points in Algorithm 3 that might induce warp divergence. The loop on line 7 is the first such instance. Because we align the threads in this loop to the sequential chunks of the bit vectors, only the warp covering the end of `CompData` may experience divergence. This is because the total number of threads used are a multiple of 32 and the length of `CompData` may not be. This behavior is identical for subsequent loops on lines 20 and 30 regarding the warps of threads that cover the bit vectors `StartingPoints` and `WordIndex`, respectively. The conditional expressions on lines 8, 32, and 35 pose a different possibility: any warp could experience divergence. Nested branches (like the if-else statements in Algorithm 3 on lines 8, 32, and 35) can be additionally problematic, as already divergent warps can experience further divergence. This imposes additional thread serialization and overhead. In total, the three loops in Algorithm 3 may cause warp divergence in one warp out of those spanning their respective bit vectors, and the three conditional statements may cause warp divergence in any number (zero to all) of the executing warps.

The worst case scenario for warp divergence occurs when a warp must execute all possible branch outcomes serially. Potential divergent pathways due to branching instructions in `Stages` 1, 3, and 5 are shown in Fig. 4. When the worst case scenario for warp divergence is encountered `Stage` 1, 3, or 5, the total branch outcomes that must be executed serially are three, two, and four, respectively. Again, each of these divergent branch outcomes incur some additional overhead by the CUDA runtime to resolve the warp divergence.

5 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.

Figure 5a depicts the steps required for our baseline implementation of Algorithm 3. 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 management routines (allocations and frees). Synchronous memory operations combined with data dependencies in Algorithm 3 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 3, *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*. We use a similar in-place scan on *EndPoints* in `Stage 4`. Moreover, we can reuse *EndPoints* for *DecompData*. After the data are read from *EndPoints* (line 31), the results of the writes in line 36 and line 38 can be written back to *EndPoints* without loss of data. Figure 5b 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 3`. 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 3) 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 3 can be skipped. Figure 5c depicts an extension of our *data structures reuse* system enhanced with `Stage 2` metadata. At start up 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. Starting the algorithm at `Stage 3` has the additional benefit of eliminating 2 points of potential warp divergence (see Sect. 4.4).

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 5d shows a system that uses `Stage 4` metadata. In addition to skipping `Stages 1-4`, this method provides the added benefit of eliminating additional (total of 3) points of potential warp divergence (see Sect. 4.4). However, this method 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 Fig. 5d 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 are 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 data type. 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

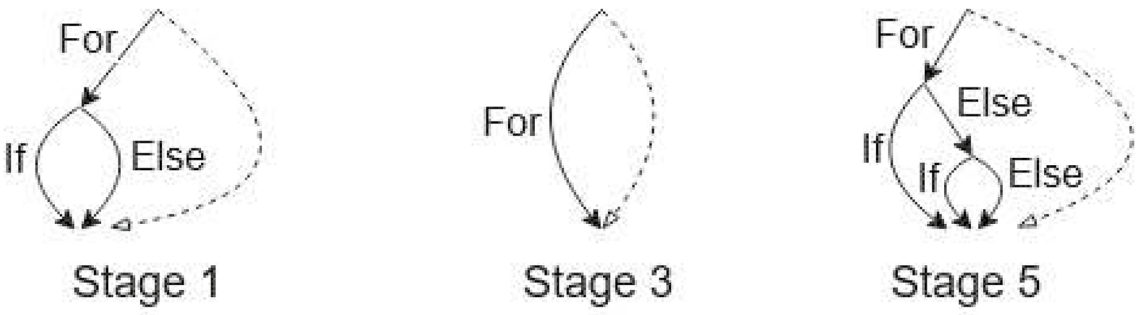


Fig. 4 Possible divergent pathways due to branching instructions (conditionals and loops) in stages 1, 3 and 5. The dotted lines represent the branch outcome that does not execute the loop contents

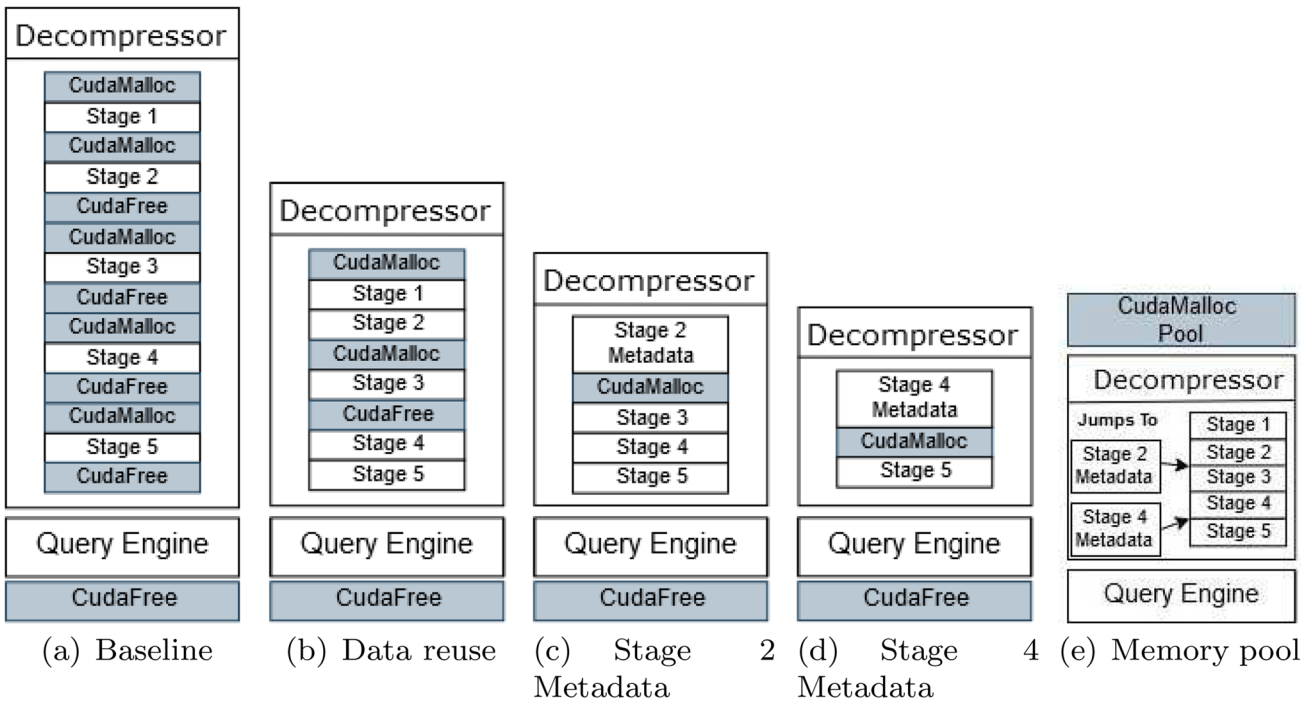


Fig. 5 Various implementations of a GPU-WAH system specialized for range queries

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 decompressed bitmap and circumvent the entire decompression routine.

Memory Pool A common approach to avoid the overhead of cuda memory operations (e.g., `cudaMalloc()` and `cudaFree()`) is to create a preallocated static memory pool (e.g., [24, 42, 48]). 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. Threads lock their portion of the array during processing and release them only after all of the query have been processed. 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.

Figure 5e 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 enhancement strategy.

Appropriately sizing the memory pool can be done analytically. We used the cumulative distribution function for a Poisson distribution to determine the effectiveness of any given memory pool size at handling queries with different amounts of bit vectors. This approach is illustrated in Fig. 6. The shaded region in Fig. 6 shows the quantity of overall queries (the sizes of which are described by the Poisson distribution) that can be handled by the memory pool. From the Poisson distribution analysis, the memory pool size can be calculated as $s\lceil \mu + k\sigma \rceil$ where s is the number of bytes in a decompressed column, μ is the mean number of bit vectors in a query, σ is the standard deviation of the distribution, and $k \in \mathbb{R}^+$. Setting the memory pool size using $k = 2$ would effectively capture ~96.82% of queries. Calculating a memory pool size using $k = 3$ would be capable of fielding ~99.45% of queries.

6 Experiments

In this section, we describe the configuration of our testing environment and the process that was used to generate our results. All tests were executed on two different machines. System A is 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. System B is a machine running Ubuntu 19.10 LTS equipped with a 16-core hyperthreading (32 total concurrent threads) Intel Xeon Gold 6130 CPU (at 2.10 GHz) and 64 GB of RAM. The CPU side of the system was compiled with GCC v8.4.0. The GPU components were compiled with CUDA v10.1.243 and run on a Quadro RTX 4000 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` – measurements reported from 20 synchrophasors (measures magnitude and phase of AC waveform) deployed by Bonneville power administration over the Pacific Northwest power grid [5]. Data from each synchrophasors were collected over approximately one month. The data arrived at a rate of 60 measurements per second and was discretized into 1367 bins. We used 7, 273, 800 row subset of the measured data.
- `linkage`—anonymous records from the Epidemiological Cancer Registry regarding the German state of North Rhine-Westphalia [37]. The data set contains 5, 749, 132 rows and 12 attributes. The 12 attributes were discretized into 130 bins.
- `kddcup` – 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 [28]. Continuous attributes were discretized into 25 bins using Lloyd's Algorithm [29], resulting in 475 bins.
- `Zipf`—data generated using a Zipf distribution. These are the only synthetic data sets 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 $skew = 0, 1$, and 2 for 10 attributes, which generated three data sets containing 100 bins (*i.e.,* ten attributes discretized into ten bins each) and 32 million rows.

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

1. Data structure reuse (unused only in baseline).
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 resulted in 10 augmented configurations plus the baseline approach (shown in Fig. 5a).

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 was discarded to remove transient effects, and the arithmetic mean of the remaining 5 execution times was recorded. We used the average to calculate our performance

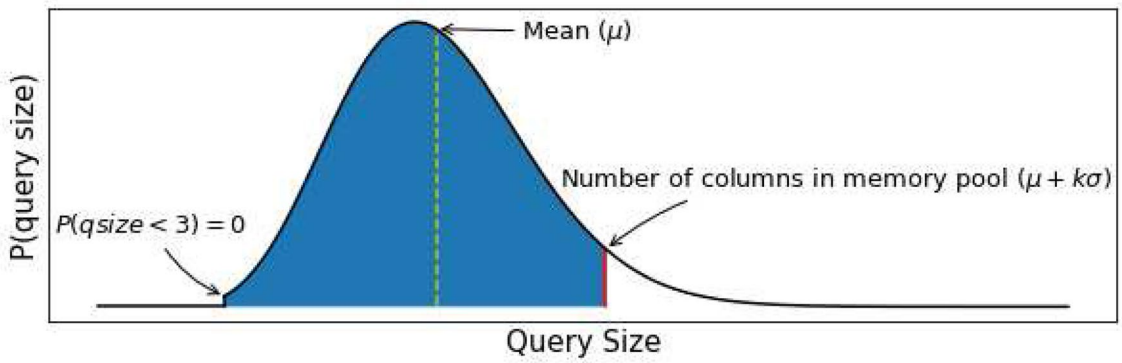


Fig. 6 An example Poisson distribution that could be considered when setting the size of the memory pool. The shaded region indicates the means for creating a memory pool that can store queries of $\mu + k\sigma$ bit vectors, where μ is the mean number of bit vectors in a query, σ is the standard deviation of the distribution, and $k \in \mathbb{R}^+$. The horizontal axis represents the range of possible query sizes (in number of columns), and the vertical axis shows the probability of receiving a query of any given size

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 decompression algorithm and best performing implementation of GPU-WAH from [33].

7 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 Sect. 5.

The performance provided by some of the techniques in this paper comes at the cost of additional memory. Figure 7 illustrates this as a cost relative to the standard approach of storing compressed bit vectors. Relative to standard storage requirements, the average increase in 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 9.04×, respectively.

The speedup provided by reuse of data structures to eliminate memory operations is presented in Fig. 8. As shown, significant performance improvements are realized due to the elimination of 3 memory allocations and 3 memory frees. For system A, this reduction provided a maximum speedup of 23.7× and an average speedup of 13.9×, with the linkage data set exhibiting greater speedup than others. For system B, data structure reuse provided a maximum of 5.14× speedup and an average of 3.64× speedup. Again, the linkage data set exhibited the greatest speedup.

The results of our other enhancement strategies on System A are shown in Fig. 9. Performance improvements provided by the use of a memory pool are shown in Fig. 9a. This enhancement consistently provided an average speedup of 22.1× across all databases and a maximum speedup of 37.0×. Incorporating metadata also provided consistent results as can been seen in Fig. 9b. Using `Stage 2` metadata provided an average of 14.1× speedup. `Stage 4` metadata was more beneficial with an average of 19.3× speedup. Varying data type size yielded negligible performance improvements. When a memory pool was not used, as shown in Fig. 9b, there was no observable performance difference between 32-bit and 64-bit data types. On average, the use of the 32-bit data type size was 1.003×. When a memory pool was used, as shown in Fig. 9c, there was still negligible improvement when using 32-bit data types with an average improvement of 1.04× over 64-bit types. Using a combination of metadata, data type size, and memory pool techniques produced the greatest performance benefit, as seen in Fig. 9c. Across all databases, using `Stage 2 metadata` and a memory pool provided a maximum speedup of 58.8× and an average speedup 33.6×. Using `Stage 4` metadata with a memory pool provided a maximum of 166× speedup and an average 98.7× speedup.

The results of the same enhancement strategies on System B are shown in Fig. 10. Improvements from using a memory pool are shown in Fig. 10a with a maximum 5.64× speedup and an average 3.67× speedup. Performance enhancement when using the metadata and varying data type size strategies on System B is shown in Fig. 10b. We see a maximum improvement of 9.70× and an average performance enhancement of 3.83×. The difference between 32-bit and 64-bit data types only appears when using Stage 2 metadata and was negligible. Using 32-bit only provided an average of 1.003× speedup over 64-bit when not using a memory pool. When including the memory pool, varying data type size remained negligible, with 32-bit types providing an average 1.02× speedup. The performance enhancement provided by combining all strategies is shown in Fig. 10c. Combining the memory pool and metadata storage methods provided the most performance enhancement. For all data sets, using `Stage 2` metadata and a memory pool provided a maximum of 7.65× speedup and an average of 4.75× speedup. Using `Stage 4` metadata with the memory pool, we achieved a maximum 21.1× speedup and and average 12.6× overall.

8 Discussion of Results

The performance provided by data reuse is dependent 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.

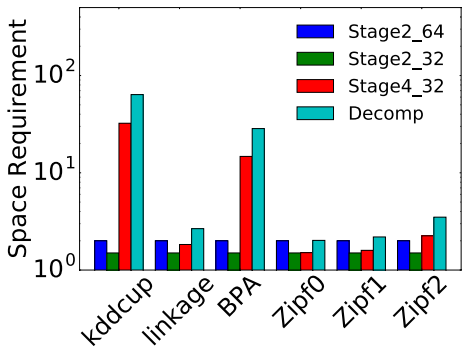


Fig. 7 Metadata memory space requirements relative to the baseline approach storing only compressed bitmaps. Note, the vertical axis is logarithmic

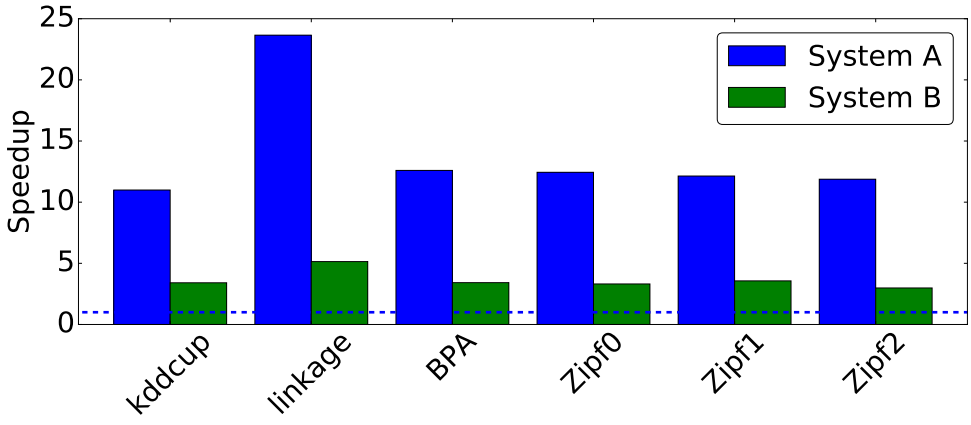
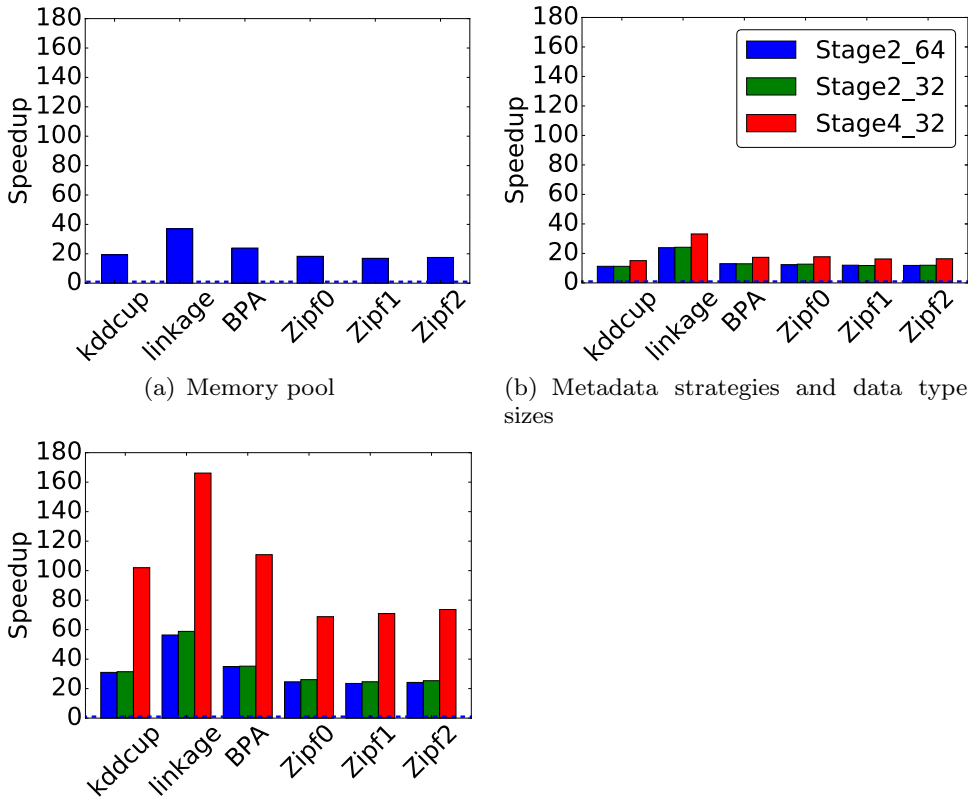


Fig. 8 Speedup provided by data structure reuse. The dashed horizontal line indicates a speedup of 1



(a) Memory pool

(b) Metadata strategies and data type sizes

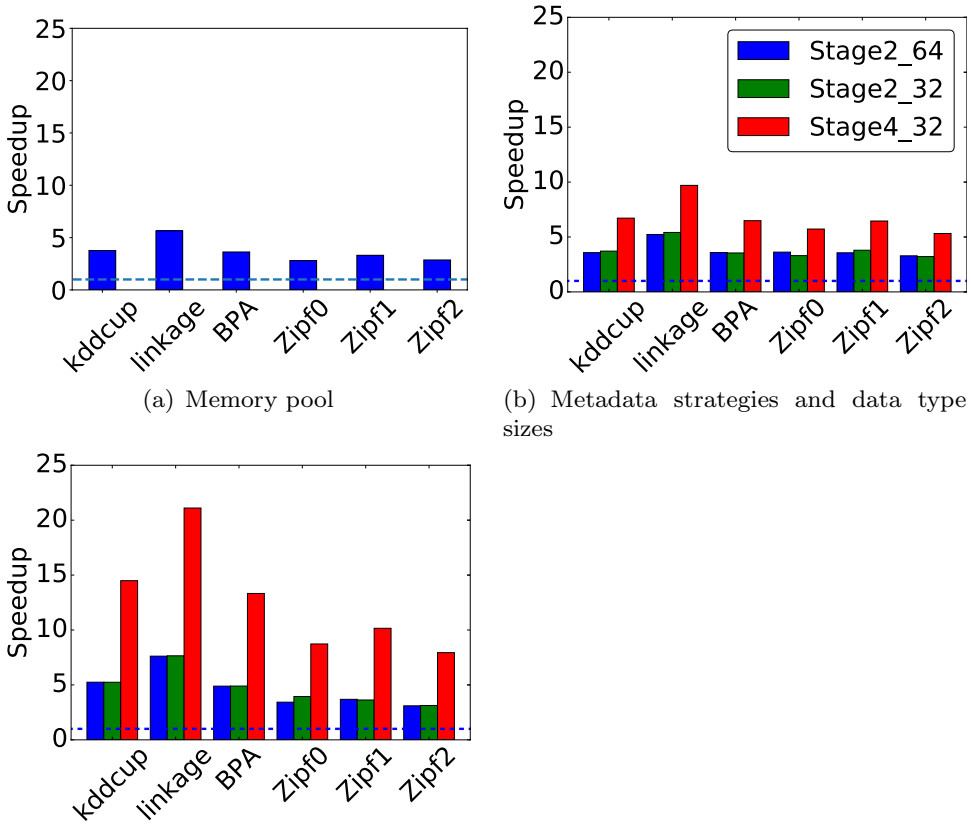
(c) Metadata strategies, data type sizes, and a memory pool

Fig. 9 Shown are System A 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

Storing the results of the first exclusive scan as 32-bit metadata instead of 64-bit not only reduced storage requirements but also provided slightly faster execution times (0.35% 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 emulated as sequences of 32-bit operations.

By 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 Algorithm 3 to start at the stage using the metadata as input (`Stage 3` or `Stage 5`).

The results for the linkage database from both systems are compared in Fig. 11. In Fig. 11a, we see the execution times for both systems for all enhancement strategies. System B is on average 4.75 ms slower than than System A. A



(a) Memory pool

(b) Metadata strategies and data type sizes

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

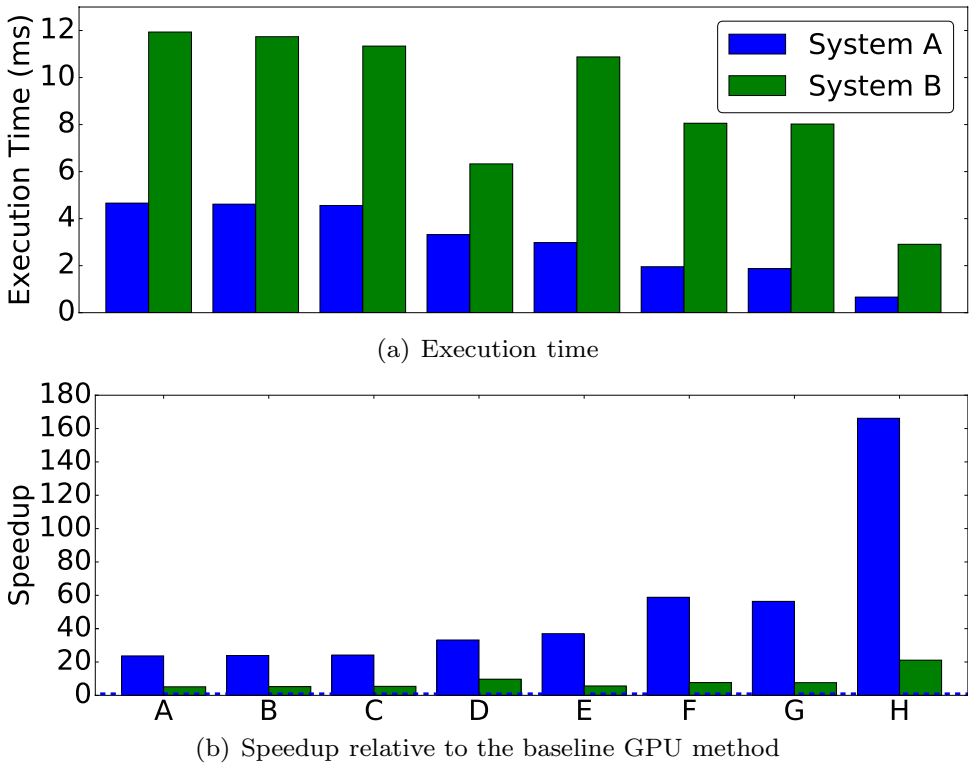
Fig. 10 Shown are System B performance results for **a** memory pool usage, **b** different metadata strategies and data type sizes, and **c** different metadata strategies, data type sizes, and a memory pool. The dashed horizontal line indicates a speedup of 1. Figures **b** and **c** share a legend

comparison of GPU speedups for each system against their respective baselines is shown in Fig. 11b. Systems A and B have maximum speedups of 166× and 21.1×, respectively, when using `Stage 4` metadata with a memory pool. Average speedups for each system are 52.9 × and 8.44×, respectively. System A consistently outperforms System B, despite the fact that it has an older GPU than that in System B. This happens because the specifications of the GPU in System A are more amenable to CUDA programs than those in System B. The System A GPU has a 12.1% higher clock frequency, 11.1% more CUDA cores, and 100% more usable shared memory per thread block. The GPU in System B is more expensive and sacrifices CUDA performance for additional performance for conventional graphics applications, with focus on technologies like ray-tracing and 3D-rendering.

In Fig. 12, we compare the GPU method with different combinations of our performance enhancements with the CPU method (see Sect. 4.1) using the highest-performing CPU (found on System B) and the linkage data set. The GPUs provide a maximum speedup of 15.00× and an average speedup of 3.07×. When looking at each system separately, System A has a maximum speedup of 15.00×, a minimum speedup of 2.13× and an average speedup of 4.77×. System B, on the other hand, had a maximum speedup of 3.41×, a minimum speedup of 0.832× and an average speedup of 1.37×. Four of the GPU performance enhancements on System B do not provide speedup over the CPU method. On average, these are 13% slower than the CPU method. This only occurs on System B when a combination of enhancement strategies is not used. When compared to System A, System B has a GPU that is less CUDA capable and a CPU with a faster clock rate and enhanced parallelism. This scenario reduces the overall difference in performance between the CPU and GPU methods on System B. This illustrates the sensitivity that CPU/GPU combinations can have when using various enhancement strategies.

Although it has the highest storage cost, using fully decompressed columns as metadata reduces execution time by completely avoiding the the decompression routine. Figure 13 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 691× speedup and an average of 383× speedup.

Figure 14 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`



(a) Execution time

(b) Speedup relative to the baseline GPU method

Fig. 11 Shown are execution times and speedups (relative to baseline implementation) using the linkage data set: **a** with data reuse **b** storing `Stage 2` as 64-bit **c** storing `Stage 2` as 32-bit **d** storing `Stage 4` as 32-bit **e** using the memory pool **f** storing `Stage 2` as 64-bit and using memory pool **g** storing `Stage 2` as 32-bit and using memory pool **h** storing `Stage 4` as 32-bit and using memory pool. **b**–**h** all incorporate data reuse

2 and 32-bit `Stage 4` metadata with a memory pool in (e) and (f), respectively.

Data structure reuse (shown in Fig. 14a) 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 Fig. 14b,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 Fig. 14d. 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 Fig. 14e, 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.

9 Related Work

There has been a significant amount of research conducted in the area of GPUs and database systems. We briefly describe a few of the most relevant works. Govindaraju et al. [21] presented several GPU algorithms from common database operations, including predicates, Boolean combinations, and aggregation. He et al. [23] designed and implemented a relational join algorithm for GPUs. Their approach was able to achieve a 7× speedup over an optimized CPU implementation. Bakkum and Skadron [4] implemented an SQLite command processor on the GPU. Their implementation achieved 70× speedup of `SELECT` queries. Rui and Tu [36] implemented hash joins and sort-merge join algorithms for the GPU. They were able to achieve speedups of up to 4.3× for their hash joins and 12.8× for their sort-merge joins over the respective CPU implementations. These works considered specific database operations and not indexing techniques.

There has been an abundance of work related to using GPUs to increase the efficiency of specific database indices. For example, Gosink et al. [20] created a parallel indexing data structure that uses bin-based data clusters. They showed that their GPU implementation could achieve 3× speedup over their CPU implementation. The fast architecture sensitive tree (FAST) presented in [26] is a configurable binary index tree. FAST was approximately 1.7× faster on the GPU than the CPU. Similarly, Kim et al. [27] showed that their GPU algorithm for R-tree traversal outperformed the traditional recursive R-tree traversals when answering multi-dimensional range queries. Our work pertains to WAH compressed bitmap indices.

Several works have explored the benefits of using the GPU to create bitmap indices. For example Fusco et al. [18] demonstrated that greater throughout of bitmap creation could be achieved using GPU implementations over

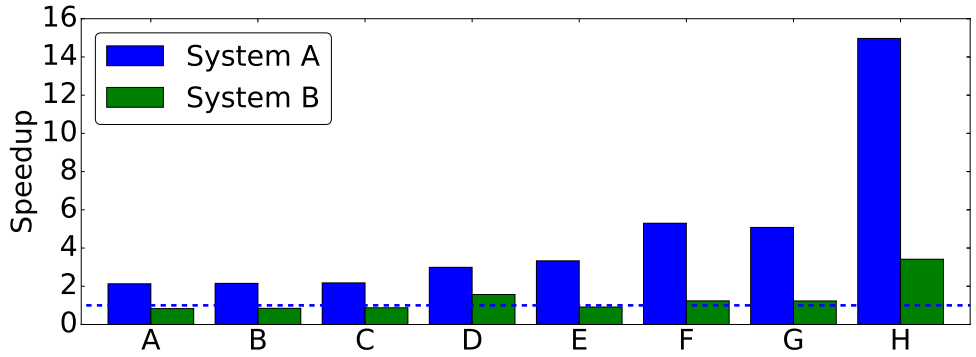


Fig. 12 Shown are experiments done on the linkage data set where the GPU speedups are relative to the CPU implementation on System B: **a** with data reuse **b** storing `Stage 2` as 64-bit **c** storing `Stage 2` as 32-bit **d** storing `Stage 4` as 32-bit **e** using the memory pool **f** storing `Stage 2` as 64-bit and using memory pool **g** storing `Stage 2` as 32-bit and using memory pool **h** storing `Stage 4` as 32-bit and using memory pool. **b**–**h** all incorporate data reuse. The dashed horizontal line indicates a speedup of 1

CPU implementations of WAH, and a related compression scheme, position list word-aligned hybrid (PLWAH) [15]. Similarly, Chen et al. [10] showed that a GPU implementation of their maximized stride with carrier (MASC) [43] bitmap index compression scheme was 19.5× faster at constructing a compressed bitmap index their CPU implementation. These works focused only on the initial creation of the bitmap index and did not address query processing, which is the focus of our work.

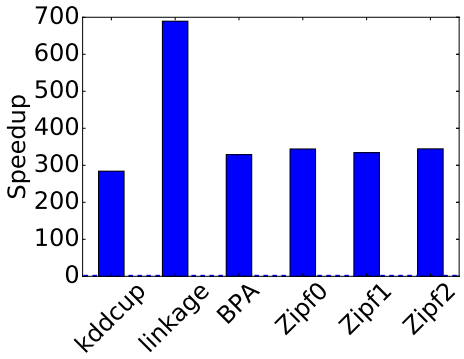


Fig. 13 Speedup provided by using decompressed bit vectors as "metadata" relative to the baseline GPU

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 [45]. Other compression schemes have employed variable alignment length [12, 22]. 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 [8, 9, 11, 15, 17, 47]. 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

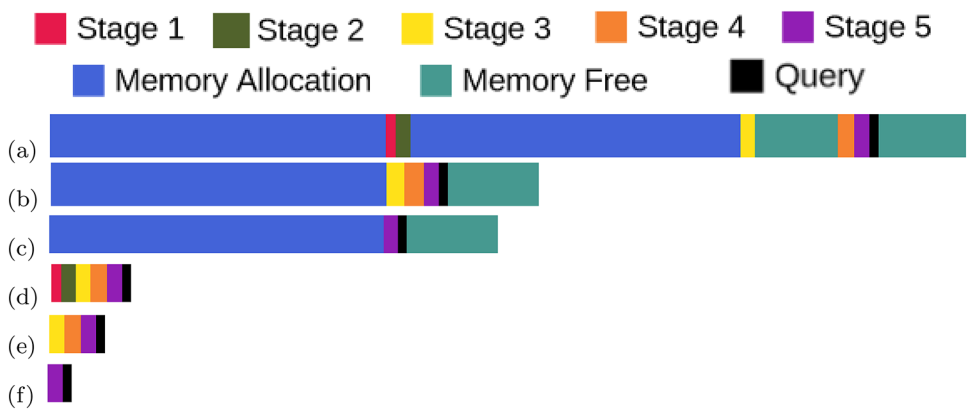


Fig. 14 A representative (albeit approximate) view of execution profiles of six identical query executions on the *linkage* database with varying enhancement strategy configurations. Query execution progresses from left to right. Longer bars correspond to longer execution times. While execution profiles for other databases exhibited slight variations, their interpretations remained consistent

GPU. However, other work has used metadata to speed up the processing of bitmap queries on the CPU. Enhanced WAH (EWAH) [47] splits a fill word into two halves. The most-significant half encodes a run much like WAH, but the least significant half stores metadata that encodes the number of succeeding literals. When performing bitwise operations between two vectors, this encoded metadata can be exploited to "short-circuit" the operation. This short-circuiting allows successive words to be skipped. Similarly, [39, 41] used externally stored metadata to enable query time short-circuiting for WAH and PLWAH, respectively.

There has been a generous body of work that has explored the benefits of memory pools on GPU in a variety of applications. For example, Hou et al. [24] 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. [42] 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. [48] 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], Nelson et al. [33, 34], and Tran et al. [40]. Andrzejewski and Wrembel introduced WAH and PLWAH [15] 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. At this time, Nelson et al. present the only other work to process WAH 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. The work presented in this paper is a direct extension of [40]. We have added an operation cost analysis of the GPU-WAH decompression algorithm. This analysis provides context for the improvements realized by our enhancements. Another addition is a discussion and analysis of sources and potential impacts of warp divergence on CUDA capable GPUs. We also present a Poisson distribution model that can be used to estimate the size requirements of the memory pool used in our approach. Additionally, we outlined an parallel reduction algorithm for processing WAH range queries on the GPU. We also greatly expanded the experimental study of our approaches leading to new insights into the importance of architectural details when selecting a GPU to process WAH range queries.

10 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 the current state-of-the-art approach. The results of our study showed that the data reuse, metadata, and memory pool strategies provided average speedups of 13.9×, 16.7×, and 22.1×, respectively. Combining these techniques provided an average of 66.2× 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. When comparing our GPU-WAH range query performance enhancement techniques against the high-end CPU, we were able to achieve a maximum speedup of 15.0× and an average speedup of 3.07×.

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. Additional GPUs would enhance 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 GPU architectures.

Acknowledgements JMM and JS would like to acknowledge the University of St. Thomas College of Arts and Science Dean's office for generously funding some of the computational resources that made this study possible.

Author contributions Algorithmic design was primarily done by JMM, JS, and DC with secondary contributions by BT and BS. The implementation of all algorithms was primarily performed by BT and BS with secondary development by JMM and JS. All parties played an equal role in manuscript development.

Funding Information BT and BS received Undergraduate Research Funding from the University of St. Thomas.

Data Availability Statement Please contact the corresponding author to obtain the data used in this manuscript.

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no competing interests.

References

1. Andrzejewski W, Wrembel R (2010) GPU-WAH: applying GPUs to compressing bitmap indexes with word aligned hybrid. In: International conference on database and expert systems applications. Springer, Berlin, pp 315–329
2. Andrzejewski W, Wrembel R (2011) GPU-PLWAH: GPU-based implementation of the PLWAH algorithm for compressing bitmaps. Control Cybern 40:627–650
3. Antoshenkov G (1995) Byte-aligned bitmap compression. In: Proceedings DCC'95 data compression conference, p 476. IEEE
4. Bakkum P, Skadron K (2010) Accelerating sql database operations on a gpu with cuda. In: Proceedings of the 3rd workshop on general-purpose computation on graphics processing units, pp 94–103
5. Bonneville power administration, http://www.bpa.gov
6. Bradlow E, Gangwar M, Kopalle P, Voleti S (2017) The role of big data and predictive analytics in retailing. J Retail 93:79–95
7. Chambi S, Lemire D, Kaser O, Godin R (2016) Better bitmap performance with roaring bitmaps. Softw Pract Exp 46(5):709–719
8. Chang J, Chen Z, Zheng W, Cao J, Wen Y, Peng G, Huang W (2015) Splwah: a bitmap index compression scheme for searching in archival internet traffic. In: IEEE international conference on communications (ICC), pp 7089–7094
9. Chang J, Chen Z, Zheng W, Wen Y, Cao J, Huang W (2014) Plwah+: a bitmap index compressing scheme based on plwah. In: ACM/IEEE symposium on architectures for networking and communications systems (ANCS), pp 257–258
10. Chen Z, Wen Y, Cao J, Zheng W, Chang J, Wu Y, Ma G, Hakmaoui M, Peng G (2015) A survey of bitmap index compression algorithms for big data. Tsinghua Sci Technol 20(1):100–115
11. Colantonio A, Di Pietro R (2010) Concise: compressed 'n' composable integer set. Inf Process Lett 110(16):644–650
12. Corrales F, Chiu D, Sawin J (2011) Variable length compression for bitmap indices. In: Database and expert systems applications, pp 381–395
13. CUDA, C.: Best practice guide (2019). https://docs.nvidia.com/cuda/cuda-c-best-practices-guide
14. Davenport T, Dyche J (2013) Big data in big companies. Tech. rep, International Institute for Analytics
15. Deliège F, Pedersen TB (2010) Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In: International conference on extending database technology, EDBT '10, pp 228–239
16. Erevelles S, Fukawa N, Swaynea L (2016) Big data consumer analytics and the transformation of marketing. J Bus Res 69:897–904
17. Fusco F, Stoecklin MP, Vlachos M (2010) Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. VLDB 3(2):1382–1393
18. Fusco F, Vlachos M, Dimitropoulos X, Deri L (2013) Indexing million of packets per second using gpus. In: Proceedings of the 2013 conference on internet measurement conference, IMC '13, pp 327–332
19. Gelado I, Garland M (2019) Throughput-oriented gpu memory allocation. In: Proceedings of the 24th symposium on principles and practice of parallel programming, pp 27–37
20. Gosink LJ, Wu K, Bethel EW, Owens JD, Joy KI (2009) Data parallel bin-based indexing for answering queries on multi-core architectures. In: Winslett M (ed) Scientific and statistical database management, pp 110–129
21. Govindaraju NK, Lloyd B, Wang W, Lin M, Manocha D (2004) Fast computation of database operations using graphics processors. In: Proceedings of the 2004 ACM SIGMOD international conference on management of data, pp 215–226
22. Guzun G, Canahuate G, Chiu D, Sawin J (2014) A tunable compression framework for bitmap indices. In: 2014 IEEE 30th international conference on data engineering, pp 484–495. IEEE
23. He B, Yang K, Fang R, Lu M, Govindaraju N, Luo Q, Sander P (2008) Relational joins on graphics processors. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data, pp 511–524
24. Hou Q, Sun X, Zhou K, Lauterbach C, Manocha D (2011) Memory-scalable GPU spatial hierarchy construction. IEEE Trans Visual Comput Graphics 17(4):466–474
25. Huang X, Rodrigues CI, Jones S, Buck I, Hwu Wm (2010) Xmalloc: a scalable lock-free dynamic memory allocator for many-core machines. In: 2010 10th IEEE international conference on computer and information technology, pp 1134–1139. IEEE
26. Kim C, Chhugani J, Satish N, Sedlar E, Nguyen AD, Kaldewey T, Lee VW, Brandt SA, Dubey P (2010) Fast: fast architecture sensitive tree search on modern cpus and gpus. In: Proceedings of the 2010 ACM SIGMOD international conference on management of data, pp 339–350
27. Kim J, Kim SG, Nam B (2013) Parallel multi-dimensional range query processing with r-trees on gpu. J Parallel Distrib Comput 73(8):1195–1207
28. Lichman M (2013) UCI machine learning repository. http://archive.ics.uci.edu/ml
29. Lloyd S (1982) Least squares quantization in pcm. IEEE Trans Inf Theory 28(2):129–137
30. Marr B (2018) Starbucks: using big data, analytics and artificial intelligence to boost performance. Forbes. https://www.forbes.com/sites/bernardmarr/2018/05/28/starbucks-using-big-data-analytics-and-artificial-intelligence-to-boost-performance/#5784902e65cd
31. McAfee A, Brynjolfsson E (2012) Big data: the management revolution. Harvard Business Review, pp 61–68
32. Merrill D (2016) Cub: Cuda unbound. http://nvlabs.github.io/cub
33. Nelson M, Sorenson Z, Myre JM, Sawin J, Chiu D (2019) 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
34. Nelson M, Sorenson Z, Myre JM, Sawin J, Chiu D (2020) Parallel acceleration of CPU and GPU range queries over large data sets. J Cloud Comput 9(1):1–21
35. Nvidia C (2020) Programming guide
36. Rui R, Tu YC (2017) Fast equi-join algorithms on GPUs: design and implementation. In: Proceedings of the 29th international

conference on scientific and statistical database management, pp 1–12
37. Sariyar M, Borg A, Pommerening K (2011) Controlling false match rates in record linkage using extreme value theory. J Biomed Inform 44(4):648–654
38. Steinberger M, Kenzel M, Kainz B, Schmalstieg D (2012) Scatteralloc: massively parallel dynamic memory allocation for the GPU. In: 2012 Innovative parallel computing (InPar), pp 1–10. IEEE
39. Taufen B, Sawin J, Chiu D (2017) Improving the querying efficiency of the plwah bitmap algorithm. In: Proceedings of the 21st international database engineering and applications symposium, pp 127–134
40. Tran B, Schaffner B, Sawin J, Myre JM, Chiu D (2020) Increasing the efficiency of GPU bitmap index query processing. In: To appear in Proceedings of the 25th international conference on database systems for advanced applications (DASFAA'20)
41. Velez M, Sawin J, Ingerson A, Chiu D (2016) Improving bitmap execution performance using column-based metadata. In: 2016 IEEE 4th international conference on future internet of things and cloud (FiCloud), pp 371–378. IEEE
42. Wang L, Ye J, Zhao Y, Wu W, Li A, Song SL, Xu Z, Kraska T (2018) 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
43. Wen Y, Wang H, Chen Z, Cao J, Peng G, Huang W, Hu Z, Zhou J, Guo J (2016) Masc: a bitmap index encoding algorithm for fast data retrieval. In: IEEE international conference on communications (ICC), pp 1–6
44. Wu J, Di B, Sun J, Chen H, Zhong X, Hu D, Huang C (2019) A fast and secure GPU memory allocator. In: 2019 IEEE 21st international conference on high performance computing and communications; IEEE 17th international conference on smart city; IEEE 5th international conference on data science and systems (HPCC/SmartCity/DSS), pp 146–153. IEEE
45. Wu K, Otoo EJ, Shoshani A (2002) Compressing bitmap indexes for faster search operations. In: Proceedings 14th international conference on scientific and statistical database management, pp 99–108. IEEE
46. Wu K, Otoo EJ, Shoshani A (2006) Optimizing bitmap indices with efficient compression. ACM Trans Database Syst 31(1):1–38
47. Wu K, Otoo EJ, Shoshani A, Nordberg H (2001) Notes on design and implementation of compressed bit vectors. Tech. Rep. LBNL/PUB-3161, Lawrence Berkeley National Laboratory
48. You S, Zhang J, Gruenwald L (2013) 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
49. Zaker M, Phon-Amnuaisuk S, Haw SC (2008) An adequate design for large data warehouse systems: bitmap index versus b-tree index. Int J Comput Commun 2:39–46