Nelson *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2020) 9:44
https://doi.org/10.1186/s13677-020-00191-w

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Parallel acceleration of CPU and GPU range queries over large data sets

Mitchell Nelson[1†], Zachary Sorenson[1†], Joseph M. Myre[1†], Jason Sawin[1*†] and David Chiu[2†]

Abstract

Data management systems commonly use bitmap indices to increase the efficiency of querying scientific data. Bitmaps are usually highly compressible and can be queried directly using fast hardware-supported bitwise logical operations. The processing of bitmap queries is inherently parallel in structure, which suggests they could benefit from concurrent computer systems. In particular, bitmap-range queries offer a highly parallel computational problem, and the hardware features of graphics processing units (GPUs) offer an alluring platform for accelerating their execution. In this paper, we present four GPU algorithms and two CPU-based algorithms for the parallel execution of bitmap-range queries. We show that in 98.8% of our tests, using real and synthetic data, the GPU algorithms greatly outperform the parallel CPU algorithms. For these tests, the GPU algorithms provide up to 54.1× speedup and an average speedup of 11.5× over the parallel CPU algorithms. In addition to enhancing performance, augmenting traditional bitmap query systems with GPUs to offload bitmap query processing allows the CPU to process other requests.

Keywords: Bitmap indices, WAH compression, Range queries, GPU

Introduction

Contemporary applications are generating a staggering amount of data. For example, the Square Kilometre Array Pathfinders are a collection of radio telescopes can generate 70 PB per year [1]. Efficient querying of massive data repositories relies on advanced indexing techniques that can make full use of modern computing hardware. Though many indexing options exist, *bitmap indices* in particular are commonly used for read-only scientific data [2, 3]. A bitmap index produces a coarse representation of the data in the form of a binary matrix. This representation has two significant advantages: it can be compressed using run-length encoding and it can be queried directly using fast primitive CPU logic operations. This paper explores algorithmic designs that enable common bitmap-index queries to execute on computational accelerators, graphics processing units (GPUs), in particular.

A bitmap index is created by discretizing a relation's attributes into a series of bins that represent either value ranges or distinct values. Consider the example shown in Table 1. The *Produce* relation records the quantity of particular fruits available at a market. A potential bitmap index that could be built from *Produce* is shown below it. The f bitmap bins under the **Fruit** attribute represent the distinct fruit items that can be referred to in the relation: f_0 encodes *Apple*, f_1 represents *Orange*, and so on. Where the f bins represent discrete values, the q bins under **Quantity** represent value ranges. Specifically, q_0 represents $[0, 100)$, q_1 is $[100, 200)$, q_2 is $[200, 300)$, q_3 is $[300, 400)$, and q_4 is $[400, \infty)$. Each row in the bitmap represents a tuple from the relation. The specific bit pattern of each row in the bitmap is generated by analyzing the attributes of the corresponding tuple. For each attribute in a tuple, a value of 1 is placed in the bin that encodes that value and a value of 0 is placed in the remaining bins for that attribute. For example, consider tuple t_1 from *Produce*. Its **Fruit** attribute is *Apple*, so a 1 is placed in f_0 and the remaining f bins in that row are assigned a 0. The **Quantity** of t_1 is 548,

*Correspondence: jason.sawin@stthomas.edu
†Mitchell Nelson, Zachary Sorenson, Joseph M. Myre, Jason Sawin and David Chiu contributed equally to this work.
[1]Department of Computer and Information Sciences, University of St. Thomas, 2115 Summit Ave., 55105 Saint Paul, Minnesota, USA
Full list of author information is available at the end of the article

Table 1 Example relation and corresponding bitmap

Produce

ID	Fruit	Quantity
t_1	Apple	548
t_2	Orange	233
t_3	Kiwi	257
t_4	Durian	3

Bitmap of *Produce*

ID	Fruit				Quantity				
	f_0	f_1	f_2	f_3	q_0	q_1	q_2	q_3	q_4
t_1	1	0	0	0	0	0	0	0	1
t_2	0	1	0	0	0	0	1	0	0
t_3	0	0	1	0	0	0	1	0	0
t_4	0	0	0	1	1	0	0	0	0

this value falls into the $[400, \infty)$ range represented by bin q_4, so that bin is assigned a 1 and all other q bins get a 0.

Bitmap indices are typically sparse, which makes them amenable to compression using hybrid run-length encoding schemes. Numerous such schemes have been developed (e.g. [4–8]), and among these, one of the most prominent is the *Word-Aligned Hybrid* code (WAH) [9]. It has been shown that WAH can compress a bitmap to just a small fraction of its original size [10].

One major benefit of bitmap indices is that they can be queried directly, greatly reducing the number of tuples that must be retrieved from disk. Considering the example from Table 1, suppose a user executes a *range query* of the form:

```
SELECT * FROM Produce
WHERE Quantity >= 100;
```

This query can be processed by executing the following bitmap formula $q_1 \vee q_2 \vee q_3 \vee q_4 = r$ where r is the result column of a bitwise `OR` between the q_1, q_2, q_3, and q_4 bins. Every row in r that contains a 1 corresponds with a tuple in Produce that has a **Quantity** in the desired range. Moreover, a WAH compressed bitmap can be queried directly without first being decompressed in a very similar manner.

Notice that the above *range* query example could easily be executed in parallel. For example, one process could be execute $r_1 = q_1 \vee q_3$, another could perform $r_2 = q_2 \vee q_4$, and the final result could be computed by $r = r_1 \vee r_2$. It is clear that the more bins that are needed to be processed to answer a range query, the more speedup a parallel approach could realize. This describes a classic parallel reduction, requiring $log_2(n)$ rounds to obtain a result.

In the past decade, the applicability of graphics processing units (GPUs) has expanded beyond graphics to general-purpose computing. GPUs are massively parallel computational accelerators that augment the capabilities of traditional computing systems. For example, an NVIDIA Titan X GPU is capable of executing 57,344 concurrent threads. Coupled with high-bandwidth memory, GPUs are a natural fit for throughput focused parallel computing and may be able to increase the efficiency of data management systems. Previous works have shown that WAH-style compression, decompression, and point queries can be processed efficiently on GPUs [11, 12]. We have built upon these efforts to create several algorithms exploiting various GPU architectural features to accelerate WAH range query processing [13].

The specific contributions of this paper are:

- We present two parallel CPU algorithms and four parallel GPU algorithms for executing WAH range queries.
- We present refinements to the GPU algorithms that exploit hardware features to improve performance.
- We present an empirical study on both real-world and synthetic data. The results of our study show:
  - The highest performing parallel CPU algorithm provides an average of 2.18× speedup over the alternative parallel CPU algorithm.
  - The GPU algorithms are capable of outperforming the CPU algorithms by up to 54.1× and by 11.5× on average.
  - When compared to only the best performing CPU tests, the GPU algorithms still provide up to 5.64× speedup for queries of 64 bins and 6.44× for queries of 4, 8, 16, 32, and 64 bins.

The remainder of the paper is organized as follows. We provide overviews of WAH algorithms and GPUs as computational accelerators in the "Background" section. We describe our parallel WAH query algorithms in the "Parallel range queries" section. Our experimental methodology is presented in the "Evaluation methodology" section. The "Results" section presents the results of our empirical study with discussion in the "Discussion of results" section. We describe related works in the "Related work" section before presenting conclusions and plans for future work in the "Conclusion and future work" section.

Background

Word-Aligned hybrid compression (WAH)

WAH compresses bitmap bins (bit vectors) individually. During compression, WAH chunks a bit vector into groups of 63 consecutive bits. Figure 1 shows an example bit vector. This vector consists of 189 bits, implying the relation it is taken from contained that many tuples. In the example, the first chunk contains both ones and

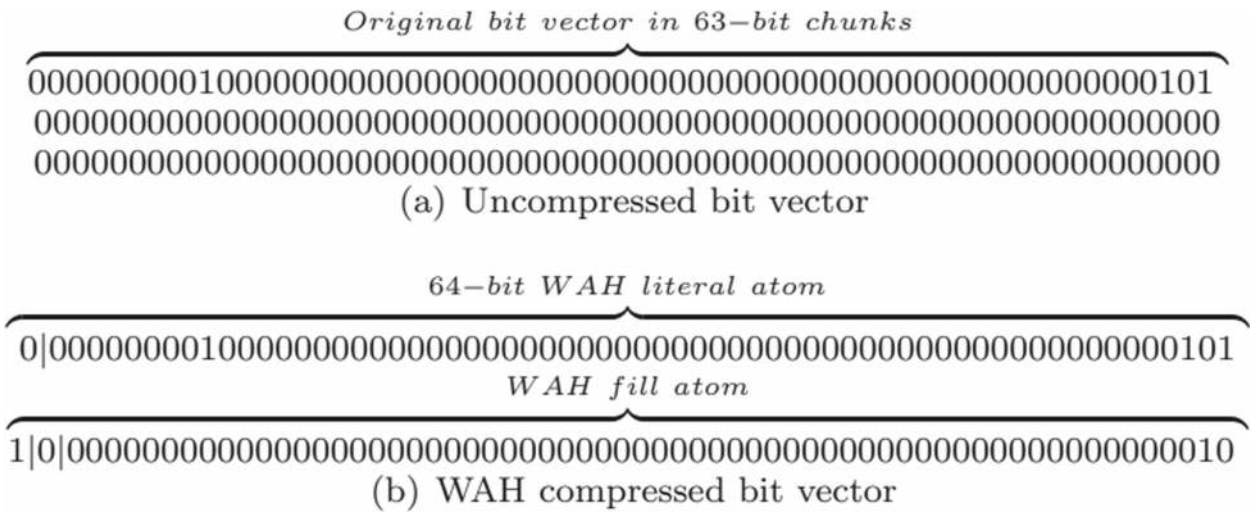

Fig. 1 An example of WAH compression using one literal and two fills

zeros, making it heterogeneous. The last two chunks are homogeneous, containing only zeros.

Heterogeneous chunks are encoded in WAH *literal* atoms. For efficient query processing, WAH atoms are tailored to the system word size. *Literal* atoms have the form $(flag, lit)$, where the most-significant-bit (MSB) or *flag* is set to 0, indicating the atom is a *literal*. The remaining 63 bits, *lit*, record verbatim the heterogeneous chunk of the original bit vector.

WAH groups sequences of homogeneous chunks and encodes them in *fill* atoms. *Fill* atoms have the form $(flag, value, len)$. The *flag*, or MSB, is set to 1 designating the atom as a *fill*. The second-MSB is *value* and records the value of the run being encoded: 1s or 0s. The remaining 62 bits are *len*, which records the number of homogeneous chunks that have been clustered together. In Fig. 1, the last two chunks are homogeneous, so they are grouped into a *fill*. The chunks are a run of 0's so the *value* bit is set to 0 and *len* is set to two since the group is made of two chunks.

One of the advantages of WAH is that the compressed bit vectors can be queried directly without decompression. For example, let X and Y be compressed bit vectors and $Z = X \circ Y$ where $\circ$ is a bitwise logical operation, and Z is the resulting bit vector. The standard query algorithm for compressed bit vectors treats X and Y as stacks of atoms. Processing starts by popping the first atom off each vector. The atoms are then analyzed until they are fully processed, or exhausted. When an atom is exhausted, the next atom from that vector is popped.

There are three possible atom-type pairings during processing:

1 **(literal,literal)**: Let a_i and a_j be the current literals being processed from X and Y respectively. In this case, a *result* literal atom is added to Z, where

   $Z.result.lit = X.a_i.lit \circ Y.a_j.lit$.

   After this operation, a new atom is popped from both X and Y as both operand literal atoms have been exhausted.
2 **(fill,fill)**: In this case, a fill atom *result* is added to Z of the form,

   $Z.result.value = X.a_i.value \circ Y.a_j.value$ and $Z.result.len = Min(X.a_i.len, Y.a_j.len)$.

   Processing fills produces side effects for the operand atoms. Specifically,

   $a_i.len = a_i.len - result.len$ and $a_j.len = a_j.len - result.len$.

   This will exhaust the atom with the shorter *len* or both if they are the same.
3 **(fill,literal)**: in this case *result* is a literal. Assume $X.a_i$ is the fill word. If $X.a_i.value$ is 1 then $Z.result.lit = Y.a_j.lit$, else $Z.result.lit = 0$. This will exhaust $Y.a_j$ and result in $X.a_i.len = X.a_i.len - 1$.

When applied to bit-vector pairs, the above approach handles *point queries*. This can easily be extended to apply to *range queries*. Range queries seek tuples with values that fall between an upper and lower bound. A bitmap index can be used to process such queries in the following manner: $R = A_1 \vee A_2 \vee ...A_n$ where A_i is a bitmap bin that encodes attributes within the desired range. The resulting bit vector R will indicate the tuples that should be retrieved from disk. A simple iterative algorithm is often employed to solve range queries. First R is initialized to A_1, then $R \leftarrow R \vee A_j$ is repeated for all j such that $2 \leq j \leq n$.

Graphics processing units (GPUs)

Using NVIDIA's compute unified device architecture (CUDA) programming platform for GPUs, thousands of

threads can be organized into 1-, 2-, or 3-dimensional Cartesian structures. This layout naturally maps to many computational problems. Hierarchically, these structures comprise thread grids, thread blocks, and threads as shown in Fig. 2. Threads are executed in groups of 32, *ergo*, thread blocks are typically composed of $32m$ threads, where m is a positive integer. These groups of 32 threads are known as warps.

The NVIDIA GPUs memory hierarchy is closely linked to its organization of threads. The memory hierarchy is composed of global, shared, and local memory. Global memory is accessible to all threads. Each thread block has private access to its own low-latency shared memory ($\sim 100\times$ less than global memory latency) [14]. Each thread also has its own private local memory.

To fully realize high-bandwidth transfers from global memory, it is critical to *coalesce* global-memory accesses. For a global memory access to be coalesced, it must meet two criteria: 1) the memory addresses being accessed are sequential and 2) the memory addresses span the addresses $32n$ to $32n + 31$, for some integer, n. Coalesced global memory accesses allow the GPU to batch memory transactions in order to minimize the total number of memory transfers.

A classic challenge to ensuring high computational throughput on a CUDA capable GPU is warp divergence (sometimes called thread divergence). Warp divergence is a phenomenon that occurs when threads within the same warp resolve a branching instruction (commonly stemming from loops or if-else statements) to different outcomes. At an architectural level, CUDA GPUs require all threads within a warp to follow the same execution pathway. When warp divergence occurs, a CUDA GPU will execute the multiple execution pathways present in the warp serially. This makes it important to minimize the amount of branching instructions in a CUDA program as the loss in computational throughput can significantly reduce performance (as seen in [15, 16]).

Parallel range queries

In the "Word-Aligned hybrid compression (WAH)" section we briefly described how a range query of the form $A_1 \vee \ldots \vee A_n$, where A_i is a bitmap bin can be solved iteratively. However, the same problem could be solved in parallel by exploiting independent operations. For example, $R_1 = A_1 \vee A_2$ and $R_2 = A_3 \vee A_4$ could be solved simultaneously. An additional step of $R_1 \vee R_2$ would yield the final result. This pattern of processing is called a parallel reduction. Such a reduction transforms a serial $\mathcal{O}(n)$ time process to a $\mathcal{O}(\log n)$ algorithm, where n is the number of bins in the query.

Further potential for parallel processing arises from the fact that row operations are independent of one another (e.g., the reduction along row_i is independent of the reduction along row_{i+1}). In actuality, independent processing of rows in compressed bitmaps is very challenging. The difficulty comes from the variable compression achieved by fill atoms. In the sequential-query algorithm this is not a problem as compressed bit vectors are treated like stacks, where only the top atom on the stack is processed and only after all of the represented rows have been exhausted is it removed from the stack. This approach ensures row alignment. Without additional information, it would be impossible to exploit row independence. When selecting an atom in the middle of a compressed bit vector, its row number cannot be known without first examining the preceding atoms to account for the number of rows compressed in fills.

In the remainder of this section, we present parallel algorithms for processing WAH range queries using GPUs and multi-core CPUs.

GPU decompression strategy

All of our GPU-based range query algorithms rely on the same preparations stage. In this stage, the CPU sends compressed columns to the GPU. As concluded in [12], it is a natural decision to decompress bitmaps on GPUs when executing queries as it reduces the communication costs between CPU and GPU. Once the GPU obtains the compressed columns, it decompresses them in parallel using Algorithm 1. Once decompressed, the bit vectors involved in the query are word-aligned. This alignment makes the bitwise operation on two bit vectors embarrassingly parallel and an excellent fit for the massively parallel nature of GPUs.

Algorithm 1 Parallel column decompression

```
1: procedure DECOMPRESSION(Cols)
2:     ▷ Cols is a collection of compressed bit vectors
3:     for all C ∈ Cols in parallel do
4:         dCols ← dCols ⋃ Decomp(C, C_size, C_decomp_size)
5:     end for
6:     return dCols
7: end procedure
```

The procedure *Decompression* (Algorithm 1) takes, as input, a set of compressed bit vectors, *Cols*. This is the set of bins that have been identified as necessary to answer a range query. *Decompression* processes each bit vector in *Cols* in parallel, sending each of them to the *Decomp* function. Ultimately, *Decompression* returns a set of fully decompressed bit vectors.

The *Decomp* procedure (Algorithm 2) is a slightly modified version of the decompression algorithm presented

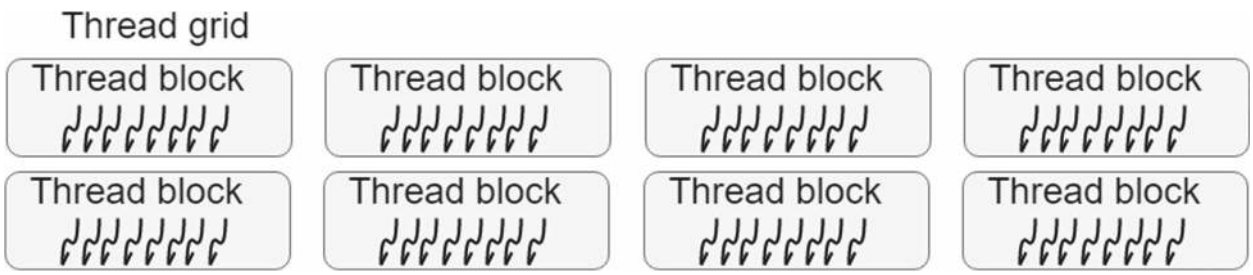

Fig. 2 The organizational hierarchy used by NVIDIA's CUDA to structure groups of threads. Shown is a 2 × 4 2-dimensional thread grid of thread blocks, where each 1-dimensional thread block is composed of 8 threads along the x-dimension

by Andrzejewski and Wrembel [11]. One notable implementation difference is that 32-bit words were used in [11, 12], while we use 64-bit words. We also modified their algorithm to exploit data structure reuse. Algorithm 2 takes a single WAH compressed bit vector, *CData*. It also requires, the size of *CData* in number of 64-bit words, *Csize*, and *Dsize* which is the size number of 64-bit words required to store the decompressed version of *CData*.

Algorithm 2 Parallel decompression of data

```
1:  procedure DECOMP(CData, CSize, DSize)
2:      for i ← 0 to CSize − 1 in parallel do
3:          if CData(i)_63 = 0b then
4:              DecompSizes[ i] ← 1
5:          else
6:              DecompSizes[ i] ← CData(i)_0→61
7:          end if
8:      end for
9:      StartingPoints ← exclusive scan on DecompSizes
10:     EndPoints an array of zeroes, size DSize
11:     for i ← 1 to CSize − 1 in parallel do
12:         EndPoints[ StartingPoints[ i] −1] ← 1
13:     end for
14:     WordIndex ← exclusive scan on EndPoints
15:     for i ← 0 to DSize − 1 in parallel do
16:         tempWord ← CData[ WordIndex[ i] ]
17:         if tempWord_63 = 0b then
18:             DecompData[ i] ← tempWord
19:         else
20:             if tempWord_62 = 0b then
21:                 DecompData[ i] ← 0_64
22:             else
23:                 DecompData[ i] ← 0_1 + 1_63
24:             end if
25:         end if
26:     end for
27:     return DecompData
28: end procedure
```

In Algorithm 2, lines 2 to 8, *Decomp* generates the *DecompSizes* array, which is the same size as *CData*. For each WAH atom in *Cdata*, the *DecompSizes* element at the same index will hold the number of words being represented by that atom. The algorithm does this by generating a thread for each atom in *Cdata*. If *Cdata*[*i*] is a literal, then $thread_i$ writes a 1 in *DecompSizes*[*i*], as that atom encodes 1 word (line 4). If *Cdata*[*i*] holds a fill atom, which are of the form (*flag*, *value*, *len*) (see "Background" section), then $thread_i$ writes the number of words compressed by that atom, or *len*, to *DecompSizes*[*i*] (line 6).

Next, *Decomp* performs an exclusive scan (parallel element summations) on *DecompSizes* storing the results in *StartingPoints* (line 9). *StartingPoints*[*i*] contains the total number of decompressed words represented by *CData*[0] to *CData*[$i - 1$], inclusive. *StartingPoints*[*i*] ∗63 is the number of the bitmap row first represented in *CData*[*i*].

In lines 10 to 13, the *EndPoints* array is created and initializes. This array has a length of *Dsize* and is initially filled with 0s. *Decomp* then processes each element of *StartingPoints* in parallel. A 1 is assigned to *EndPoints* at index *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. At line 14 another exclusive scan is performed, this time on *EndPoints*. The result of this scan is saved to *WordIndex*. *WordIndex*[*i*] stores the index to the atom in *CData* that contains the information for the *ith* decompressed word.

The final for-loop (lines 15 - 26) is a parallel processing of every element of *WordIndex*. For each *WordIndex* element, the associated atom is retrieved from *CData*. If *CData*[*WordIndex*[*i*]] is a literal atom (designated by a 0 value in the most significant bit (MSB)), then it is placed directly into *DecompData*[*i*]. Otherwise, the atom must be a fill. 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.

Figure 3 illustrates a thread access pattern for the final stage of *Decomp*. As shown, *CData*, the WAH compressed bit vector, is composed of three literal atoms (L0, L1, and L2) and two fill atoms (the shaded sectors).

For each literal, *Decomp* uses a thread that writes the value portion of the atom to the *DecompData* bit vector. Fill atoms need as many threads as the number of compressed words the fragment represents. For example, consider the first fill in *CData*, it encodes a run of three words of 0. *Decomp* creates three threads all reading the same compressed word but writing 0 in the three different locations in *DecompData*. If a run of 1s had been encoded, a value of 0x3*FFF FFFF FFFF FFFF* would have been written instead of 0.

GPU range query execution strategies

Here we present four methods for executing range queries in parallel on GPUs. These are column-oriented access (COA), row-oriented access (ROA), hybrid, and ideal hybrid access approaches. These approaches are analogous to structure-of-arrays, array-of-structures, and a blend thereof. Structure-of-arrays and array-of-structures approaches have been used successfully to accelerate scientific simulations on GPUs [17, 18], but differ in the how data is organized and accessed which can impact GPU efficiency.

Column-oriented access (COA)

Our COA approach to range query processing is shown in Algorithm 3. The COA procedure takes a collection of decompressed bit vectors needed to answer the query and performs a column oriented reduction on them. At each level of the reduction, the bit vectors are divided into two equal groups: low-order vectors and high-order vectors. The s variable in Algorithm 3 stores the divide position (lines 5 and 14). During processing, the first low-order vector is paired with the first high-order, as are the seconds of each group and so on (lines 8 and 9). The bitwise operation is performed between these pairs. To increase memory efficiency, the result of the query operation is written back to the low order column (Algorithm 3, line 11). The process is then repeated using only the low-order half of the bit vectors as input until a single decompressed bit vector remains. The final bit-vector containing the result can then be copied back to the CPU.

Algorithm 3 Column-oriented access query processing

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

Figure 4a shows the COA reduction pattern for a range query across bit vectors 0 through 3. A 1-dimensional

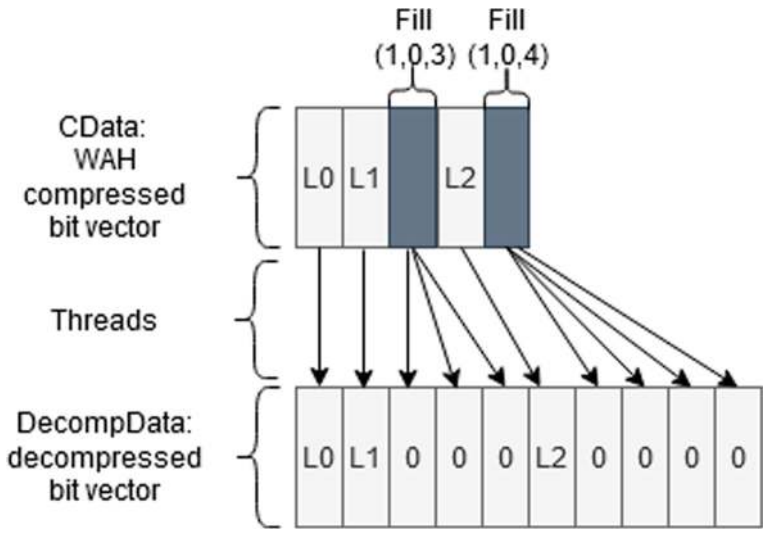

Fig. 3 Algorithm 2's thread access pattern

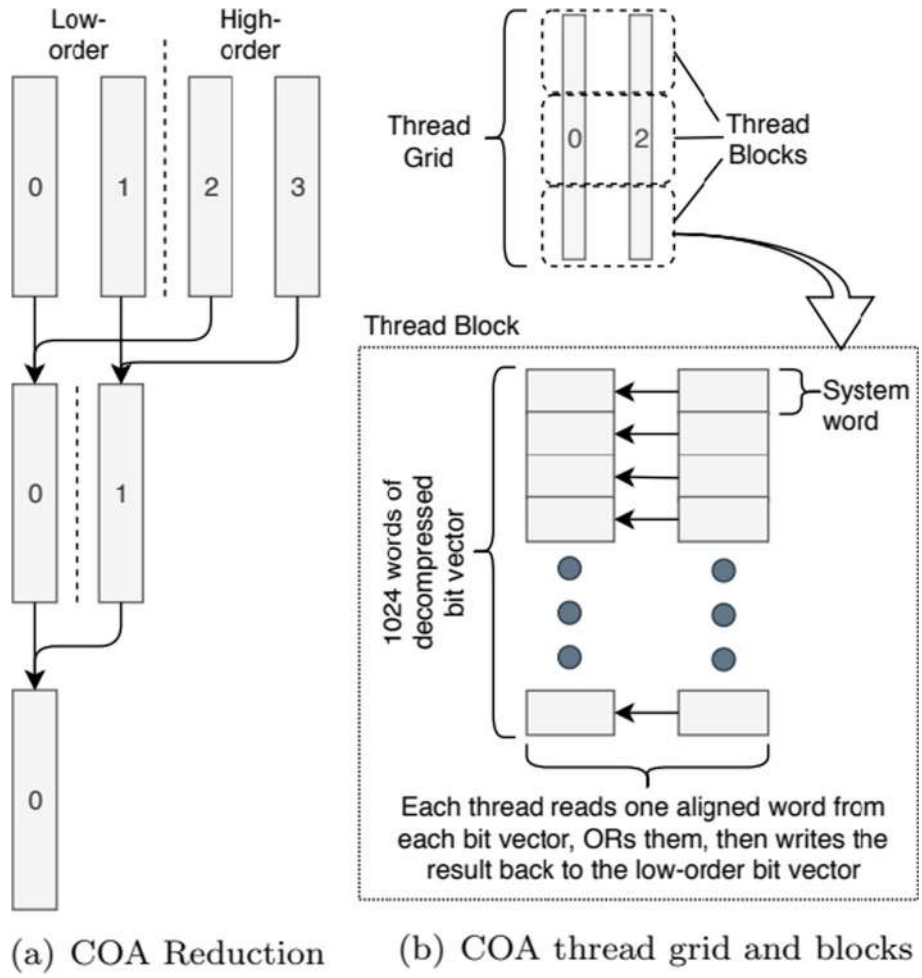

Fig. 4 Shown here are (**a**) the reduction pattern used by the COA method and (**b**) the mapping of thread blocks within thread grids to the WAH query data

thread grid is assigned to process each pair of bit vectors. Note that multiple thread blocks are used within the grid, as a single GPU thread block cannot span the full length of a decompressed bit vector. Figure 4b shows how the thread grid spans two columns and illustrates the inner workings of a thread block. As shown, a thread block encompasses 1024 matched 64-bit word pairs from two columns. A thread is assigned to each pair of words. Each thread performs the *OR* operation on its word pair and writes the result back to the operand word location in the low ordered column. As each thread block only has access to a very limited shared memory (96 kB for the GPU used in this study), and since each round of the COA reduction requires the complete result of the column pairings, all of COA memory reads and writes have to be to global memory. Specifically, given a range query of m bit vectors, each with n rows, and a system word size of w bits, the COA approach performs $(2m-2)\frac{n}{w}$ coalesced global memory reads and $(m-1)\frac{n}{w}$ coalesced global memory writes on the GPU.

Row-oriented access (ROA)

Algorithm 4 presents our ROA approach to range query processing. Because all rows are independent, they can be processed in parallel. To accomplish this, ROA uses many 1-dimensional thread blocks that are arranged to create a one-to-one mapping between thread blocks and rows (Algorithm 4, line 5).

This data access pattern is shown in Fig. 5. The figure represents the query $C_0 \vee C_1 \vee C_2 \vee C_3$, where C_x is a decompressed bit vector. As shown, the individual thread blocks are represented by rectangles with perforated borders. Unlike COA, where thread blocks only span two columns, the ROA thread blocks span all columns of the query (up to 2048, 2× the maximum number of threads in a thread block.)

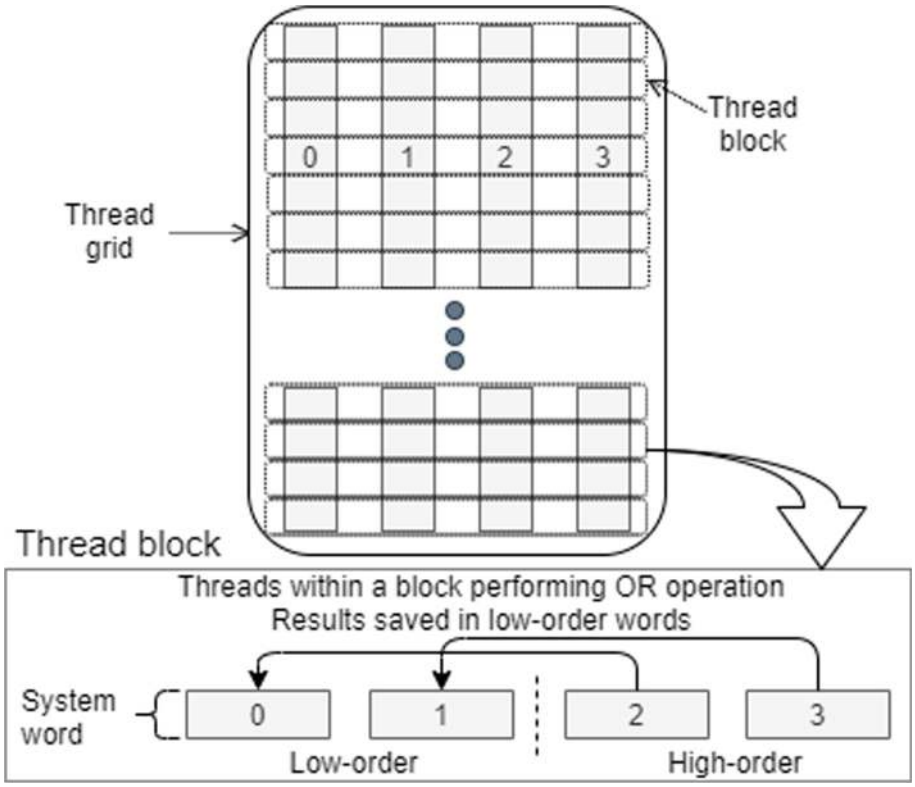

Fig. 5 The data access pattern and work performed by each ROA thread block

Algorithm 4 Row-oriented access query processing

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

Inside any given ROA thread block, the column access pattern within it is identical to the COA pattern (Algorithm 4 line 8-11). The words of the row are partitioned into *low-order* and *high-order* by column ID. Each thread performs a bitwise OR on word pairs, where one operand word is from the low-order columns, and the other is from the high-order set (shown in the *Thread block* of Fig. 5). The results of the operation are written back to the low order word.

Like COA, a ROA reduction has $log_2(n)$ levels, where n is the number of bit vectors in the query. However, all of ROA processing is limited in scope to a single row. By operating along rows, the ROA approach loses coalesced global memory accesses as row data are not contiguous in memory. However, for the majority of queries, the number of bit vectors is significantly less than the number of words in a bit vector. This means that ROA can use low-latency GPU shared memory to store the row data (up to 96 kB) and intermediate results necessary for performing the reduction. Using shared memory for the reduction avoids repeated reads and writes to high-latency global memory ($\sim$ 100$\times$ slower than shared memory). Given a range query of m bins, each with n rows, and a system word size of w bits, the ROA approach performs $\frac{mn}{w}$ global memory reads and $\frac{n}{w}$ global memory writes. A significant reduction of both relative to COA.

Hybrid

We form the hybrid approach to range query processing by combining the 1-dimensional COA and ROA data

access patterns into 2-dimensional thread blocks. These 2D thread blocks are tiled to provide complete coverage of the query data. An example tiling is shown in Fig. 6. To accomplish this tiling the hybrid method uses a thread grid of $p \times q$ thread blocks, where p and q are integers. Each thread block is composed of $k \times j$ threads and spans $2k$ columns and j rows, where k and j are integers. With this layout, each thread block can use the maximum of 1024 threads.

A single thread block in the hybrid process performs the same work as multiple ROA thread blocks stacked vertically. A major difference being that thread blocks in the hybrid process do not span all bit vectors. Using these 2-dimensional thread blocks provides the hybrid approach the advantages of both coalesced memory accesses of COA, and ROA's use of GPU shared memory to process the query along rows. The disadvantage of the hybrid approach is that the lowest order column of each thread block along the rows must undergo a second round of the reduction process to obtain the final result of the range query. This step combines the answers of the individual thread block tiles.

The hybrid process is shown in Algorithm 5 where the first round of reductions are on lines 8-20 and the second round of reductions are on lines 22-34.

Due to the architectural constraints of NVIDIA GPUs, the hybrid design is limited to processing range queries of $\leq 2^{22}$ bins. This is far beyond the scope of typical bitmap range queries and GPU memory capacities. Given a range query of m bins, each with n rows, a system word size of w, and k thread blocks needed to span the bins, up to $(m+k)\frac{n}{w}$ global memory reads and $(k+1)\frac{n}{w}$ global memory writes are performed. Although the hybrid approach requires more global memory reads and writes than the ROA approach, its use of memory

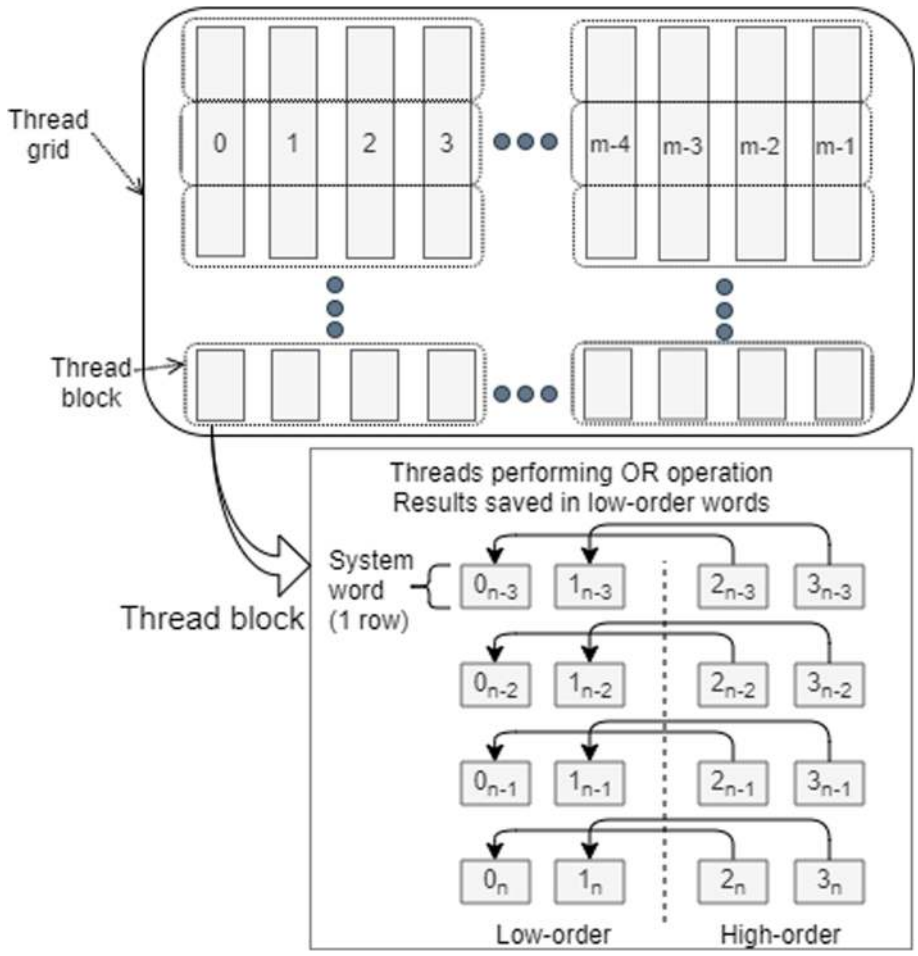

Fig. 6 The data access pattern and reduction work performed by each hybrid thread block. In standard scenarios, each thread block would comprise 1024 threads

Algorithm 5 Hybrid query processing

```
1:  procedure HYBRID(Cols, p, q)
2:      ▷ Cols is a collection of decompressed bit vectors
3:      ▷ p is the number of tiles in the x-dimension
4:      ▷ q is the number of tiles in the y-dimension
5:      m ← |Cols|▷ the number of bit vectors in the query
6:      n ← |Cols_0|  ▷ the number of words in a bit vector
7:      ▷ First set of loops performs reductions within tiles
8:      for rb ← 0 to n/q − 1 in parallel do
9:          for t ← rb∗n/q to (rb+1)∗n/q−1 in parallel do
10:             s ← m/(2 ∗ p)
11:             while s ≥ 1 do
12:                 for c ← 0 to s − 1 in parallel do
13:                     c1 ← Cols_c
14:                     c2 ← Cols_{c+s}
15:                     c1_t ← c1_t ∨ c2_t
16:                 end for
17:                 s ← s/2
18:             end while
19:         end for
20:     end for
21:     ▷ Second set of loops performs reductions across tiles
22:     for rb ← 0 to n/q − 1 in parallel do
23:         for t ← rb∗n/q to (rb+1)∗n/q−1 in parallel do
24:             s ← p/2
25:             while s ≥ 1 do
26:                 for c ← 0 to s − 1 in parallel do
27:                     c1 ← Cols_c
28:                     c2 ← Cols_{c+s}
29:                     c1_t ← c1_t ∨ c2_t
30:                 end for
31:                 s ← s/2
32:             end while
33:         end for
34:     end for
35:     return Cols_0
36: end procedure
```

coalescing can enhance the potential for computational throughput.

Ideal hybrid

In practice, most WAH range queries involve less than 1024 columns. This mean that in most query scenarios it is possible to map a single 2-dimensional thread block tile across multiple rows and all of the columns (bit vectors) of the query. This purely vertical tiling is shown in Fig. 7. Such a tiling improves throughput by allowing each thread block to comprise the maximum of 1024 threads. Like the hybrid method, each thread block retains the advantages of coalesced memory accesses and the use of GPU shared memory. Further, this tiling pattern eliminates the need for a second round of reduction. The result of this arrangement is the ideal hybrid algorithm as described in Algorithm 6.

Algorithm 6 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
```

The theoretical expressions for global reads and writes in the hybrid algorithm agree that an "ideal" hybrid layout is one where a single thread block of $k \times j$ threads spans all $2k$ columns. Multiple $k \times j$ thread blocks are still used to span all of the rows. This layout limits the number of global writes in the first round to 1 and removes the need to perform the second reduction between thread blocks along rows. For processing a range query of m bins, each with n rows, and a system word size of w, the ideal hybrid layout thereby reduces the total number of global memory reads and writes to $\frac{mn}{w}$ and $\frac{n}{w}$, respectively. These are the same quantities obtained for ROA, but the ideal hybrid method guarantees a higher computational throughput as each $k \times j$ thread block has 1024 threads.

Multi-core CPU methods

For an experimental baseline, we created a CPU-based parallel algorithm for processing range queries. Most multi-core CPUs cannot support the number of concurrent operations needed to fully exploit all of the available parallelism in WAH bitmap query processing. For this reason, we limited the CPU algorithm to two approaches: 1)

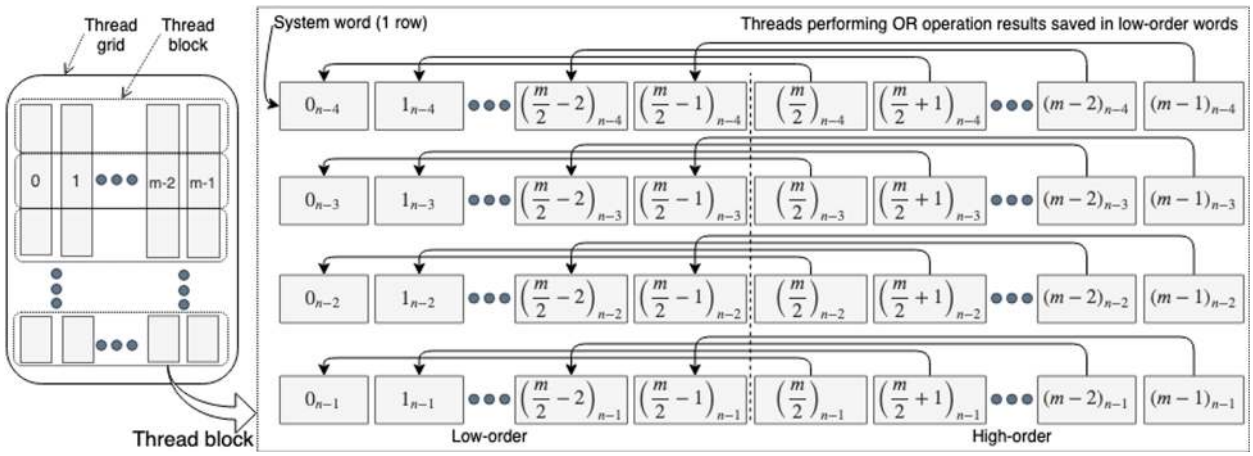

Fig. 7 The data access pattern and reduction work performed by each ideal hybrid thread block. In standard scenarios, each thread block would comprise 1024 threads

a baseline approach that iterates through bit vectors to execute a query and 2) a COA style reduction approach.

Given an *np*-core CPU, approach 1 uses OpenMP [19] to execute up to *np* parallel bitwise operations on paired compressed bit vectors. Once a set of paired bit vectors is processed, the CPU iterates to execute up to *np* parallel bitwise operations on the result and the next remaining bit vector to process.

Approach 2 uses OpenMP to execute up to *np* parallel bitwise operations on paired compressed bit vectors for any reduction level. If more than *np* bit vector pairs exist in a given reduction level, the CPU must iterate until all pairs are processed and the reduction level is complete. The range-query result is obtained once the final reduction level is processed. The pattern of the CPU reduction process is similar to the COA pattern shown in Fig. 4a.

Theoretical analysis

The GPU and CPU algorithms presented earlier in this section all perform the same amount of work. Performance variations between those algorithms come from data access patterns and the parallelism that the CPU or GPU architectures are capable of achieving when using those patterns.

In ideal scenarios (unlimited threads, no resource contention, etc.), all of the GPU algorithms yield the same run time complexity of $\mathcal{O}(n\log_2(m)/t)$, where m is the number of bit vectors in the query, n is the number of system words in a decompressed bit vector, and t is the number of executing threads. The n/t term corresponds to many parallel bitwise operations between paired bit vectors and the $\log_2(m)$ term corresponds to the number of reduction levels required to produce a result.

Even in ideal scenarios, CPUs are not capable of the same degree of parallelism as GPUs. Our two CPU algorithms are manifestations of focused application of parallelism. The iterative approach (algorithm 1) focuses parallelism on the bitwise operations between paired bit vectors. This yields a run time complexity of $\mathcal{O}(nm/t)$. The reduction approach (algorithm 2) focuses parallelism on performing a reduction, yielding a run time complexity of $\mathcal{O}(nlog_2(m))$.

It is important to note that the idealized scenarios used to guide our theoretical analysis of our CPU and GPU algorithms are not realistic. The behavior of real implementations of these algorithms will deviate from the theoretical descriptions due to the influence of architectural effects, including finite parallelism, resource contention, cache effects, and memory usage.

Evaluation methodology

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

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

- KDD – this data set was procured from KDD (knowledge discovery and data mining) Cup'99 and is network flow traffic. The data set contains 4,898,431 rows and 42 attributes [20]. Continuous attributes

were discretized into 25 bins using Lloyd's Algorithm [21], resulting in 475 bins.

- `linkage` – This data set contains 5,749,132 rows and 12 attributes. The attributes were discretized into 130 bins. The data are anonymized records from the Epidemiological Cancer Registry of the German state of North Rhine-Westphalia [22].
- `BPA` (Bonneville power administration) – this data set contains measurements reported from 20 synchrophasors deployed over the Pacific Northwest power grid over approximately one month [23]. Data from all synchrophasors arrive at a rate of 60 measurements per second and are discretized into 1367 bins. There are 7,273,800 rows in this data set.
- `Zipf` – we generate three synthetic data sets using a Zipf distribution. Zipf distributions represent clustered approaches to discretization and mimic real-world data [24]. The Zipf distribution generator assigns each bit a probability of: $p(k, n, skew) = (1/k^{skew})/\sum_{i=1}^{n}(1/i^{skew})$ 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 $k = 10$, $n = 10$, and create three different synthetic Zipf data sets using $skew = 0, 1$, and 2. These generated data sets each contain 100 bins (*i.e.,* ten attributes discretized into ten bins each) and 32 million rows.

For each of these six data sets (three real and three synthetic), we use the GPU based range-query methods described in the "GPU decompression strategy" section (i.e., COA, ROA, hybrid, and ideal hybrid), as well as the parallel CPU methods described in the "Multi-core CPU methods" section (processing iteratively and using a reduction), to execute a range query of 64 random bit vectors. This query size is sufficiently large that there is negligible variation in execution time when different bit vectors are selected. We also conduct a test where query size is varied. For this test we use the highest performing CPU and GPU methods to query all data sets using 4,8,16,32, and 64 bins.

For GPU methods, we use the maximum number of threads per thread block (32 for ROA and 1024 for the others) and the maximum number of thread blocks per thread grid required for the problem at hand. When using the CPU methods, we conduct multiple tests using 1, 2, 4, 8, and 16 cores.

Each experiment was run six times and the execution time of each trial was recorded. To remove transient program behavior, the first result was discarded. The arithmetic mean of the remaining five execution times is shown in the results. We use the averaged execution times to calculate our comparison metric, speedup.

Results

Here we present the results of the experiments described in the previous Section. We first present a comparison of GPU performance enhancement over the two CPU methods organized by data set. We then compare the relative performance of the two CPU methods. A focused view of GPU performance relative to the highest performing CPU scenarios (using 16 cores) for each data set is then provided. We then examine the relative performance of only the GPU methods. Finally, we present GPU and CPU results for queries of varying size.

Results are shown for all GPU tests compared to the iterative CPU method, organized by data set, in Fig. 8. Iterative CPU range query performance typically improves with additional cores for every data set. The only exception being the BPA data set when transitioning from 8 to 16 cores. The GPU methods outperform the iterative CPU method in 96.2% of these tests with an average speedup of 14.50×. The GPU methods are capable of providing a maximum speedup of 54.14× over the iterative CPU method. On average, the GPU methods provide 1.45×, 20.24×, 11.45×, 17.36×, 18.76×, and 17.72× speedup for the KDD, linkage, BPA, Zipf ($skew = 0$), Zipf ($skew = 1$), and Zipf ($skew = 2$) data sets, respectively.

Results for GPU tests compared to the CPU method using a reduction, organized by data set, are shown in Fig. 9. The performance of the CPU reduction method improves with additional cores for every data set. The GPU methods outperform the parallel reduction CPU method in 100% of these tests with an average speedup of 8.50×. The GPU methods are capable of providing a maximum speedup of 26.01× over the CPU reduction method. On average, the GPU methods provide 7.03×, 9.47×, 8.60×, 7.57×, 9.46×, and 9.11× speedup for the KDD, linkage, BPA, Zipf ($skew = 0$), Zipf ($skew = 1$), and Zipf ($skew = 2$) data sets, respectively.

Speedups provided by the CPU reduction method over the iterative CPU approach (when using 16 cores) are shown for each data set in Fig. 10. The CPU reduction method provides an average of 2.18× speedup over the iterative CPU approach and a maximum of 3.77×. KDD is the only data set where the CPU reduction approach is not beneficial and incurs a 0.44× slowdown.

A comparison of the GPU methods to the highest performing iterative CPU (16 core) tests is shown in Fig. 11a. On average, the GPU methods provide 4.97× speedup over the iterative CPU method when using 16 cores. The KDD data set is the only instance where the GPU methods do not outperform the CPU method using 16 cores.

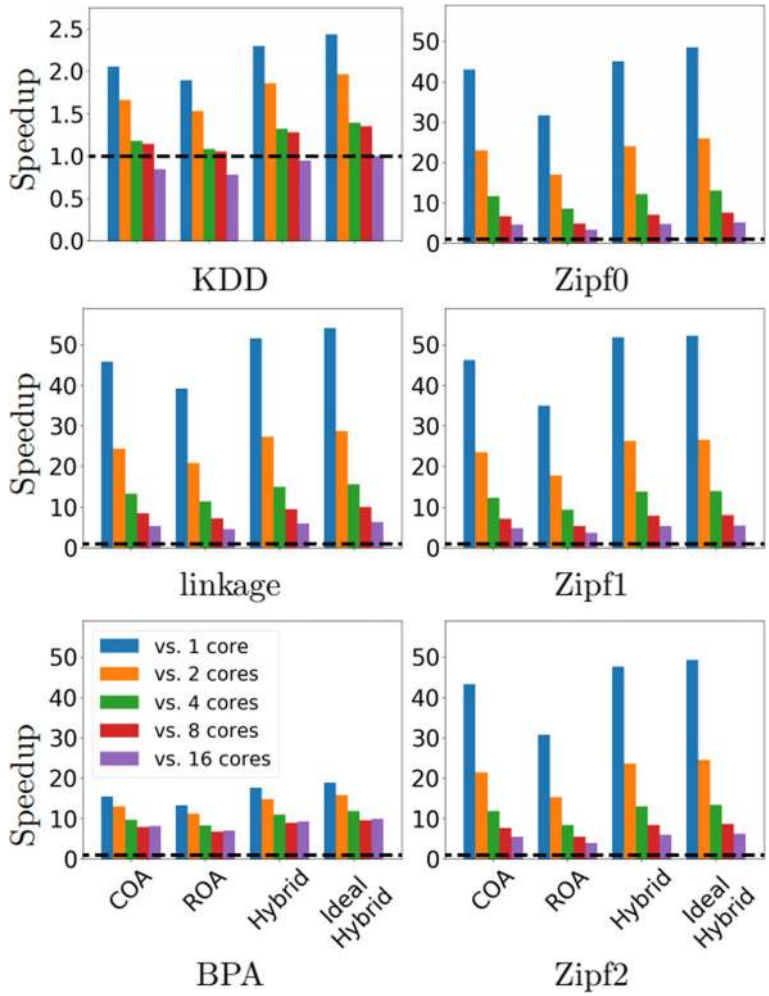

Fig. 8 Speedups (vertical axes) for the GPU methods compared to the iterative CPU method (using the number of cores shown in the legend) grouped by the GPU range query method (horizontal axes). The Zipf data set skews are appended (e.g., Zipf0 is the Zipf data set with a skew of 0). The horizontal dashed line indicates a speedup of 1×. All plots share the same legend

In this scenario, only the ideal hybrid GPU approach outperforms the iterative CPU method (using 16 cores) with a speedup that is > 1 (1.002×). Despite this, the average speedup provided by the ideal hybrid method over the CPU using 16 cores is 5.69×.

A comparison of the GPU methods to the highest performing CPU reduction (16 core) tests is shown in Fig. 11b. The GPU methods outperform the CPU reduction method (using 16 cores) in all tests. On average, the GPU methods provide 2.16× speedup over the reduction CPU method when using 16 cores.

Speedups provided by the COA, hybrid, and ideal hybrid GPU methods relative to the lowest-performing GPU method (ROA) are shown in Fig. 12. For these tests, the COA and hybrid methods always outperform the ROA method, with the hybrid methods providing the most significant performance improvement over ROA. On average, the COA, hybrid, and ideal hybrid methods provide 25.09%, 38.47%, and 45.23% speedup over the ROA method, respectively.

The effects that varying query size had on our algorithms are shown in Fig. 13a and b. In Fig. 13a, the solid lines represent the time required by the iterative CPU method (using 16 cores) for the given query size which is shown in number of bit-vectors. The solid lines in Fig. 13b, show the times required by the CPU reduction method (using 16 cores) for the given query sizes. For comparison, the time required by the ideal hybrid GPU approach are shown as dashed lines in both figures. GPU execution times are relatively consistent compared to the CPU times which grow at a faster rate (seen in Fig. 13a and b). For varied query size, the GPU method outperforms the iterative

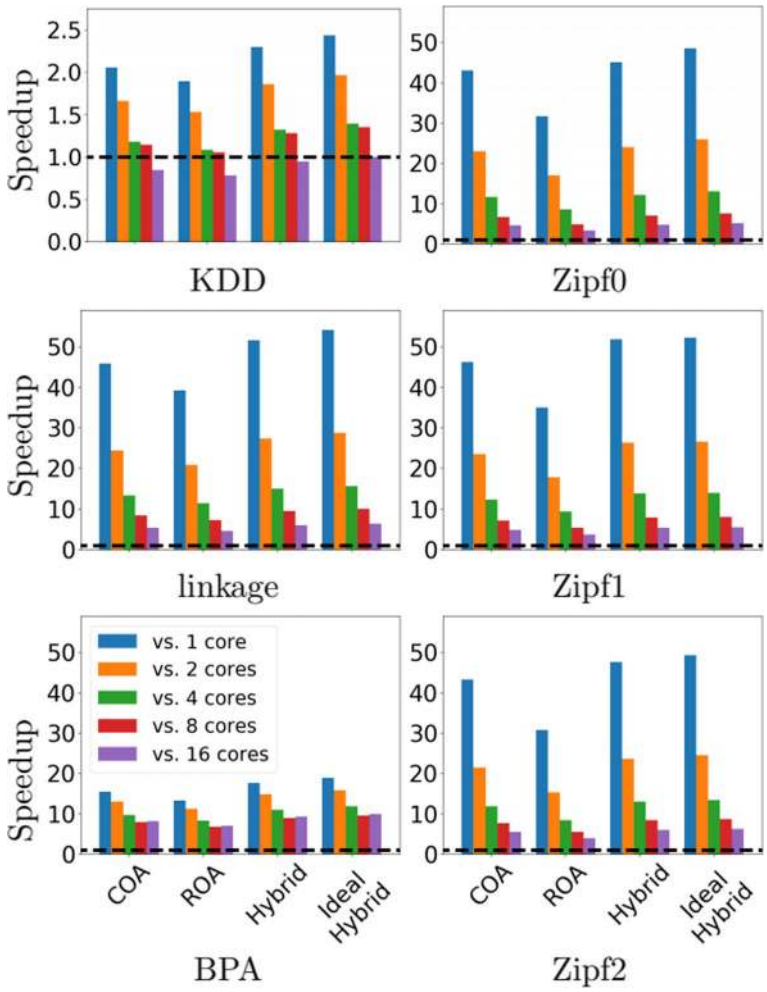

Fig. 9 Speedups (vertical axes) for the GPU methods compared to the CPU reduction method (using the number of cores shown in the legend) grouped by the GPU range query method (horizontal axes). The Zipf data set skews are appended (e.g., Zipf0 is the Zipf data set with a skew of 0). The horizontal dashed line indicates a speedup of 1×. All plots share the same legend

CPU method by a factor of 6.44×, on average, and the CPU reduction method by 3.06×, on average.

Discussion of results

Using a general (not related to bitmaps) benchmark suite of optimized GPU and optimized CPU programs, Intel found that GPUs outperformed CPUs by 3.5× on average [25]. The ideal hybrid method provides an average speedup of 4.0× relative to both parallel CPU methods (using 16 cores). This aligns well with expectations when comparing optimized GPU and CPU programs.

When compared to the parallel iterative CPU approach, a major factor determining relative GPU performance is the degree of the data set's column compression. This behavior is shown in Fig. 14. It is most consequential for tests using the KDD data set, which is the only data set where the GPU methods do not always outperform the iterative CPU method. This only occurs when the CPU method is used with 16 cores. The relatively high-performance of the CPU method is entirely due to the highly compressed nature of the KDD data set. The compressibility of the KDD data set is such that compressed queried bit vectors can be held entirely in cache. Further, the branch prediction and speculative execution capabilities of the CPU allow enhanced performance over GPUs when querying highly compressed bit vectors. GPUs have no such branch predictors and do not benefit from bitmap compression beyond storage and transmission efficiency. The remaining data sets did not exhibit the same degree of compression, thereby reducing CPU performance and enhancing GPU speedup. When data sets are less compressible, there is greater variance in the relative performance of the GPU methods when compared to the iterative CPU method. This is apparent in the variation

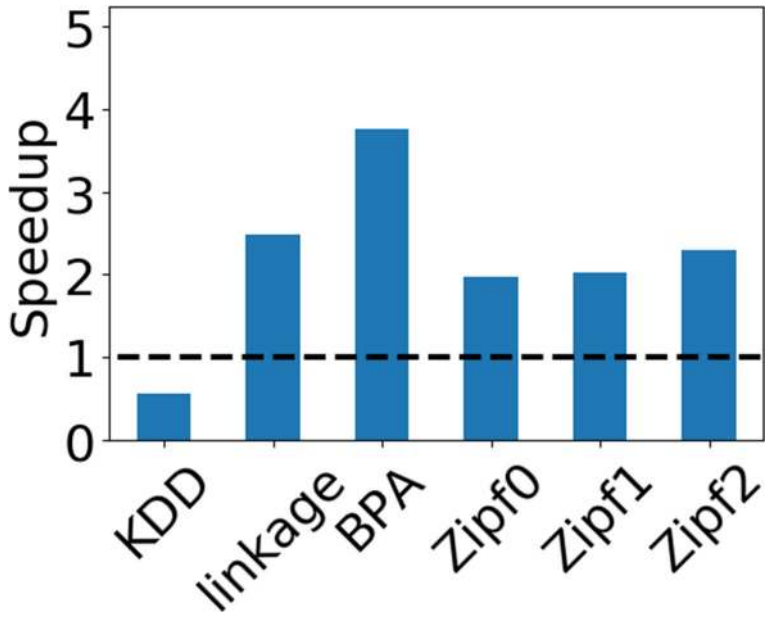

Fig. 10 Speedups provided by the CPU reduction method relative to the iterative CPU method. The Zipf data set skews are appended (e.g., Zipf0 has a skew of 0). The horizontal dashed line indicates a speedup of 1×

in speedup results across the GPU methods in Fig. 14a, where there is less variation for highly compressible data sets and more variation for less compressible data sets.

When compared to the parallel reduction CPU method, the effect of compression ratio is greatly reduced. This is seen in Fig. 14b, where the data look like damped versions of their counterparts from Fig. 14a. The relative performance of the GPU methods is reduced (compared to the relative GPU performance vs the iterative CPU approach). These features occur because the parallel reduction approach provides more consistent query execution behavior. The parallel reduction consecutively halves the amount data remaining to be processed until the query is completed. This halving means that the data of interest can be completely stored in lower latency CPU caches fairly early in the reduction. The same is not

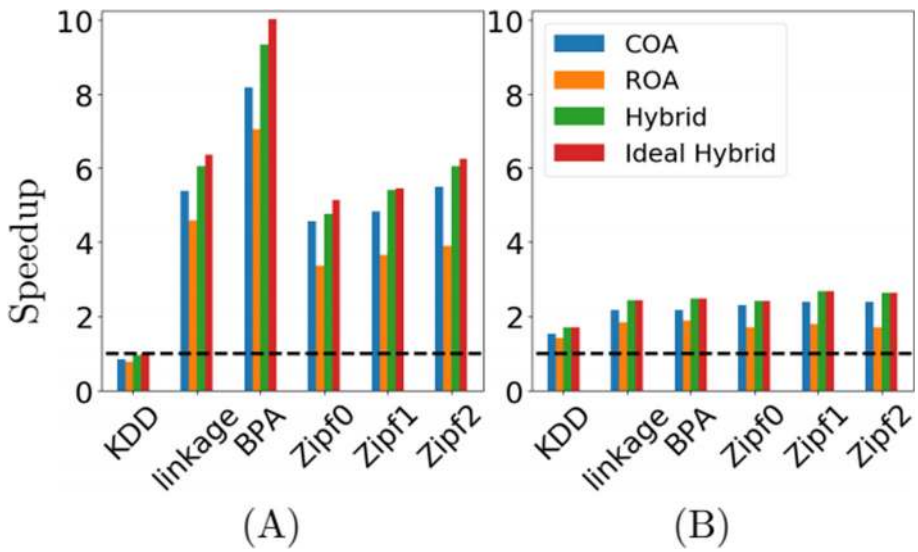

Fig. 11 Speedups for all GPU methods vs. **a** the iterative CPU method using 16 cores and vs. **b** the reduction CPU method using 16 cores. The Zipf data set skews are appended (e.g., Zipf0 is the Zipf data set with a skew of 0). The horizontal dashed line indicates a speedup of 1×. All plots share the same legend

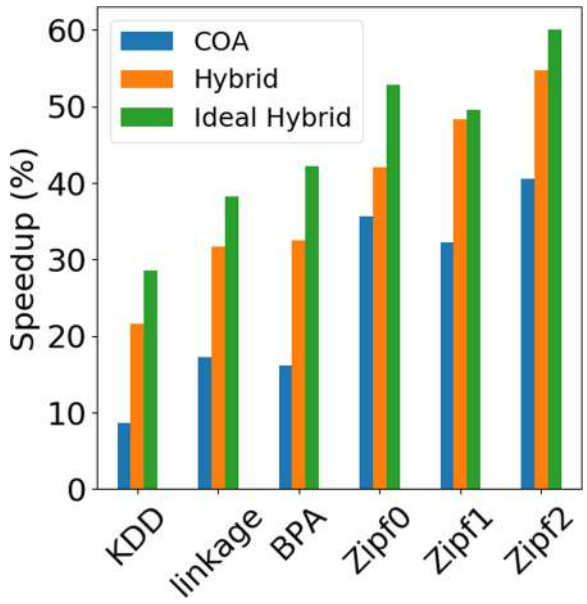

Fig. 12 Percent speedups for the GPU COA, hybrid, and ideal hybrid methods relative to the ROA method. The Zipf data set skews are appended (e.g., Zipf0 has a skew of 0). A speedup of 1× is a percent speedup of zero

true for the iterative approach which necessitates iterative loads of bit vectors from memory.

When varying query size, we find the ideal hybrid GPU method provides a consistent performance enhancement over the parallel CPU method (Fig. 13). The relatively consistent results for the GPU in these tests are due to the massively parallel nature of our algorithms. As the majority of the processing happens in parallel, the additional cost of adding columns is negligible. The synthetic data set Zipf skew 0-2 produces the only results where a variation in GPU execution time can be easily be observed. This variance is likely due to additional contention for memory resources as Zipf has more than 4× the number of rows than the largest real-world data set. The iterative CPU method (Fig. 13a) displays consistent behavior where execution times appear to be "plateauing" as query size increases. Conversely, the CPU reduction method (Fig. 13b) displays an execution time trough for queries between 8 and 32 (inclusive) bit-vectors in size. For larger queries, the execution times rise appreciably. This rise for queries of 64 columns demonstrates the consequences of using a reduction to handling queries that are beyond the parallelism of which the CPU is capable.

Each of our GPU methods can take advantage of certain GPU architectural features due to their differing data access patterns. These differences make each GPU method suited for particular types of queries. A listing of architectural advantages, disadvantages, and ideal queries is provided in Table 2.

An example of query suitability is apparent in our experimental results. In our tests, the ROA method is consistently outperformed by the COA method. This occurs because we limit our tests to queries of 64 columns, thereby limiting ROA thread blocks to 32 threads, far below the potential 1024 thread limit. The consequences of this are severely limiting the benefits of using shared memory and reducing the computational throughput of each thread block. For larger queries ($\leq$2048 columns), the ROA method could potentially outperform the COA method due to increased computational throughput. However, such queries are not commonly encountered in practice.

Related work

There has been a significant amount of research conducted in the area of bitmap indices and their compression. The work presented in this paper is concerned with the widely adopted WAH [9] bitmap compression scheme. However, there are many similar techniques. One of the first hybrid run-length encoding schemes was Byte-aligned Bitmap Compression (BBC) [4]. BBC uses byte-alignment to compress runs and which, in certain cases, allows it to achieve greater compression than other compression schemes [9]. This increase in compression is often achieved at the expense of query times. For this reason many of the recent encoding schemes (e.g., [5–8, 26–28]) use *system word* alignment. We believe that many of the bitmap-compression schemes could realize significant query speed-up by employing similar parallel algorithms as presented in this paper.

Previous works have explored parallel algorithms for bitmaps indices. Chou et al. [29] introduced FastQuery (and several later augmentations, e.g. [30, 31]) which provides a parallel indexing solution that uses WAH compressed bitmap indices. Su et al. [32] presented a parallel indexing system based on two-level bitmap indices. These works focused on generating the bitmaps in parallel and not necessarily the parallel processing of actual bitwise operations, nor did they implement their algorithms for GPUs.

With CUDA, GPUs have exhibited a meteoric rise in enhancing the performance of general-purpose computing problems. Typically, GPUs are used to enhance the performance of core mathematical routines [33–35] or parallel programming primitives [36, 37] at the heart of an algorithm. With these tools, GPUs have been used to create a variety of high-performance tools, including computational fluid dynamics models [38, 39], finite element methods [40], and traditional relational databases [41].

Several researchers have explored using hardware systems other than standard CPUs for bitmap creation and querying. Fusco, et al. [42] demonstrated that greater throughout of bitmap creation could be achieved using

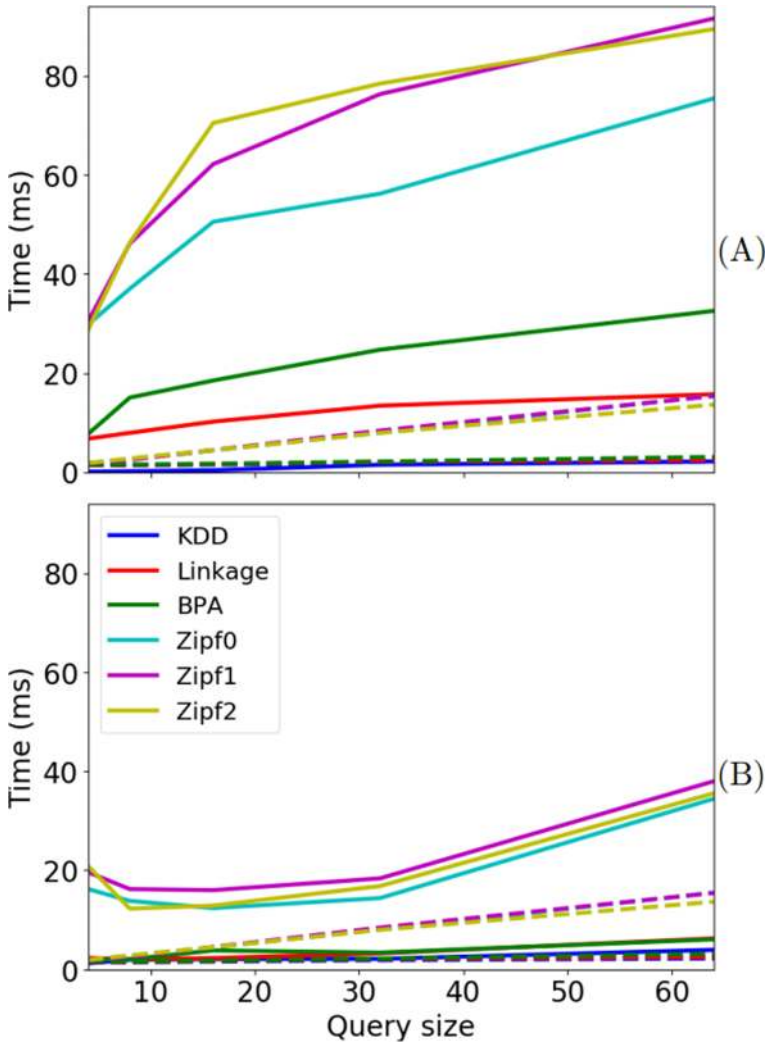

Fig. 13 Average execution times for the iterative CPU method (**a**) using 16 cores (solid lines), the CPU reduction method (**b**) using 16 cores, and the ideal hybrid GPU method (shown as dashed lines in **a** and **b**) by query size. The Zipf data set skews are appended (e.g., Zipf0 has a skew of 0). Both plots share the same x-axis

GPU implementations over CPU implementations of WAH, and a related compression scheme, PLWAH (Position List Word-Aligned Hybrid) [6]. Nguyen, et al. [43] showed that field-programmable gate arrays (FPGAs) could be used to create bitmap indices using significantly less power than CPUs or GPUs. These works did not explore querying algorithms. Haas et al. [44] created a custom instruction set extensions for the processing of compressed bitmaps. Their study showed that integrating the extended instruction set in a RISC style processor could realize more than 1.3× speedup over an Intel i7-3960X when executing WAH AND queries. Their study did not investigate parallel solutions.

Other works have developed systems that use GPUs to answer range queries using non-bitmap based approaches. Heimel and Markl [45] integrated a GPU-accelerated estimator into the optimizer of PostgreSQL. Their experiments showed that their approach could achieve a speedup of approximately 10× when compared to a CPU implementation. Gosink et al. [46] created a parallel indexing data structure that uses bin-based data clusters. They showed that their system could achieve 3× speedup over their CPU implementation. Kim et al. [47] showed that their massively parallel approach to R-tree traversal outperformed the traditional recursive R-tree traversals when answering multi-dimensional range queries. Our work focuses on increasing the efficiency of systems relying on WAH compressed bitmaps.

Orthogonal to our parallel approach, other works have investigated non-parallel methods to increase the efficiency of range query processing using bitmap indices. Wu et al. [48] used a size ordered priority queue to

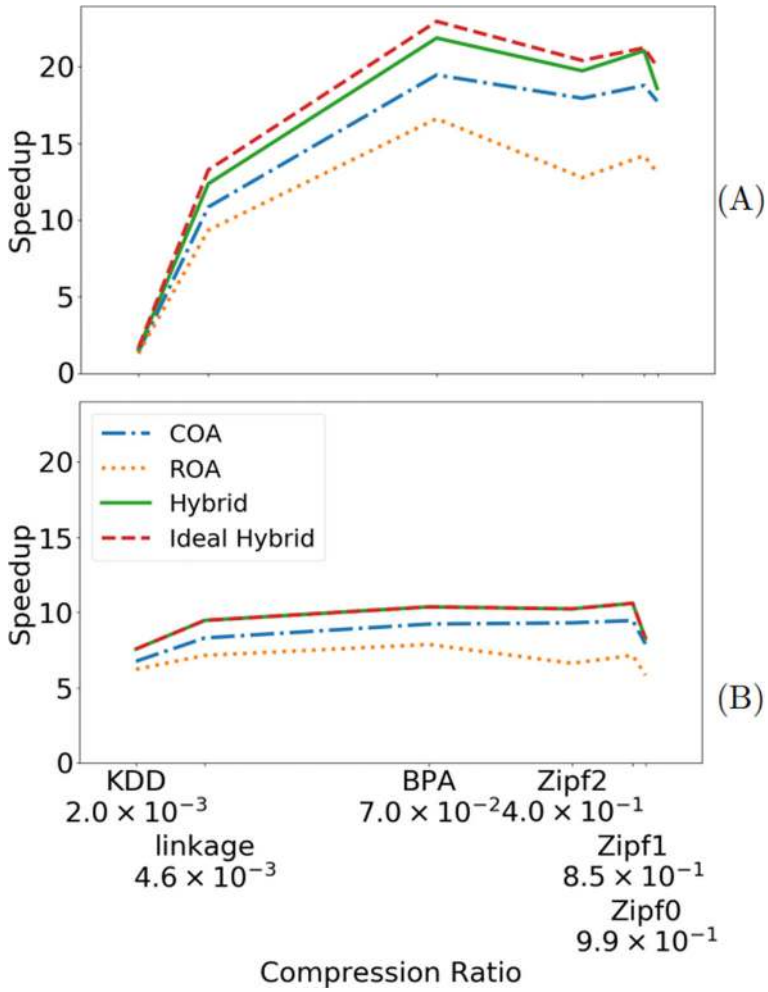

Fig. 14 Average GPU speedup **a**) relative to the iterative CPU method and B) relative to the reduction CPU method vs. data set compression ratio (compressed size / uncompressed size, shown below each data set on the x-axis). The data for hybrid and ideal hybrid in (**b**) are indistinguishable at the scale of this figure. The Zipf data set skews are appended (e.g., Zipf0 has a skew of 0). The horizontal axis is logarithmically scaled. Both plots share the same legend and x-axis

sequence the column processing of WAH and BBC compressed bitmap range queries. Their empirical study showed that this approach requires less core memory and often performed better than a random sequence of column processing. Additionally, they explored an in-place query algorithm, in which the largest column was decompressed and used to start all intermediate results. This approach was shown to be faster than the priority queue algorithm but required more memory. Slechta, et al. [49] explored several similar column ordering techniques for the range query processing of Variable-Aligned Length [27] compressed bitmaps. Chmiel et al. [50] proposed a hierarchically organized bitmap index (HOBI) for dimensional models used in data warehouses. HOBI creates a bitmap index for each dimensional level. An upper level bitmap is essentially the aggregation of the lower level bitmaps. Their experiments showed that the hierarchical structure of HOBI was able to outperform Oracle's native bitmap join index. Similarly, Nagarkar et al. [51] proposed a compressed spatial hierarchical bitmap (cSHB) index to support spatial range queries. Their approach converts a 2D space into a 1D space using Z-order traversal. A hierarchy is then imposed over the 2D space where each node of the hierarchy corresponds to a bounded subspace. Their experimental study showed that cSHB and their bitmap selection process performed better than alternative indexing structures for spatial range queries.

The works of Andrzejewski and Wrembel [11, 12] are closest to the work presented in this paper. They introduced WAH and PLWAH compression and decompression algorithms for GPUs. Their decompression work details a parallel algorithm for decompressing a single

Table 2 Advantages, disadvantages, and ideal query application for the GPU methods

Method	Advantages		Disadvantages		Ideal queries
	Memory coalescing	Shared memory	Extra global memory accesses	Limited throughput	
COA	✓		✓		Point
ROA		✓		✓	Range, where 2048 ≥ columns >1024
Hybrid	✓	✓	✓		Range, where columns >2048
Ideal hybrid	✓	✓			Range, where columns ≤ 1024

WAH compressed bit vector. Our previous work [13] and this extension builds upon their approach so that multiple WAH compressed bit vectors can be decompressed in parallel. Their work also examined parallel queries that were limited to bitwise operations between two bit vectors. While executing bitwise operations between two decompressed bit vectors is obvious, Andrzejewski and Wrembel presented a parallel GPU algorithm for such an operation between two compressed bit vectors [12]. We explored range queries which require bitwise operations to be performed on sequences of bit vectors. As demonstrated above, range queries provide an excellent application for exploiting the highly parallel nature of GPUs.

It should be noted that in [12], Andrzejewski and Wrembel presented a comparison between their GPU-WAH and GPU-PLWAH implementations. Their results showed that GPU-PLWAH was slower due to an additional step needed to decompress PLWAH data. However, after the data was decompressed, their query algorithms were essentially the same. The use of our algorithms (COA, ROA, Hybrid, and Ideal Hybrid) would produce a similar result. Our approaches take decompressed columns as input, so the only timing difference between WAH and PLWAH would be due to the varying decompression algorithms.

As mentioned above, this work is an extension of our previous work [13]. In this paper, we more thoroughly presented all of our algorithms, including a formal presentation of Ideal Hybrid and a theoretical analysis of all our approaches. We also introduced two novel parallel algorithms for processing bitmap range queries on the CPU. Additionally, we significantly expanded our empirical study.

Conclusion and future work

In this paper, we present parallel methods for executing range queries on CPUs and GPUs. The CPU methods comprise iterative and reduction based approaches. All GPU methods perform a reduction across the queried bitmaps. To extract parallelism, the CPU and COA GPU methods operate primarily along paired WAH bit vectors, the ROA GPU method operates along rows (all of which are independent), and the hybrid GPU methods operate along multiple rows at once. The GPU methods exploit the highly parallel nature of GPUs and their architectural details to extract additional performance. These include mechanisms to accelerate memory transfers (coalescing) and the use of low-latency GPU shared memory.

We conducted an empirical study comparing the GPU methods to the CPU methods. Of the two CPU methods, the reduction approach provided greater performance when querying five of the six data sets. It realized an average speedup of 2.18× over the iterative CPU approach. The results of our study showed that the GPU methods outperform the CPU in 98.8% of our tests, providing a maximum speedup of 54.1× and an average speedup of 11.5×. When compared to the highest performing iterative CPU tests, the GPU methods provide an average speedup of 5.69× for queries of 64 bins and 6.44× for queries of 4, 8, 16, 32, and 64 bins. When compared to the highest performing CPU reduction tests, the GPU methods provide an average speedup of 2.16× for queries of 64 bins and 3.06× for queries of 4, 8, 16, 32, and 64 bins.

We plan to pursue additional work analyzing the parameter space of the hybrid method and subsequent performance. This includes the effect of database characteristics, varying tile dimension, and distributing tiles and queries across multiple GPUs. We also intend to continue exploring means to accelerate bitmap query execution using computational accelerators. In particular, we plan to use non-NVIDIA GPUs and future generations of NVIDIA GPUs to investigate additional means of enhancing bitmap query throughput. We would also like to explore the feasibility of refactoring other bitmap schemes such as roaring bitmaps [52] and byte-aligned bitmap codes [4] to run on GPUs.

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; BBC: Byte-aligned bitmap compression; PLWAH: Position list word-aligned hybrid;

FPGA: Field programmable gate array; RISC: Reduced instruction set computer; HOBI: Hierarchically organized bitmap index; cSHB: Compressed spatial hierarchical bitmap; GB: Gigabyte; RAM: Random access memory; KDD: Knowledge discovery and data mining; BPA: Bonneville power administration

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. We would also like to thank our anonymous reviewers for their constructive feedback and positive comments contributing to the improvement of this manuscript.

Author's contributions
Algorithmic design was primarily done by JMM, JS, and DC with secondary contributions by MN and ZS. The implementation of all algorithms was primarily performed by MN and ZS with secondary development by JMM and JS. All parties played an equal role in manuscript development. The author(s) read and approved the final manuscript.

Funding
MN and ZS received Undergraduate Research Funding from the University of St. Thomas.

Availability of data and materials
Please contact the corresponding author to obtain the data used in this manuscript.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Department of Computer and Information Sciences, University of St. Thomas, 2115 Summit Ave., 55105 Saint Paul, Minnesota, USA. [2]Department of Mathematics and Computer Science, University of Puget Sound, 1500 N. Warner St., 98416 Tacoma, Washington, USA.

Received: 25 March 2020 Accepted: 14 July 2020
Published online: 05 August 2020

References

1. Norris RP (2010) Data challenges for next-generation radio telescopes. In: Proceedings of the 2010 Sixth IEEE International Conference on e-Science Workshops. E-SCIENCEW '10. IEEE. pp 21–24. https://doi.org/10.1109/esciencew.2010.13
2. Stockinger K (2001) Design and implementation of bitmap indices for scientific data. In: International Database Engineering and Application Symposium. pp 47–57. https://doi.org/10.1109/ideas.2001.938070
3. Kesheng W, Koegler W, Chen J, Shoshani A (2003) Using bitmap index for interactive exploration of large datasets. In: International Conference on Scientific and Statistical Database Management. pp 65–74. https://doi.org/10.1109/ssdm.2003.1214955
4. Antoshenkov G (1995) Byte-aligned bitmap compression. In: Proceedings DCC'95 Data Compression Conference. IEEE. p 476. https://doi.org/10.1109/dcc.1995.515586
5. Corrales F, Chiu D, Sawin J (2011) Variable length compression for bitmap indices. In: Hameurlain A, Liddle SW, Schewe K-D, Zhou X (eds). Database and Expert Systems Applications. Springer, Berlin. pp 381–395
6. 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. https://doi.org/10.1145/1739041.1739071
7. 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
8. Wu K, Otoo EJ, Shoshani A, Nordberg H (2001) Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory
9. 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. IEEE. pp 99–108. https://doi.org/10.1109/ssdm.2002.1029710
10. Wu K, Otoo EJ, Shoshani A (2006) Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. 31(1):1–38
11. 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
12. Andrzejewski W, Wrembel R (2011) GPU-PLWAH: GPU-based implementation of the PLWAH algorithm for compressing bitmaps. Control Cybern 40:627–650
13. 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, Applications and Technologies. BDCAT '19. Association for Computing Machinery, New York. pp 11–20
14. CUDA C (2019) Best practice guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide. Accessed 1 Mar 2020
15. Djenouri Y, Bendjoudi A, Mehdi M, Nouali-Taboudjemat N, Habbas Z (2015) Gpu-based bees swarm optimization for association rules mining. J Supercomput 71(4):1318–1344
16. Djenouri Y, Bendjoudi A, Habbas Z, Mehdi M, Djenouri D (2017) Reducing thread divergence in gpu-based bees swarm optimization applied to association rule mining. Concurr Comput Pract Experience 29(9):3836
17. Tran N-P, Lee M, Choi DH (2015) Memory-efficient parallelization of 3D lattice Boltzmann flow solver on a GPU. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC). IEEE. pp 315–324. https://doi.org/10.1109/hipc.2015.49
18. Weber N, Goesele M (2017) MATOG: array layout auto-tuning for CUDA. ACM Trans Archit Code Optim (TACO) 14(3):28
19. Dagum L, Menon R (1998) OpenMP: An industry-standard API for shared-memory programming. Comput Sci Eng 5(1):46–55
20. Lichman M (2013) UCI Machine Learning Repository. http://archive.ics.uci.edu/ml. Accessed 1 Aug 2019
21. Lloyd S (1982) Least squares quantization in pcm. IEEE Trans Inf Theory 28(2):129–137
22. Sariyar M, Borg A, Pommerening K (2011) Controlling false match rates in record linkage using extreme value theory. J Biomed Inf 44(4):648–654
23. Bonneville Power Administration, http://www.bpa.gov
24. Newman M (2005) Power laws, pareto distributions and zipf's law. Contemp Phys 46(5):323–351
25. Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P, et al (2010) Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ACM SIGARCH Comput Archit News 38(3):451–460
26. Colantonio A, Di Pietro R (2010) Concise: Compressed 'n' composable integer set. Inf Process Lett 110(16):644–650
27. 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. IEEE. pp 484–495. https://doi.org/10.1109/icde.2014.6816675
28. van Schaik SJ, de Moor O (2011) A memory efficient reachability data structure through bit vector compression. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. SIGMOD '11. pp 913–924. https://doi.org/10.1145/1989323.1989419
29. Chou J, Howison M, Austin B, Wu K, Qiang J, Bethel EW, Shoshani A, Rübel O, Prabhat Ryne RD (2011) Parallel index and query for large scale data analysis. In: International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11. pp 30–13011. https://doi.org/10.1145/2063384.2063424
30. Dong B, Byna S, Wu K (2014) Parallel query evaluation as a scientific data service. In: 2014 IEEE International Conference on Cluster Computing (CLUSTER). pp 194–202. https://doi.org/10.1109/cluster.2014.6968765
31. Yildiz B, Wu K, Byna S, Shoshani A (2019) Parallel membership queries on very large scientific data sets using bitmap indexes. Concurr Comput Pract Experience:5157. https://doi.org/10.1002/cpe.5157
32. Su Y, Agrawal G, Woodring J (2012) Indexing and parallel query processing support for visualizing climate datasets. In: 2012 41st International Conference on Parallel Processing. IEEE. pp 249–258. https://doi.org/10.1109/icpp.2012.33
33. Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Tomov S, Yamazaki I (2014) Accelerating numerical dense linear algebra calculations with GPUs. Numer Comput GPUs:1–26. https://doi.org/10.1007/978-3-319-06548-9_1

34. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36(5-6):232–240
35. Tomov S, Nath R, Ltaief H, Dongarra J (2010) Dense linear algebra solvers for multicore with GPU accelerators. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). IEEE. pp 1–8. https://doi.org/10.1109/ipdpsw.2010.5470941
36. Bell N, Hoberock J (2012) Thrust: A productivity-oriented library for CUDA. In: GPU Computing Gems Jade Edition. pp 359–371. https://doi.org/10.1016/b978-0-12-811986-0.00033-9
37. Merrill D (2016) Cub: Cuda unbound. http://nvlabs.github.io/cub. Accessed 1 Aug 2019
38. Bailey P, Myre J, Walsh SD, Lilja DJ, Saar MO (2009) Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In: 2009 International Conference on Parallel Processing. IEEE. pp 550–557. https://doi.org/10.1109/icpp.2009.38
39. Myre J, Walsh SD, Lilja D, Saar MO (2011) Performance analysis of single-phase, multiphase, and multicomponent lattice-Boltzmann fluid flow simulations on GPU clusters. Concurr Comput Pract Experience 23(4):332–350
40. Walsh SD, Saar MO, Bailey P, Lilja DJ (2009) Accelerating geoscience and engineering system simulations on graphics hardware. Comput Geosci 35(12):2353–2364
41. 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. GPGPU-3. ACM, New York. pp 94–103
42. 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. https://doi.org/10.1145/2504730.2504756
43. Nguyen X, Hoang T, Nguyen H, Inoue K, Pham C (2018) An FPGA-based hardware accelerator for energy-efficient bitmap index creation. IEEE Access 6:16046–16059
44. Haas S, Karnagel T, Arnold O, Laux E, Schlegel B, Fettweis G, Lehner W (2016) Hw/sw-database-codesign for compressed bitmap index processing. In: 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP). pp 50–57. https://doi.org/10.1109/asap.2016.7760772
45. Heimel M, Markl V (2012) A first step towards gpu-assisted query optimization. In: Bordawekar R, Lang CA (eds). International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS. VLDB endowment. pp 33–44
46. 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. https://doi.org/10.1007/978-3-642-02279-1_9
47. Kim J, Kim S-G, Nam B (2013) Parallel multi-dimensional range query processing with r-trees on gpu. J Parallel Distrib Comput 73(8):1195–1207
48. Wu K, Otoo E, Shoshani A (2004) On the performance of bitmap indices for high cardinality attributes. In: VLDB'04. pp 24–35. https://doi.org/10.1016/b978-012088469-8.50006-1
49. Slechta R, Sawin J, McCamish B, Chiu D, Canahuate G (2014) Optimizing query execution for variable-aligned length compression of bitmap indices. In: International Database Engineering & Applications Symposium. pp 217–226. https://doi.org/10.1145/2628194.2628252
50. Chmiel J, Morzy T, Wrembel R (2009) Hobi: Hierarchically organized bitmap index for indexing dimensional data. In: Data Warehousing and Knowledge Discovery. pp 87–98. https://doi.org/10.1007/978-3-642-03730-6_8
51. Nagarkar P, Candan KS, Bhat A (2015) Compressed spatial hierarchical bitmap (cshb) indexes for efficiently processing spatial range query workloads. Proc VLDB Endow 8(12):1382–1393
52. Chambi S, Lemire D, Kaser O, Godin R (2016) Better bitmap performance with roaring bitmaps. Softw Pract Exper 46(5):709–719

Publisher's Note