GPU Acceleration of Range Queries over Large Data Sets

Mitchell Nelson
Zachary Sorenson
Computer and Information Sciences
University of St. Thomas
St. Paul, MN, USA

Joseph M. Myre
Jason Sawin
Computer and Information Sciences
University of St. Thomas
St. Paul, MN, USA

David Chiu
Mathematics and Computer Science
University of Puget Sound
Tacoma, WA, USA

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 three GPU algorithms and one CPU-based algorithm for the parallel execution of bitmap-range queries. We show that in 95% of our tests, using real and synthetic data, the GPU algorithms greatly outperform the parallel CPU algorithm. For these tests, the GPU algorithms provide up to 87.7× speedup and an average speedup of 30.22× over the parallel CPU algorithm. In addition to enhancing performance, augmenting traditional bitmap query systems with GPUs to offload bitmap query processing allows the CPU to process other requests.

ACM Reference Format:
Mitchell Nelson, Zachary Sorenson, Joseph M. Myre, Jason Sawin, and David Chiu. 2019. GPU Acceleration of Range Queries over Large Data Sets. In *Proceedings of the IEEE/ACM 6th International Conference on Big Data Computing, Applications and Technologies (BDCAT '19), December 2–5, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3365109.3368789

1 INTRODUCTION

Modern applications are generating a staggering amount of data. For example, the Square Kilometre Array (SKA) Pathfinders are a collection of radio telescopes can generate 70 PB per year [30]. 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. 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 that 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.

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

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 represent $[0, 100)$, 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 0. The **Quantity** of t_1 is 548, 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 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. [3, 11, 14, 17, 41]), and

BDCAT '19, December 2–5, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-7016-5/19/12…$15.00
https://doi.org/10.1145/3365109.3368789

among these, one of the most prominent is the *Word-Aligned Hybrid* code (WAH) [39]. It has been shown that WAH can compress a bitmap to just a small fraction of its original size [40].

One major benefit of bitmap indices is that they can be queried directly, greatly reducing the number of tuples that must be scanned 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.

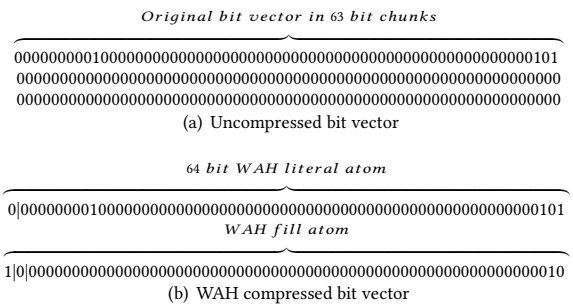
Notice that the above *range* query example could easily be executed in parallel. For example, one process could be execute $q_1 \vee q_3 = r_1$, another could perform $q_2 \vee q_4 = r_2$, and the final result could be computed by $r_1 \vee r_2 = r$. It is clear that the more bins that are needed to 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 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 [1, 2]. Our work explores algorithms that exploit various GPU architectural features to accelerate WAH range-query processing.

The specific contributions of this paper are:

- We present parallel algorithms for executing WAH range queries on multi-core CPUs and GPUs.
- We present refinements to the GPU algorithm that exploit hardware features to improve performance.
- We present an empirical study on both real-world and synthetic data. Our results show that the GPU algorithms are capable of outperforming the CPU algorithm by up to 87.7× and by 30.22× on average. When compared to only the best performing CPU tests, the GPU algorithms still provide 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 Section 2. We describe our parallel WAH query algorithms in Section 3. Our experimental methodology is presented in Section 4. Section 5 presents the results of our empirical study with discussion in Section 6. We describe related works in Section 7 before presenting conclusions and plans for future work in Section 8.



(a) Uncompressed bit vector

(b) WAH compressed bit vector

Figure 1: WAH Compression

2 BACKGROUND

2.1 Word-Aligned Hybrid Compression (WAH)

WAH compresses the bitmap bins (bit vectors) individually. During compression, WAH chunks a bit vector into groups of 63 consecutive bits. Figure 1(a) 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 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 $(flag, value, runLen)$. 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 $runLen$, which records the number of homogeneous chunks that have been clustered together. In Figure 1(a), 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 $runLen$ is set to two since the group is made of two chunks.

One of the advantages is that WAH-compressed bit vectors can be queried directly without decompression. For example, let X and Y be compressed bit vectors and $X \circ Y = Z$ where $\circ$ is a bitwise logical operation, and Z is the resulting bit vector. The query algorithm 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
$Z.result.runLen = Min(X.a_i.runLen, Y.a_j.runLen)$

Processing fills produce side effects for the operand atoms. Specifically, $a_i.runLen = a_i.runLen - result.runLen$ and $a_j.runLen = a_j.runLen - result.runLen$ This will exhaust the atom with the shorter *runLen* 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 $X.a_i.runLen = X.a_i.runLen - 1$.

Note that this approach of processing atom pairs ensures tuple alignment.

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 $2 \leq j \leq n$.

2.2 Graphics Processing Units (GPU)

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 that naturally map to many computational problems. Hierarchically, these structures comprise thread grids, thread blocks, and threads as shown in Figure 2. Threads are executed in groups of 32, *ergo*, thread blocks are typically composed of $32m$ threads where $m \geq 1$.

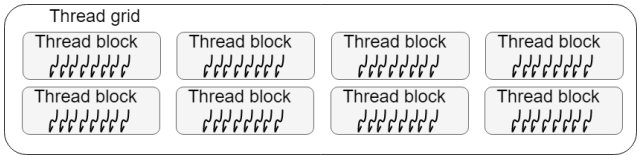


Figure 2: The organizational hierarchy used by NVIDIA's CUDA to structure groups of threads. Shown is a 2×4 2-dimensional thread grid of thread blocks, where each 1-dimensional thread block is composed of 8 threads along the x-dimension.

Alongside the thread hierarchy, The NVIDIA GPUs memory hierarchy conforms 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× less than global memory latency) [12]. Each thread has its own private local memory. To fully realize high-bandwidth transfers from global memory, it is critical to *coalesce* global-memory accesses. For an access to be coalesced, it must meet two criteria: 1) the set of memory addresses are sequential and 2) the set of 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.

3 PARALLEL RANGE QUERIES

In Section 2.1 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 $O(n)$ time process to a $O(\log n)$ 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}). Thus, in principle, all the bitwise operations performed along a row can be processed in parallel using a reduction technique. 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. Without additional information, it would be impossible to select an atom in the middle of a compressed bit vector and know the rows it represents 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 WAH range queries for GPUs and Multi-Core CPUs.

3.1 GPU Methods

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 [2], 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. One notable implementation difference is that 32-bit words were used in [1, 2], while we use 64-bit words. We also modified their algorithm to exploit data structure reuse.

Algorithm 1 Parallel column decompression

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

The key to this algorithm is the call to *DecompressVector*. This procedure is a slightly modified version of the decompression algorithm presented by Andrzejewski and Wrembel [1]. For brevity, we do not present the pseudocode for *DecompressVector*. At a high

level, it uses several *exclusive scans* (parallel element summations) to create an indexing structure of the compressed vector. The procedure then uses this structure to assign a single thread to build each word in the decompressed column. Figure 3 illustrates an example of this thread alignment, in which C is a WAH compressed bit vector composed of three literal atoms (L0, L1, and L2) and two fill atoms (the shaded sectors). For each literal, $DecompressVector$ uses a single thread that writes the value portion of the atom to the *Result* bit vector. Fill atoms need as many threads as the number of compressed words the fragment represents. For example, consider the first fill in C, it encodes a run of three words of 0. $DecompressVector$ creates three threads all reading the same compressed word but writing 0 in the three different locations in *Result*. If a run of 1s had been encoded, a value of $0x3FFF\ FFFF\ FFFF\ FFFF$ would have been written instead of 0.

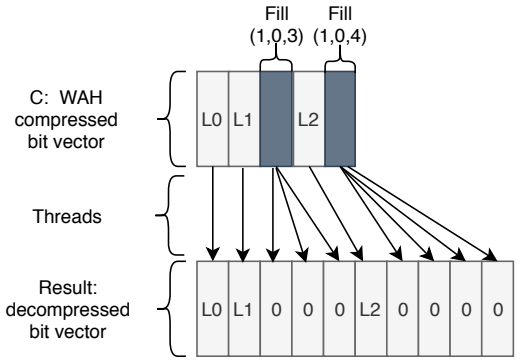


Figure 3: Representation of $DecompressVector$ procedure.

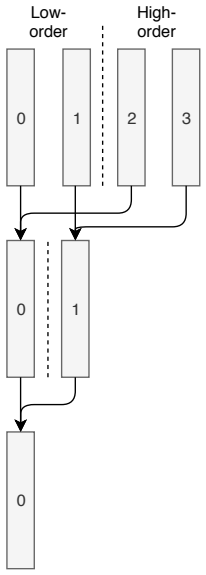
Here we present three different methods for executing range queries in parallel on GPUs. These are column-oriented access (COA), row-oriented access (ROA), and hybrid access approaches. These approaches are analogous to structure-of-arrays, array-of-structures, and hybrid access methods; revolving around how data is organized and accessed to improve global memory coalescing and have been used successfully to accelerate scientific simulations on GPUs [35, 38].

3.1.1 Column-Oriented Access (COA). Our COA approach to range query processing is shown in Algorithm 2. This naïve GPU approach is shown in Figure 4a. The COA procedure takes a collection of decompressed bit vectors needed to answer the query. At each level of the reduction, the bit vectors that require processing are divided into two equal groups: low-order vectors and high-order vectors. The s variable in Algorithm 2 stores the divide position (lines 6 and 15). 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 9 and 10). 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 2, line 12). The process is then repeated using only the low-order vectors as input until a single decompressed bit vector remains. The final bit-vector can be copied back to the CPU.

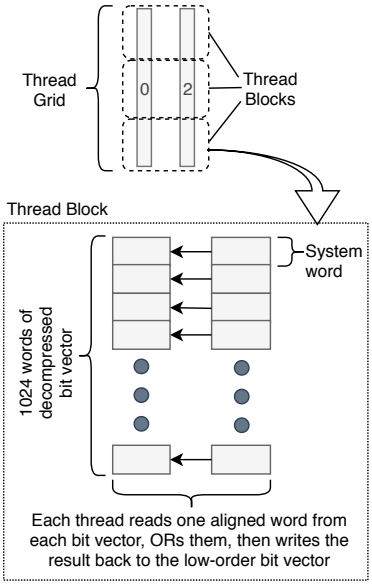
Figure 4a shows this pattern of column accesses for the range query across bit vectors 0 through 3. A 1-dimensional thread grid

Algorithm 2 Column-oriented (COA) query processing

```
1:  procedure COA(Cols)
2:      ▷ Cols is a collection of decompressed bit vectors
3:
4:      m ← |Cols|        ▷ m is the number of bit vectors in the query
5:      n ← |Cols_0|      ▷ n is the number of words in a bit vector
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:             for t ← 0 to n − 1 in parallel do
12:                 c1_t ← c1_t ⋁ c2_t
13:             end for
14:         end for
15:         s ← s/2
16:     end while
17:     return Cols_0
18: end procedure
```



(a) COA Reduction



(b) COA thread grid, block, and work layout

Figure 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.

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 grid 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.

3.1.2 Row-Oriented Access (ROA). Algorithm 3 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 3, line 6). This data access pattern is shown in Figure 5. The figure represents the query $C_0 \bigvee C_1 \bigvee C_2 \bigvee C_3$, 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 2× the maximum number of threads in a thread block.)

Inside any given ROA thread block, the column access pattern within it is identical to the COA pattern (Algorithm 3 line 9-12). 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 Figure 5). The results of the operation are written back to the low order word.

Algorithm 3 Row-oriented access (ROA) query processing

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

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 perform the reduction. Using shared memory for the reduction avoids repeated reads and writes to high-latency global memory (~ 100× 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.

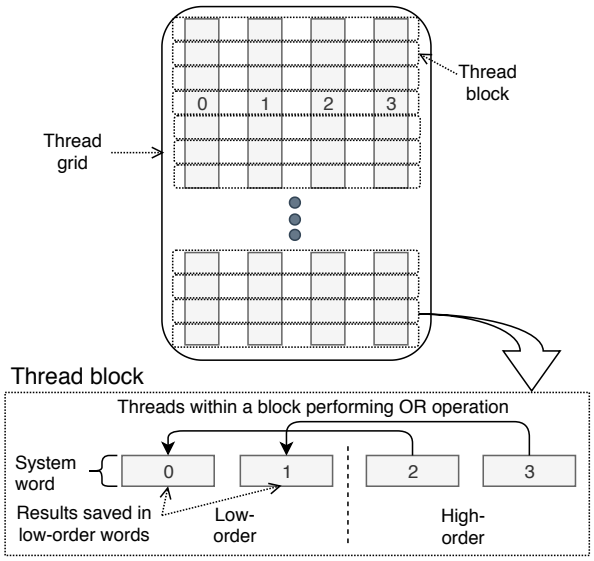


Figure 5: The data access pattern and work performed by each ROA thread block.

3.1.3 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 Figure 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 per thread block.

A single thread block in the hybrid process performs the same work as multiple ROA thread blocks stacked vertically. 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 4 where the first round of reductions are on lines 9-21 and the second round of reductions are on lines 23-35.

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,

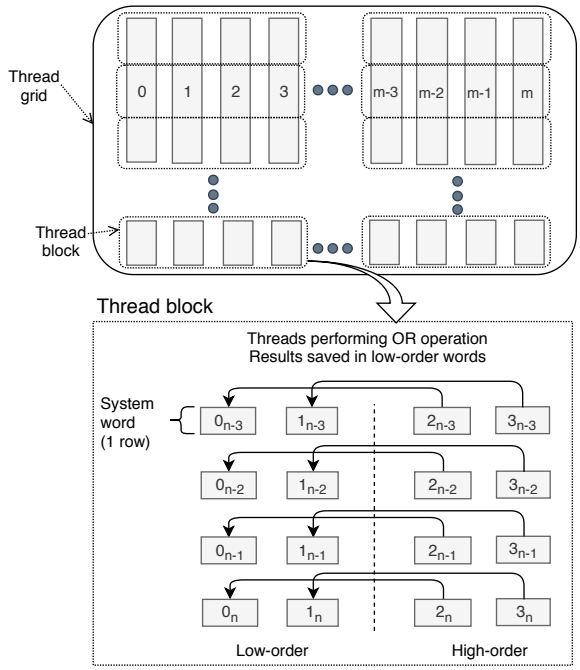


Figure 6: The data access pattern and reduction work performed by each hybrid thread block within tiles (thread blocks). In standard scenarios each thread block would comprise 1024 threads.

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 coalescing can enhance the potential for computational throughput.

The theoretical expressions for global reads and writes imply 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. The ideal layout thereby reduces the 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 $k \times p$ thread block has 1024 threads.

3.2 Multi-Core CPU Method

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 a COA style reduction approach with the difference that compressed columns pairs are processed sequentially (similar to the method outlined at the beginning of this Section.)

Given an *np*-core CPU, this method uses OpenMP [13] to execute up to *np* parallel bitwise operations on paired compressed bit vectors

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

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 Figure 4a.

4 EVALUATION METHODOLOGY

In this section we explain the testing methodology that was used to yield our results. Our tests were executed on a machine running Ubuntu 16.04.5 LTS. 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 Cup'99 and captures network flow traffic. The data set contains 4,898,431 rows and 42 attributes [25]. Continuous attributes were discretized into 25 bins using Lloyd's Algorithm [26], resulting in 475 bins.
- `linkage` – contains anonymized records from the Epidemiological Cancer Registry of the German state of North Rhine-Westphalia [31]. This data set contains 5,749,132 rows and 12 attributes. The 12 attributes were discretized into 130 bins.
- `BPA` – contains measurements reported from 20 synchrophasors deployed over the Pacific Northwest power grid over approximately one month [7]. 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` – a synthetic data set generated using a Zipf distribution. A Zipf distributions represent a clustered approach to discretization. In this process, the density of data is represented in the bitmap, creating a skewed distribution. 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 elements determined by cardinality, k is their rank, and the coefficient $skew$ creates an exponentially skewed distribution. We set $k = 10$, $n = 10$, and $skew = 2$. This generated a data set containing 100 bins (*i.e.,* ten attributes discretized into ten bins each) and 32 million rows.

For each of the four data sets, we use the GPU based range-query methods described in Section 3.1 (*i.e.*, COA, ROA, hybrid, and ideal hybrid) as well as the reference CPU method described in Section 3.2 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.

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

Each experiment was run six times, and the execution time of each 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.

5 RESULTS

Here we present the results of the experiments described in the previous Section, by data set first. We then focus on comparing the highest performing CPU results to the results of the three GPU method described in Section 3.1 and finally on the relative performance of only the GPU methods.

Results for all tests, organized by data set, are shown in Figure 7. CPU range query performance improves with additional cores for every data set.The GPU methods outperform the CPU method in 95.3% of the tests with an average speedup of 30.22×. Further, the GPU methods are capable of providing a maximum speedup of 87.7× over the CPU method. On average, the GPU methods provide 11.45×, 20.23×, 28.96×, and 1.45× speedup for the BPA, linkage, Zipf, and KDD data sets, respectively.

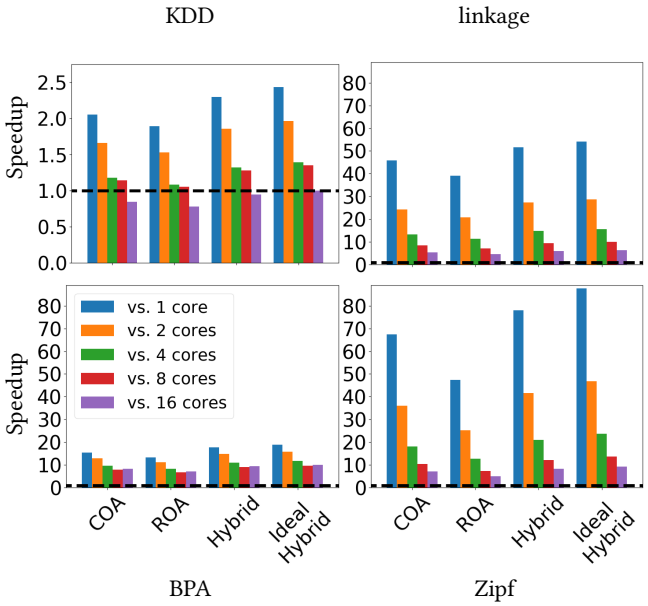


Figure 7: Speedups (vertical axes) for the GPU methods compared to the CPU method (using the number of cores shown in the legend) grouped by the GPU range query method (horizontal axes). The horizontal dashed line indicates a speedup of 1×. All plots share the same legend.

A comparison of the GPU methods to the highest performing CPU (16 core) tests is shown in Figure 8A. On average, the GPU methods provide 5.64× speedup over the CPU when using 16 cores. The KDD data set is the only instance where the GPU methods do unanimously outperform the CPU method using 16 cores. In this scenario, only the ideal hybrid GPU approach provides 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 6.67×.

Speedups provided by the COA, hybrid, and ideal hybrid GPU methods relative to the lowest-performing GPU method (ROA) are shown in Figure 8B. 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 21.08%, 37.67%, and 48.54% speedup over the ROA method, respectively.

We explored the effects varying query sizes had on our algorithms using the CPU method (16 cores) and the ideal hybrid GPU method. These results are shown in Figure 9. GPU execution times are relatively consistent compared to the CPU times which grow at

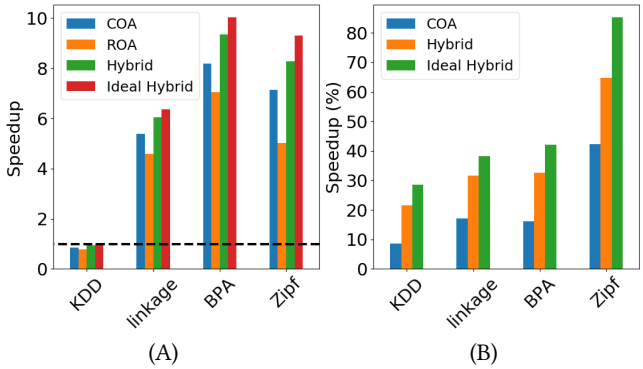


Figure 8: A) Speedups for all GPU methods vs the CPU method using 16 cores. The horizontal dashed line indicates a speedup of 1×. B) Percent speedups for the GPU COA, hybrid, and ideal hybrid methods relative to the ROA method. A speedup of 1× is a percent speedup of zero.

a faster rate. For varied query size, the GPU method outperforms the CPU method by a factor of 6.44×, on average.

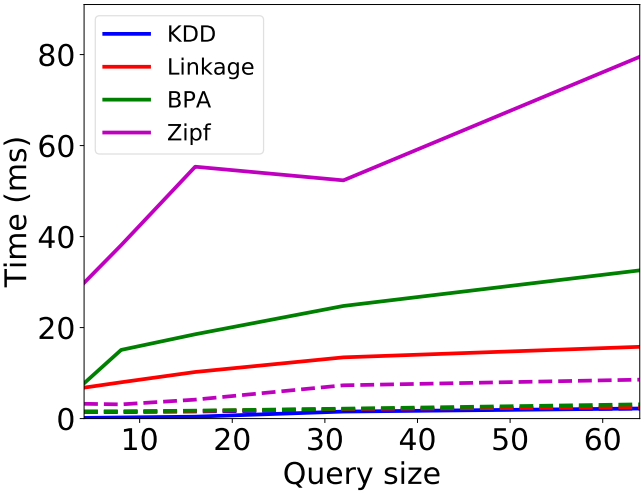


Figure 9: Average execution times for the CPU method using 16 cores (solid lines) and the ideal hybrid GPU method (dashed lines) by query size.

6 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 [24]. The 6.67× speedup of the ideal hybrid method (Section 5) compared to the parallel CPU method using 16 cores aligns well with expectations.

For these tests, a major factor determining GPU performance is the degree of the data set's column compression. This behavior is shown in Figure 10 and 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 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 branch prediction and speculative execution capabilities of the CPU allow enhanced performance over GPUs when querying highly compressed bit vectors as GPUs have no such branch predictors and do not benefit from bitmap compression beyond storage efficiency. The remaining data sets did not exhibit the same degree of compression, thereby reducing CPU performance and enhancing GPU speedup. Further, when data sets are less compressible, there is greater variance in the performance of the GPU methods. This is apparent in the variation in speedup results across the GPU methods in Figure 10, where there is less variation for highly compressible data sets and more variation for less compressible data sets.

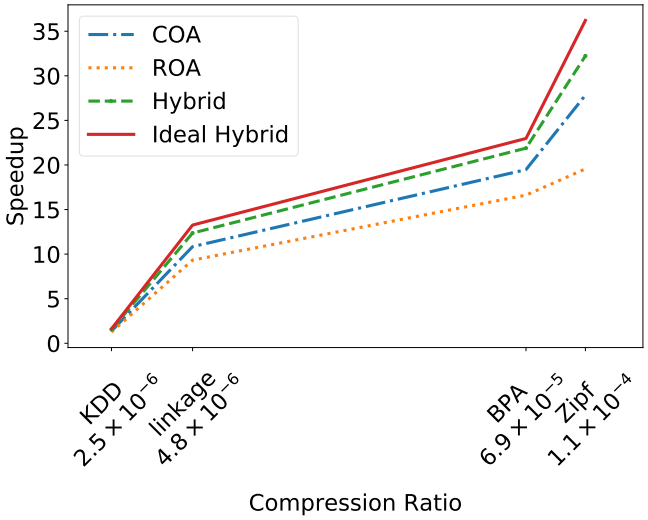


Figure 10: Average GPU speedup vs. data set compression ratio (compressed size / uncompressed size). The horizontal axis is logarithmically scaled.

When varying query size, we find the ideal hybrid GPU method provides a consistent performance enhancement over the parallel CPU method. The relatively consistent results for the GPU in this test 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 one data set where a difference in GPU speed can be observed is Zipf. The variance is likely due to additional contention for memory resources as Zipf has over 4× the number of rows than the next biggest data set

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 test 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 benefits from using shared memory and reduced

Table 2: Advantages, disadvantages, and ideal query application for all 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

computational throughput of each thread block. For larger queries (≤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.

7 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 [39] bitmap compression scheme. However, there are many similar techniques. One of the first hybrid run-length encoding schemes was Byte-aligned Bitmap Compression (BBC) [3]. BBC uses byte-alignment to compress runs and which, in certain cases, allows it to achieve greater compression than other compression schemes [39]. This increase in compression often is achieved at the expense of query times. For this reason many of the recent encoding schemes (e.g., [10, 11, 14, 17, 20, 36, 41]) 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. [9] introduced FastQuery (and several later augmentations, e.g. [15, 42]) 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 [16, 33, 34] or parallel programming primitives [6, 27] 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 [4, 28], finite element methods [37], and traditional relational databases [5].

Several researchers have explored using hardware systems other than standard CPUs for bitmap creation and querying. Fusco, et al. [18] demonstrated that greater throughout of bitmap creation could be achieved using GPU implementations over CPU implementations of WAH, and a related compression scheme, PLWAH [14]. Nguyen, et al. [29] 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. [21] 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 [22] 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. [19] 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. [23] 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.

The works of Andrzejewski and Wrembel [1, 2] 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 a decompressing a single WAH compressed bit vector. Our work builds upon their approach so that n 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 [2]. 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.

8 CONCLUSION AND FUTURE WORK

In this paper, we present parallel methods for executing range queries on CPUs and GPUs. All 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 method operates 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 method. The results of our study showed that the GPU methods outperform the CPU in 95% of our tests, providing a maximum speedup of 87.7× and an average speedup of 30.22×. When compared to the highest performing CPU tests, the GPU methods still provide an average speedup of 5.64× for queries of 64 bins and 6.44× for queries of 4, 8, 16, 32, and 64 bins. Our results also suggest that the GPU WAH range query processing methods benefit from data sets that have low compressibility.

We plan to pursue additional work regarding parameter space of the hybrid method and subsequent performance. This includes the effect of database characteristics, varying tile dimension, and distributing tiles/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 [8] and byte-aligned bitmap codes [3] to run on GPUs.

REFERENCES

[1] Witold Andrzejewski and Robert Wrembel. 2010. GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In *International Conference on Database and Expert Systems Applications*. Springer, Berlin,Heidelberg, 315–329.

[2] Witold Andrzejewski and Robert Wrembel. 2011. GPU-PLWAH: GPU-based implementation of the PLWAH algorithm for compressing bitmaps. *Control and cybernetics* 40 (2011), 627–650.

[3] Gennady Antoshenkov. 1995. Byte-aligned bitmap compression. In *Proceedings DCC'95 Data Compression Conference*. IEEE, 476.

[4] Peter Bailey, Joseph Myre, Stuart DC Walsh, David J Lilja, and Martin O Saar. 2009. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *2009 International Conference on Parallel Processing*. IEEE, 550–557.

[5] Peter Bakkum and Kevin Skadron. 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, NY, USA, 94–103. https://doi.org/10.1145/1735688.1735706

[6] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[7] bpa [n.d.]. Bonneville Power Administration, http://www.bpa.gov.

[8] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better Bitmap Performance with Roaring Bitmaps. *Softw. Pract. Exper.* 46, 5 (May 2016), 709–719.

[9] Jerry Chou, Mark Howison, Brian Austin, Kesheng Wu, Ji Qiang, E. Wes Bethel, Arie Shoshani, Oliver Rübel, Prabhat, and Rob D. Ryne. 2011. Parallel Index and Query for Large Scale Data Analysis. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. 30:1–30:11.

[10] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inform. Process. Lett.* 110, 16 (2010), 644–650.

[11] Fabian Corrales, David Chiu, and Jason Sawin. 2011. Variable Length Compression for Bitmap Indices. In *Database and Expert Systems Applications*, Abdelkader Hameurlain, Stephen W. Liddle, Klaus-Dieter Schewe, and Xiaofang Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–395.

[12] C CUDA. 2019. Best practice guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide

[13] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering* 5, 1 (1998), 46–55.

[14] François Deliège and Torben Bach Pedersen. 2010. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *International Conference on Extending Database Technology (EDBT '10)*. 228–239.

[15] Bin Dong, Surendra Byna, and Kesheng Wu. 2014. Parallel query evaluation as a Scientific Data Service. *2014 IEEE International Conference on Cluster Computing (CLUSTER)* (2014), 194–202.

[16] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2014. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. *Numerical Computations with GPUs* (2014), 1–26.

[17] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. 2010. Net-Fli: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *VLDB* 3, 2 (2010), 1382–1393.

[18] Francesco Fusco, Michail Vlachos, Xenofontas Dimitropoulos, and Luca Deri. 2013. Indexing Million of Packets Per Second Using GPUs. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC '13)*. 327–332.

[19] Luke J. Gosink, Kesheng Wu, E. Wes Bethel, John D. Owens, and Kenneth I. Joy. 2009. Data Parallel Bin-Based Indexing for Answering Queries on Multi-core Architectures. In *Scientific and Statistical Database Management*, Marianne Winslett (Ed.). 110–129.

[20] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 484–495.

[21] S. Haas, T. Karnagel, O. Arnold, E. Laux, B. Schlegel, G. Fettweis, and W. Lehner. 2016. HW/SW-database-codesign for compressed bitmap index processing. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 50–57.

[22] Max Heimel and Volker Markl. 2012. In *Proceedings of VLDB*. 33–44.

[23] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. 2013. Parallel Multi-dimensional Range Query Processing with R-trees on GPU. *J. Parallel Distrib. Comput.* 73, 8 (2013), 1195–1207.

[24] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH computer architecture news* 38, 3 (2010), 451–460.

[25] M. Lichman. 2013. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[26] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[27] Duane Merrill. 2016. Cub: Cuda unbound. *URL: http://nvlabs. github. io/cub* (2016).

[28] Joseph Myre, Stuart DC Walsh, D Lilja, and Martin O Saar. 2011. Performance analysis of single-phase, multiphase, and multicomponent lattice-Boltzmann fluid flow simulations on GPU clusters. *Concurrency and Computation: Practice and Experience* 23, 4 (2011), 332–350.

[29] X. Nguyen, T. Hoang, H. Nguyen, K. Inoue, and C. Pham. 2018. An FPGA-Based Hardware Accelerator for Energy-Efficient Bitmap Index Creation. *IEEE Access* 6 (2018), 16046–16059.

[30] Ray P. Norris. 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, 21–24.

[31] Murat Sariyar, Andreas Borg, and Klaus Pommerening. 2011. Controlling false match rates in record linkage using extreme value theory. *Journal of Biomedical Informatics* 44, 4 (2011), 648–654.

[32] Y. Su, G. Agrawal, and J. Woodring. 2012. Indexing and Parallel Query Processing Support for Visualizing Climate Datasets. In *2012 41st International Conference on Parallel Processing*. IEEE, 249–258.

[33] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36, 5-6 (2010), 232–240.

[34] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. 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, 1–8.

[35] Nhat-Phuong Tran, Myungho Lee, and Dong Hoon Choi. 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, 315–324.

[36] Sebastiaan J. van Schaik and Oege de Moor. 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)*. 913–924.

[37] Stuart DC Walsh, Martin O Saar, Peter Bailey, and David J Lilja. 2009. Accelerating geoscience and engineering system simulations on graphics hardware. *Computers & Geosciences* 35, 12 (2009), 2353–2364.

[38] Nicolas Weber and Michael Goesele. 2017. MATOG: array layout auto-tuning for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 28.

[39] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. 2002. Compressing bitmap indexes for faster search operations. In *Proceedings 14th International Conference on Scientific and Statistical Database Management*. IEEE, 99–108.

[40] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* 31, 1 (2006), 1–38.

[41] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. 2001. *Notes on design and implementation of compressed bit vectors*. Technical Report LBNL/PUB-3161. Lawrence Berkeley National Laboratory.

[42] Beytullah Yildiz, Kesheng Wu, Suren Byna, and Arie Shoshani. 2019. Parallel membership queries on very large scientific data sets using bitmap indexes. *Concurrency and Computation: Practice and Experience* (2019), e5157.