GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid*

Witold Andrzejewski and Robert Wrembel

Poznań University of Technology, Institute of Computing Science, Poznań, Poland
{Witold.Andrzejewski,Robert.Wrembel}@cs.put.poznan.pl

Abstract. Bitmap indexes are one of the basic data structures applied to query optimization in data warehouses. The size of a bitmap index strongly depends on the domain of an indexed attribute, and for wide domains it is too large to be efficiently processed. For this reason, various techniques of compressing bitmap indexes have been proposed. Typically, compressed indexes have to be decompressed before being used by a query optimizer that incurs a CPU overhead and deteriorates the performance of a system. For this reason, we propose to use additional processing power of the GPUs of modern graphics cards for compressing and decompressing bitmap indexes. In this paper we present a modification of the well known WAH compression technique so that it can be executed and parallelized on modern GPUs.

1 Introduction

A data warehouse architecture has been developed for the purpose of integrating data from multiple storage systems within an enterprise. The integrated data are stored in a central database, called a data warehouse. Data stored there are analyzed by OLAP queries, for the purpose of discovering trends, anomalies, hidden dependencies between, and predicting trends. OLAP queries typically access, filter, aggregate large volumes of data. Efficient processing of OLAP queries is often supported by the so-called bitmap indexes [21].

A *bitmap index*, in the simplest form, is composed of the collection of bitmaps (cf. Section 2). A bitmap is a vector of bits. Each bit is mapped to a row in an indexed table. One bitmap B_v is created for one value v of an indexed attribute. If the value of a bit in B_v is equal to 1, then a row corresponding to this bit has value v.

Queries whose predicates involve attributes indexed by bitmap indexes can be answered fast by performing bitwise AND, or OR, or NOT operations on bitmaps, that is a big advantage of bitmap indexes. Unfortunately, the size of a bitmap index increases when the cardinality of an indexed attribute increases. Thus, for attributes of high cardinalities (wide domains) bitmap indexes become very large. As a consequence, they cannot fit in main memory and the

* This work was supported from the Polish Ministry of Science and Higher Education grant No. N N516 365834.

P. García Bringas et al. (Eds.): DEXA 2010, Part II, LNCS 6262, pp. 315–329, 2010.
© Springer-Verlag Berlin Heidelberg 2010

efficiency of accessing data with the support of such indexes deteriorates [29]. In order to improve the efficiency of accessing data with the support of bitmap indexes defined on attributes of high cardinalities various bitmap index compression techniques have been proposed in the research literature (cf. Section 3). Typically, compressed indexes have to be decompressed before being used by a query optimizer that incurs a CPU overhead and deteriorates the performance of a query.

Paper Contribution. In this paper we propose an extension to the well-known Word Aligned Hybrid (WAH) bitmap compression technique that has been reported to provide the shortest query execution time [25,26]. Our extension, called GPU-WAH, allows to parallelize compressing and decompressing steps of WAH and execute them on Graphics Processing Units (GPU). In our implementation (cf. Section 5) we take advantage of the fact, that modern GPUs may process up to 240 threads in parallel, to obtain blazingly fast compression and decompression as well as possible massively parallel comparison of multiple bitmaps. In our experiments we compared the performance of standard WAH run on a CPU with the performance of GPU-WAH. The results show (cf. Section 6) that GPU-WAH significantly reduces compression/decompression time.

2 Definitions

A *bitmap* is a vector of bits. Bitmap literals will be denoted as a string of ones and zeros starting with the most significant bit, and finished with letter "b", e.g., $111000b$. Each bit in a bitmap is assigned a unique, consecutive number starting with 0. The i-th bit of bitmap B is denoted as B_i. The number of bits stored in bitmap B is called a bitmap *length* and it is denoted as $\|B\|$. We define operation *concatenation* of two bitmaps, denoted as $+$, that creates a new bitmap such that it contains all bits of the first bitmap, followed by all bits of the second bitmap. Formally, given bitmaps A and B, their concatenation creates new bitmap C such that: $\|C\| = \|A\| + \|B\| \wedge \forall_{i=0...\|A\|-1} C_i = A_i \wedge \forall_{i=\|A\|...\|C\|-1} C_i = B_{i-\|A\|}$ (e.g., $01b + 10b = 1001b$; this stems from the fact, that the second operand of operator $+$ consists of the bits that will be more significant in the result).

A *subbitmap* of B is any subvector of B that may be created by removing some of the bits from the beginning and from the ending of B. A subbitmap of bitmap B, such that it contains bits from i to j is denoted as $B_{i\to j}$. Formally, for a given bitmap B, subbitmap $C = B_{i\to j}$ must satisfy the condition: $j < \|B\| \wedge \forall_{k=i...j} B_k = C_{k-i}$.

Substitution is an operation that replaces a subbitmap of a given bitmap with another bitmap. Given bitmaps B and C, substituting subbitmap $B_{i\to j}$ with C is denoted as $B_{i\to j} \leftarrow C$, and is formally defined as: $B \leftarrow B_{0\to i-1} + C + B_{j+1\to\|B\|-1}$.

We distinguish two special bitmaps: 1_x and 0_x which are composed of x ones or x zeros respectively. We assume all bitmaps to be divided into 32bit subbitmaps called *words*. Given bitmap B, we denote the i-th word by $B(i)$ (0 based). Formally, $B(i) \equiv B_{i*32\to i*32+31}$. In case, where the length of B is not

the multiplication of 32, we assume the missing trailing bits to be 0. We distinguish several classes of words. Any word whose 31 less significant bits equal 1 is called a *pre-fill full word.* Any word whose 31 less significant bits equal 0 is called a *pre-fill empty word.* Any word D such, that $D_{30 \rightarrow 31} = 10b$, and the rest of the bits encode a number is called a *fill empty word.* Any word D such, that $D_{30 \rightarrow 31} = 11b$, and the rest of the bits encode a number is called a *fill full word.* Any word D such that $D_{31} = 0b$ and the rest of the bits are zeros and ones, is called a *tail word.*

Given any array A of numbers we define operation *exclusive scan* that creates array SA of the same size as A, such that $\forall_{k>0} SA[k] = \sum_{i=0}^{k-1} A[i] \wedge SA[0] = 0$.

A graphics card hardware (GPU, memory) will be called a *device.* The computer hardware (CPU, memory, motherboard), which sends tasks and data to the device, will be called a *host.* A function which is run concurrently in many threads on a device will be called a *kernel.* The subset of data stored in a database will be called a *query.* The process of finding data specified in a query by means of compressed bitmap indexes will be called *query processing.* During a query execution, bitmaps have to be decompressed and processed by bitwise operations.

3 Related Work

Multiple bitmap compression techniques have been proposed in the research literature. Some of them are based on the the run-length encoding, e.g., BBC [2], WAH [25,27,28], PLWAH [7], and some combine the run-length encoding with the Huffman compression, e.g., RL-Huffman [18], RLH [24]. The run-length encoding consists in representing a continuous vector of bits having the same value (either "0" or "1") as: (1) the common value of all bits in the vector and (2) the length of the vector. A bitmap is divided into words before being encoded. Words that include all ones or all zeros are compressed (they are called fills). Words that include intermixed zeros and ones cannot be compressed (they are called tails). Words are organized into *runs* that typically include a fill and a tail.

BBC divides bit vectors into 8-bit words, WAH and PLWAH divide them into 31-bit words, whereas RLH uses words of a parameterized length. PLWAH is the modification of WAH. PLWAH improves compression if tail T that follows fill F differs from F on few bits only. In such a case, the fill word encodes the difference between T and F on some dedicated bits. Moreover, BBC uses four different types of runs, depending on the length of a fill and the structure of a tail. WAH, PLWAH, and RLH use only one type of a run.

The compression techniques proposed in [18] and [24] additionally apply the Huffman compression [16] to the run-length encoded bitmaps. The main differences between [18] and [24] are as follows. First, in [18] only some bits in a bit vector are of interest, the others, called "don't cares" can be replaced either by zeros or ones, depending on the values of neighbor bits. In RLH all bits are of interest and have their exact values. Second, in [18] the lengths of homogeneous subvectors of bits are counted and become the symbols that are encoded by the Huffman compression. RLH uses run-length encoding for representing distances

between bits having value 1. Next, the distances are encoded by the Huffman compression.

Utilizing GPUs in database applications is as of yet not a very well researched field of computer science. Most of the research is being focused on such areas, as: advanced rendering, image and volume processing as well as scientific computations (e.g., numerical algorithms and simulation). The application of GPUs to the compression of images has been presented in [9]. A few papers on increasing the processing power of typical database operations by means GPUs have been proposed so far. They mainly focus on efficient sorting (cf. [11,13,5]), evaluation of query predicates and computing aggregates (cf. [12]), query execution with the support of GPUs and processing indexes (R-trees [17], hash [10], inverted lists [8]). Some approaches to accelerating data mining techniques on GPUs have also been proposed (cf. [3,4,1,23]). None of the aforementioned approaches applies GPUs to compressing and decompressing bitmap indexes.

4 Algorithms

In this section we present three algorithms: an algorithm for extending an input bitmap (denoted as Algorithm 1), an algorithm for compressing an extended input bitmap (Algorithm 2) and an algorithm for decompressing of a compressed bitmap to its extended version (Algorithm 3). We also present several suggestions on query processing scheme using bitmap indexes and a device.

As the first step of compression, bitmaps are extended by Algorithm 1, which appends a single 0 bit after each consecutive 31bit subbitmap. The algorithm starts with appending zeros to the end of input bitmap B, so that its length is a multiplication of 31 (line 1). Next, the number n of 31bit subbitmaps of B is calculated. Once the value n is calculated, it is possible to find the size of the output bitmap E $(32*n)$ and allocate it (line 3). Given E, the algorithm, obtains subbitmaps of B, appends a 0 bit and stores the results in the appropriate words of E. Notice that each operation of storing refers to a different word of the output bitmap, and therefore each word may be computed in parallel.

Algorithm 1. Parallel extension of data

Require: : Input bitmap B
Ensure: : Extended input bitmap E
1: $B \leftarrow B + 0_{31-\|B\| \bmod 31}$
2: $n \leftarrow \|B\|/31$
3: Create bitmap E filled with zeros, such that $\|E\| = 32 * n$
4: **for** $i \leftarrow 0$ to $n-1$ **in parallel do**
5: $\quad E(i) \leftarrow B_{i*31 \rightarrow (i+1)*31-1} + 0_1$
6: **end for**
7: E contains the extended bitmap B

Extended bitmaps are compressed by Algorithm 2. It is composed of five stages. Executions of stages must be sequential, however each of these stages is composed of operations that may be executed in parallel.

The first stage (lines 2–7) determines classes of each of the words in the input bitmap E (whether each of the words is either a tail word or a pre-fill word). The most significant bit in each of the words is utilized to store the word class information. If the word is is a pre-fill word, we store 1 in the most significant bit. If the word is a tail word, we leave it without change as the most significant bit is zero by default (cf. Algorithm 1). To distinguish between a full and empty pre-fill word, one just needs to check the second most significant bit. Let us notice, that tail words already have the final form, consistent with the WAH algorithm. Pre-fill words also have correct two the most significant bits, but they are not yet rolled into a single word and their 30 less significant bits do not encode counters (i.e., they are not yet fill words). This will be achieved in the subsequent stages. Notice, that each word in this stage is processed independently and therefore all of the words may be calculated in parallel.

The second stage (lines 8–15) divides the input bitmap into blocks of words of a single class, where each block will be compressed (in the subsequent stages) into a single fill or tail word. To store the information about ending positions of the aforementioned blocks we use array F. F has the size equal to the number of words in the input bitmap E. The array stores 1 at position i if the corresponding word $E(i)$ in the bitmap is the last word of the block. Otherwise, the array stores 0. Word number i is the last word of the block if is a tail word (the most significant bit is equal to zero), or the word number $i + 1$ a pre-fill word of a different class (the words differ on the two most significant bits). This stage may also be easily parallelized, as each of the positions in array F may be calculated independently.

The third stage (line 16) performs an exclusive scan on array F and stores the result int array SF. The result of this operation is directly tied to storing compression results. As each block found in the previous stage will result in a single word in the compressed bitmap, we know that the algorithm will output as many words, as there are ones in array F. It is easy to notice, that for consecutive indexes i such that $F[i] = 1$, values $SF[i]$ will be consecutive natural numbers starting with zero. Such values, may therefore be used as the output indexes into the output compressed bitmap. Moreover, it is possible to obtain the number of ones in array F by summing the last value stored in array SF with last value in array F (notice, that the last value in F is always equal to 1). Efficient, parallel algorithms for performing the scan operation have been proposed in the research literature (cf. [15,22]).

The fourth stage (lines 18–23) prepares array T of the size equal to the number of words in the output bitmap. For each word $E(i)$, for which value $F[i]$ is equal to 1 (last words of the blocks) the algorithm stores, in array T at the position $SF[i]$, the number of words in all of the blocks up to, and including the considered word. The aforementioned number of words is equal to $i + 1$. Values stored in T are used by the last compression stage for calculating the numbers of words in blocks as well as they allow to retrieve words from the input bitmap E. This stage may be easily parallelized, as all of the writes are independent and may be performed in any order.

The last, fifth stage (lines 24–36) generates the final compressed bitmap. Computations performed in previous stages, allow to compute each word of the compressed bitmap in parallel and independently on the other words. The stage starts with obtaining the preprocessed words from bitmap E, from the positions, where array F stores ones (using the indexes stored in array T). Each of these words is a representative for its block. If the retrieved word is a tail word, it is stored in the output bitmap C in its original state. If the retrieved word is a pre-fill word, two operations are performed. First, the number of words in the corresponding block is calculated using the data stored in array T. The number of words in block i is equal to $T[i] - T[i-1]$, except for $i = 0$ where it is equal to $T[0]$. Second, the calculated number of words is encoded on 30 less significant bits of the retrieved pre-fill word (which creates a new fill word). Regardless of the class of the obtained word, it is stored in the output, i.e., compressed bitmap C. Once these operations are finished, bitmap C contains the compressed result. This stage may be easily parallelized as well, as all of the output words are computed independently.

Let us now analyze Algorithm 3 that implements decompression. It is composed of several stages, each of which must be completed, before the next one is started, however each stage may process input data in parallel.

The first stage (lines 1–9) creates array S of the size equal to the number of words in the compressed bitmap C. For every word $C(i)$ in the compressed bitmap, the algorithm calculates the number of words that should be generated in the output decompressed bitmap, based on the data contained in word $C(i)$, and store the calculated value in array S at position i. This stage is just a prerequisite for the next stage. Notice that this stage may be easily parallelized, as each value of S may be calculated independently.

The second stage (line 10) performs an exclusive scan on array S and stores the result in array SS. The result of this operation is directly tied to storing decompression results. Notice that after exclusive scan, for each word $C(i)$, array SS at position i stores the number of the word in the output decompressed bitmap at which decompression of the considered word should start. Based on the results of the exclusive scan one may also calculate the size of the output decompressed bitmap. This size is equal to the sum of the last values in arrays S and SS.

The third stage (lines 11–15) creates array F, whose size is equal to the number of words in the output decompressed bitmap. The array initially contains only zeros. Next, for each position $SS[i]$ stored in array SS we store 1 in array F at position $SS[i] - 1$. We omit position stored in $SS[0]$ as it is always equal to 0, and there are no entries of negative positions. The aim of this stage is to create an array, where 1 marks the end of the block into which some fill or tail word is extracted. Each assignment in this stage may be executed in parallel, as each assignment targets a different entry in array F.

The fourth stage (line 16) performs an exclusive scan on array F and stores the result in array SF. Once this stage is completed, array SF contains at each

Algorithm 2. Parallel compression of extended data

```
Require: : Extended input bitmap E
Ensure: : Compressed Bitmap C
 1: n ← ||E||/32
 2: Create an array F of size n {0 based indexing}
 3: for i ← 0 to n − 1 in parallel do
 4:    if E(i) = 0_32 or E(i) = 1_31 + 0_1 then
 5:       E(i)_31 ← 1b
 6:    end if
 7: end for
 8: for i ← 0 to n − 2 in parallel do
 9:    if E(i)_30→31 ≠ E(i + 1)_30→31 or E(i)_31 = 0 then
10:       F[i] ← 1
11:    else
12:       F[i] ← 0
13:    end if
14: end for
15: F[n − 1] ← 1
16: SF ← exclusive scan on the array F
17: m ← F[n − 1] + SF[n − 1] {m is the number of words in the compresed bitmap}
18: Create an array T of size m {0 based indexing}
19: for i ← 0 to n − 1 in parallel do
20:    if F[i] = 1 then
21:       T[SF[i]] ← i + 1
22:    end if
23: end for
24: Create a bitmap C such, that ||C|| = m * 32
25: for i ← 0 to m − 1 in parallel do
26:    j ← T[i] − 1
27:    X ← E(j)
28:    if X_31 = 1b then
29:       count ← j + 1
30:       if i ≠ 0 then
31:          count ← count − T[i − 1]
32:       end if
33:       X ←30bit representation of count + X_30→31
34:    end if
35:    C(i) ← X
36: end for
37: C contains the compressed bitmap E
```

position i the number of the word in the input compressed bitmap C, which should be used to generate output word $E(i)$.

Fifth stage (lines 17–29) performs the final decompression. For each word $E(i)$ in the output bitmap, the algorithm performs the following tasks. First, the number of the word in compressed bitmap C which should be used to generate word $E(i)$ is retrieved from array SF, from position i. Second, the word of the retrieved number is read from compressed bitmap C, and based on its type, value $E(i)$ is derived. If the retrieved word is a tail word, it is inserted into $E(i)$ without any further processing. If the retrieved word is a fill word, depending on whether it is an empty or full word, 0_{32} or $1_{31} + 0_1$ is inserted into $E(i)$, respectively. Once the last stage is finished, E contains the decompressed bitmap. As all of the output words are calculated independently, calculation of each word may be run in parallel.

Compressing/decompressing bitmaps using a GPU requires data transfer between the host memory and the device memory. This transfer is done by means of the PCI-Express x16 bus. The transfer is very slow as compared to the internal

Algorithm 3. Parallel decompression of compressed data

Require: : Compressed Bitmap C
Ensure: : Extended input bitmap E
1: $m \leftarrow \|C\|/32$
2: Create an array S of size m
3: **for** $i \leftarrow 0$ to $m-1$ **in parallel do**
4: **if** $C(i)_{31} = 0b$ **then**
5: $S[i] \leftarrow 1$
6: **else**
7: $S[i] \leftarrow$ the value of *count* encoded on bits $C(i)_{0\rightarrow 30}$
8: **end if**
9: **end for**
10: $SS \leftarrow$ exclusive scan on the array S
11: $n \leftarrow SS[m-1] + S[m-1]$ {n contains the number of words in a decompressed bitmap}
12: Create an array F of size n filled with zeroes {0 based indexing}
13: **for** $i \leftarrow 1$ to $m-1$ **in parallel do**
14: $F[SS[i]-1] \leftarrow 1$
15: **end for**
16: $SF \leftarrow$ exclusive scan on the array F
17: Create a bitmap E of length $\|E\| = n * 32$
18: **for** $i \leftarrow 0$ to $n-1$ **in parallel do**
19: $D \leftarrow C(SF[i])$
20: **if** $D_{31} = 0b$ **then**
21: $E(i) \leftarrow D$
22: **else**
23: **if** $D_{30} = 0b$ **then**
24: $E(i) \leftarrow \mathit{0}_{32}$
25: **else**
26: $E(i) \leftarrow \mathit{1}_{31} + \mathit{0}_{1}$
27: **end if**
28: **end if**
29: **end for**
30: E contains a decompressed bitmap C

Algorithm 4. Extension and checking of classes of words

```
1: start=i*31;
2: off=start&31; // %32
3: start>>=5; // /32
4: result=(B[start]>>off)&BM31;
5: result|=(B[start+1]<<(32-off))&BM31;
6: test=(result==0 || result==BM31);
7: result=result | (-test & BMMSB);
```

Algorithm 5. Calculating of the number of words to be extracted from a compressed word

```
1: bool test=(data&BMMSB)!=0;
2: res=(!test)|((data&BM30)&(-test));
```

Algorithm 6. Deriving of the output word based on the compressed word

```
1: bool testCase1=((data&BM2MSB)!=BMMSB);
2: bool testCase2=((data&BM2MSB)==BM2MSB);
3: res=(data&(-testCase1))|(-testCase2);
4: res=res&BM31
```

device memory bandwidth [20]. This problem can be partially eliminated by processing all of the query on the device. There are several benefits of such an approach: (1) there is no need to download decompressed bitmaps from the device, (2) only compressed bitmaps need to be uploaded that are small and the transfer may be performed in parallel with computations performed on the device, (3) the computing power of the device can be used in order to perform bitwise operations. The only task of the host during the query processing should be to initiate data transfers when needed and to start kernels on the device. Other than that, the host is free to do any other tasks. The device should decompress the received bitmaps and perform bitwise operations on them. Once all of the calculations are finished, the final bitmap should be transfered from the device to the host. This last stage is unfortunately very slow as the device→host transfers are the slowest. Moreover, the resulting bitmap is decompressed, and therefore very large. Nonetheless it is beneficial, as we only need to transfer one such bitmap (we would have to download every decompressed bitmap if the query was performed on the host).

5 Implementation

The algorithms presented in the previous section were implemented in C++ and C for CUDA using the NVIDIA CUDA platform [6]. In this section we outline their implementations and focus on the implementation details that allowed us to remove almost all of branching (alternative flows of control, e.g. if-then-else structures) from the kernel code.

In our implementation we used a straightforward storage model for bitmaps. Each bitmap is represented as an array of 32bit unsigned integers. The same representation is used on the device and the host. For the exclusive scan operation on the CUDA platform we use a very efficient implementation, which is a part of the CUDPP library [14].

The code fragments, presented in the following paragraphs, utilize several constants: `BM31`, `BMMSB`, and `BM30`. All of these constants are 32bit unsigned integers and contain values `0x7fffffff`, `0x80000000`, and `0x3fffffff` respectively.

In our implementation we have created two compression procedures, where one of them incorporates the extension algorithm and the other does not. The first of these two implementations integrates the extension algorithm with the first stage of compression. Let us consider the aforementioned, integrated portion of the source code (cf. Algorithm 4). This code performs two operations: it extracts 31 bit subbitmap from the input bitmap and appends either 0 or 1 depending on whether the retrieved 31 bit subbitmap contains only the same bits (a pre-fill word) or both values of bits (a tail word).

The next source code is a fragment of the implementation of the decompression algorithm (cf. Algorithm 5). This fragment calculates the number of words that should be generated from the given fill or tail word stored in variable `data`, and it roughly corresponds to the lines 4–8 of the algorithm 3. It does not require any if-the-else structures.

The last fragment of the source code (cf. Algorithm 6) also represents the implementation part of the decompression algorithm. This code derives, based on the given fill or tail word (stored in `data`), a word that should be stored into the output decompressed bitmap, and it roughly corresponds to the lines 20–28 of the algorithm 3. It does not require any if-the-else structures as well.

6 Experiments

Experiments were performed on an Core i7 2.8GHz CPU and NVIDIA Geforce 285 GTX graphics card. Their aim was to measure:

- compression, decompression, and recompression (applied to an extended bitmap, cf. Algorithm 1) time using CPU,
- compression, decompression, and recompression time using GPU,
- time of uploading the input bitmap to the graphics cards memory (for each type of operation separately),
- time of downloading the output bitmap from the graphics cards memory (for each type of operation separately).

Each of the tested input bitmaps was composed of $96 * 10^7$ bits. In the experiments we used bitmaps with their densities varying from 0.5 to 1/65536 ($1/2^i$ where $i = 1, 2, \ldots, 16$). The bits whose values were set to 1 were randomly selected. We generated 10 instances of each of the bitmaps for each experiment. The execution times discussed below represents averages of 10 experiments.

The results of the experiments for compression and recompression are presented in Figures 1 and 2, respectively. Both of these figures are very similar, as essentially both of them present results from measuring the same algorithm. The only difference is that the compression includes the extension algorithm and the recompression does not. While comparing these two charts one may notice, that the extension algorithm requires about 70ms on host for the tested bitmaps, and indeed the difference between compression and recompression time is about 57ms in the best case and 115ms and the worst case. The same difference on the device is much smaller: 0.85ms in the best case and 1,12ms in the worst case.

While analyzing Figures 1 and 2 one may also notice that the compression is about 16–24 times faster on the device, than it is on the host. Unfortunately, if we include the transfer times, the difference in speed is reduced to 3.6–5.8 times faster than host. The similar numbers calculated for recompression are 12.4–21 times faster on device without data transfers and 2.6–4 times faster on device with data transfers. One may also notice, that the device→host transfer times monotonically depend on the bitmap density as the less dense the bitmap is, the smaller the compression (recompression) result.

Let us now consider Figure 3 that presents comparison of the decompression times on the device and the host. The decompression on the device is 4.75–10 times faster than on the host. Unfortunately, after decompression, one must transfer the large decompressed bitmap from the device to the host memory over the slow PCI-Express x16 bus. Moreover, graphics cards are designed to optimize the host→device transfers rather then back. This results in large transfer

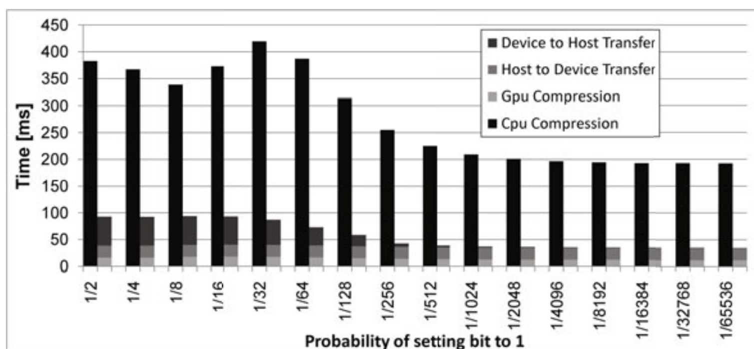

Fig. 1. Compression and data transfer times for bitmaps of varying density

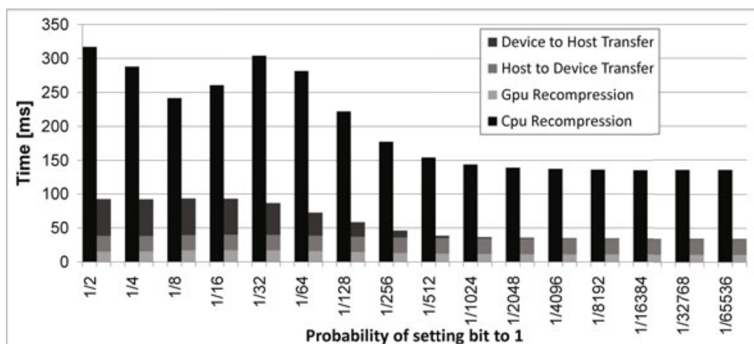

Fig. 2. Recompression and data transfer times for bitmaps of varying density

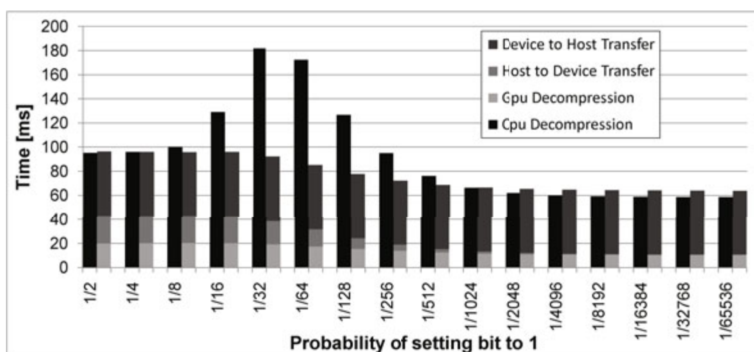

Fig. 3. Decompression and data transfer times for bitmaps of varying density

times that consume all of the benefit from performing the decompression on the device. From Figure 3 we can observe that the whole process of transferring and decompressing a bitmap may be even slower on the device, than it is on the host. We may also notice, that host→device transfer times depend monotonically on the bitmap density, similarly as observed for the device→host transfers during compression/recompression of data. In this case, the same data is sent back to the device in order to be decompressed.

Let us consider the observed problem of large data transfer times between the device and the host. Let us notice, that even when the total decompression time is a bit slower on the device we still benefit from the fact, that the host is free and may in the same time perform other tasks. Moreover, we may also utilize the fact, that data transfers and computations on the device may be performed in parallel. However, as was suggested in the section 4, by performing the whole query on the device, we may be able to accelerate query processing, while still freeing the host from most of the work and removing the need for the costly device→host transfers.

As an example, let us consider an attribute whose domain cardinality is equal to 1024. As shown in Figure 3 total decompression time on the device and host are almost equal. The decompression on the host requires 66.42ms, whereas the decompression on the device requires 66.64ms, where the host→device transfer took 1.64ms, the transfer device→host took 53.46ms and the decompression itself took 11.54ms. To analyze the query performance time we also need the times of performing bitwise operations on bitmaps. The time of performing a bitwise operation on two bitmaps of size $96 * 10^7$ bits on the device consumes about 2.79ms. The same time on the host consumes about 62.67ms. Let us also assume, that the buffer for the compressed bitmaps on the device may contain at most two compressed bitmaps. Let us now consider a hypothetical query which requires 3 bitmaps to be decompressed and processed.

Figure 4 illustrates the hypothetical query processing in the Gannt chart. The query processing starts with the DMA data transfer of the first compressed bitmap $C1$ from the host to the device. Once the transfer is finished, the device may start decompression of the transfered data (the result is stored in bitmap $E1$). Meanwhile, the transfer of the second compressed bitmap $C2$ is started. Once the bitmap $C1$ is decompressed, it may be discarded and the transfer of the last compressed bitmap $C3$ may start. While bitmap $C3$ is transfered, the device may decompress bitmap $C2$ and store the decompression result in $E2$. Once bitmaps $C1$ and $C2$ are decompressed, the device may perform an in-place bitwise operation on their decompressed versions ($E1$ and $E2$), and store the result in $E1$. Bitmap $E2$ may now be discarded, and the decompression of the compressed bitmap $C3$ may start (the result is stored in $E3$). Once bitmap $C3$ is decompressed, the device may perform an in-place bitwise operation between bitmaps $E1$ and $E3$, storing the result in $E1$. $E1$ now stores a decompressed bitmap with the results of the query and may therefore be sent to the host. The whole query processing time, including transfers, takes 95.61ms. Notice, that the query is performed only on the device, and data transfers are performed by

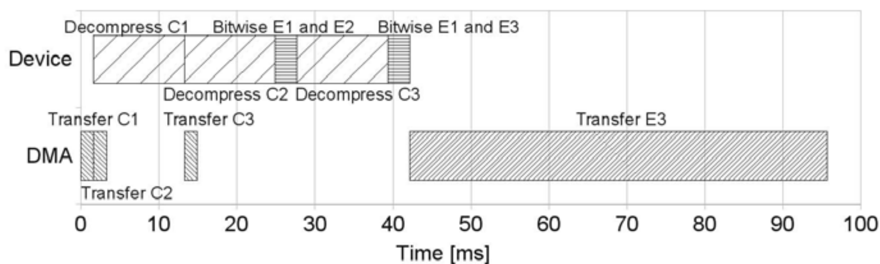

Fig. 4. Gannt chart showing the query processing on the GPU

the DMA. The only work performed by the host is the initialization of DMA transfers and starting of kernels.

Let us now consider the same query, but performed entirely on the host. We need to perform 3 decompressions and 2 bitwise operations on the decompressed bitmaps, which gives the total time equal to 3*66.42ms+2*62.67ms=324.6ms. Notice, that for such a simple case, we have processed a query over 3 times faster on the device, than on the host. The more bitmaps are used, the higher the speedup is. Moreover, while the device processes a query, the host is free to do any other tasks.

We would also like to address the strange, non-monotonic compression and decompression times dependency on the bitmap density. They are probably caused by an increased number of misses in the branch prediction. Notice, that the worst processing time appears for bitmaps of density 1/32 and similar. For such bitmaps, there is a high probability of interleaving of fill and tail words. In our implementation, different parts of code are used for each type of word. We count the words sequentially, and therefore some optimizations presented in the previous sections do not apply here. As there is a high probability that the next generated word will be a different than the previous one, it may cause the branch prediction algorithm to give inaccurate results and lead to the decreased efficiency of data processing.

7 Summary

In this paper we presented an extension, called GPU-WAH, of the WAH compression/decompression technique. GPU-WAH allows to parallelize compressing and decompressing steps and to execute them on the graphics processing units on the CUDA platform. We also discussed the implementation of GPU-WAH and presented its experimental comparison to the standard CPU-based WAH. As the experiments showed, GPU-WAH performs 3.6-5.8 times faster than the CPU-based WAH. We concluded that the decompression is several times faster on a GPU than on the CPU, but the data transfer between GPU memory and computer memory is a bottleneck. Nonetheless, we have presented a query processing scheme which may reduce query processing time by several times.

Future work will focus on: (1) implementing on the CUDA platform other compression techniques including BBC, PLWAH, RLH-n, and comparing their efficiency, (2) implementing query processing technique according to the proposed scheme and test whether its performance is consistent with the theoretical results, (3) applying several optimizations dedicated to given graphics cards computing capabilities, including the upcoming FERMI architecture[19].

References

1. Andrzejewski, W.: Fast K-Medoids Clustering on PCs. In: ADMKD Workshop (2007)
2. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in Oracle RDB. VLDB Journal 5(4), 229–237 (1996)
3. Böhm, C., Noll, R., Plant, C., Wackersreuther, B.: Density-based clustering using graphics processors. In: Proc. of ACM Conference on Information and Knowledge Management (CIKM), pp. 661–670 (2009)
4. Cao, F., Tung, A.K.H., Zhou, A.: Scalable Clustering using graphics processors. In: Yu, J.X., Kitsuregawa, M., Leong, H.-V. (eds.) WAIM 2006. LNCS, vol. 4016, pp. 372–384. Springer, Heidelberg (2006)
5. Chen, S., Zhao, J., Qin, J., Xie, Y., Heng, P.-A.: An efficient sorting algorithm with CUDA. Journal of the Chinese Institute of Engineers 32(7), 915–921 (2009)
6. CUDA. What is CUDA?, http://www.nvidia.com/object/what_is_cuda_new.html
7. Deliège, F.: Concepts and Techniques for Flexible and Effective Music Data Management. PhD thesis, Aalborg University, Denmark (2009)
8. Ding, S., He, J., Yan, H., Suel, T.: Using graphics processors for high-performance ir query processing. In: Proc. of Int. Conf. on World Wide Web, pp. 1213–1214 (2008)
9. Erra, U.: Toward real time fractal image compression using graphics hardware. In: Bebis, G., Boyle, R., Koracin, D., Parvin, B. (eds.) ISVC 2005. LNCS, vol. 3804, pp. 723–728. Springer, Heidelberg (2005)
10. Gosink, L.J., Wu, K., Bethel, E.W., Owens, J.D., Joy, K.I.: Bin-hash indexing: A parallel method for fast query processing. Research report, Lawrence Berkeley National Laboratory (2008)
11. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: high performance graphics co-processor sorting for large database management. In: Proc. of ACM SIGMOD Int. Conf. on Management of Data, pp. 325–336 (2006)
12. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 215–226 (2004)
13. Greß, A., Zachmann, G.: GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), p. 45 (2006)
14. Harris, M., Owens, J.D., Sengupta, S., Tseng, S., Zhang, Y., Davidson, A., Satish, N.: CUDA Data Parallel Primitives Library (CUDPP)
15. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with cuda. In: GPU Gems 3. Addison-Wesley, Reading (2007)
16. Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proc. of the Institute of Radio Engineers, pp. 1098–1101 (1952)

17. Kunjir, M., Manthramurthy, A.: Using graphics processing in spatial indexing algorithms. Research report, Indian Institute of Science, Database Systems Lab. (2009)
18. Nourani, M., Tehranipour, M.H.: Rl-huffman encoding for test compression and power reduction in scan applications. ACM Transactions on Design Automation of Electronic Systems 10(1), 91–115 (2005)
19. NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. White Paper, NVIDIA
20. NVIDIA CUDA Toolkit 2.3. NVIDIA CUDA C Programming Best Practices Guide
21. O'Neil, P., Quass, D.: Improved query performance with variant indexes. In: Proc. of ACM SIGMOD Int. Conference on Management of Data, pp. 38–49 (1997)
22. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Graphics Hardware 2007, pp. 97–106. ACM, New York (2007)
23. Shalom, S.A.A., Dash, M., Minh, T.: Efficient K-Means Clustering Using Accelerated Graphics Processors. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2008. LNCS, vol. 5182, pp. 166–175. Springer, Heidelberg (2008)
24. Stabno, M., Wrembel, R.: RLH: Bitmap compression technique based on run-length and Huffman encoding. Information Systems 34(4-5), 400–414 (2009)
25. Stockinger, K., Wu, K.: Bitmap indices for data warehouses. In: Wrembel, R., Koncilia, C. (eds.) Data Warehouses and OLAP: Concepts, Architectures and Solutions, pp. 157–178. Idea Group Inc., USA (2007) ISBN 1-59904-364-5
26. Wu, K., Otoo, E.J., Shoshani, A.: Compressing bitmap indexes for faster search operations. In: Proc. of Int. Conference on Scientific and Statistical Database Management (SSDBM), pp. 99–108 (2002)
27. Wu, K., Otoo, E.J., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: Proc. of Int. Conference on Very Large Data Bases (VLDB), pp. 24–35 (2004)
28. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Transactions on Database Systems (TODS) 31(1), 1–38 (2006)
29. Wu, M., Buchmann, A.: Encoded bitmap indexing for data warehouses. In: Proc. of Int. Conference on Data Engineering (ICDE), pp. 220–230 (1998)