Control and Cybernetics

vol. **40** (2011) No. 3

GPU-PLWAH: GPU-based implementation of the PLWAH algorithm for compressing bitmaps *†

by

Witold Andrzejewski and Robert Wrembel

Poznan University of Technology, Institute of Computing Science
Pl. Marii Skłodowskiej-Curie 5, Poznań, Poland
e-mail: {Witold.Andrzejewski,Robert.Wrembel}@cs.put.poznan.pl

Abstract: Bitmap indexes are data structures applied to indexing attributes in databases and data warehouses. A drawback of a bitmap index is that its size increases when the domain of an indexed attribute increases. As a consequence, for wide domains, the size of a bitmap index is too large to be efficiently processed. Hence, various techniques of compressing bitmap indexes have been proposed. A compression technique incurs some system overhead (mainly CPU) for compression and decompression operations. For this reason, we propose to use additional processing power of graphical processing units (GPUs). In this paper, we present the GPU-PLWAH algorithm that is a parallel implementation of the recently developed PLWAH compression algorithm. GPU-PLWAH was experimentally compared to its traditional CPU version as well as to our previously developed parallel GPU implementation of the WAH compression algorithm. The experiments show that applying GPUs significantly reduces compression/decompression time.

Keywords: data warehouse, GPGPU, bitmap index, bitmap index compression, PLWAH, WAH.

1. Introduction

One of the fundamental research and technological challenges in databases and data warehouses is assuring the efficiency of the system with respect to query response time. Multiple query optimization algorithms and data structures have been proposed in this area. The latter include various types of tree-based or hash-based indexes (see Lehman and Carey, 1986) or bitmap-based indexes (see O'Neil and Quass, 1997). The performance of queries based on these types of indexes differs for different patterns of queries and for different data characteristics, e.g., cardinalities of indexed attributes or data distribution (see O'Neil,

*Submitted: March 2011; Accepted: August 2011.

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

O'Neil and Wu, 2007; Wu, Otoo and Shoshani, 2004b; Zaker, Phon-Amnuaisuk and Haw, 2008). Typically, tree-based indexes (e.g., B-tree) offer good query performance when defined on attributes of high cardinalities and when used for optimizing queries that retrieve not more than 10-15% of rows. Bitmap indexes offer good query performance when defined on attributes of low cardinalities and for queries that retrieve large numbers of rows. For this reason, bitmap indexes are commonly used for optimizing analytical queries in data warehouses.

A *bitmap index*, in the simplest form, is composed of the so-called bitmaps (see Section 2), each of which is a vector of bits. Each bit is mapped to a row in an indexed table. If the value of a bit equals 1, then a row corresponding to this bit has a certain value of an indexed attribute. Queries, whose predicates involve attributes indexed by bitmap indexes can be answered fast by performing bitwise AND, or OR, or NOT operations on bitmaps, a big advantage of bitmap indexes. The size of a bitmap index strongly depends on the cardinality (domain width) of an indexed attribute, i.e., 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 efficiency of accessing data with the support of such indexes deteriorates (see Wu and Buchmann, 1998).

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 (see Section 3). Typically, compressed indexes have to be decompressed before being used by a query optimizer, which incurs a CPU overhead and deteriorates query performance.

Paper contribution. In this paper we propose an extension of the recently developed bitmap compression technique, called Position List Word Aligned Hybrid (PLWAH) (see Deliège, 2009). Our extension, further called GPU-PLWAH, allows to parallelize compressing and decompressing steps of PLWAH as well as perform bitwise operations on compressed bitmaps and allows to execute them on Graphics Processing Units (GPUs). The GPU-PLWAH implementation was evaluated experimentally. The experiments compared the performance of standard PLWAH run on a CPU with performance of GPU-PLWAH. The results show (see Section 5) that GPU-PLWAH significantly reduces compression/decompression time. Additionally, as a reference we included performance characteristics of the previously developed GPU implementation of the World Aligned Hybrid (WAH) bitmap compression technique, reported to provide the shortest query execution time (see Stockinger and Wu, 2006, and Wu, Otoo and Shoshani, 2002).

2. Definitions

A *bitmap* is a vector of bits. Bitmaps will be denoted as uppercase letters. 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., $111000b$, where ones are more significant than zeros. Each bit in a bitmap is assigned a unique, consecutive number starting with 0. The bigger the bit number, the more significant the corresponding bit is. Bit number i of bitmap B is denoted as B_i. A bitmap *length* is the number of bits stored in the bitmap. Given bitmap B, we denote its length $\|B\|$. We define *concatenation* of two bitmaps, denoted $+$, as an operation 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, concatenation $A + B$ creates new bitmap C such that: $\|C\| = \|A\| + \|B\| \wedge \forall_{i=0\ldots\|A\|-1} C_i = A_i \wedge \forall_{i=\|A\|\ldots\|C\|-1} C_i = B_{i-\|A\|}$.[1]

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 $B_{i\to j}$. Formally, for a given bitmap B, subbitmap $C = B_{i\to j}$ must satisfy the condition: $j < \|B\| \wedge \forall_{k=i\ldots j} B_k = C_{k-i}$.

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 zeroes, 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, $B(i) \equiv B_{i*32\to i*32+31}$. When the length of B is not the multiple 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 $D_{30\to31} = 10b$, and the rest of the bits encode two (5bit and 25bit) numbers is called a *fill empty word*. Any word D such, that $D_{30\to31} = 11b$, and the rest of the bits encode two (5bit and 25bit) numbers 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 numerical values we define operation *exclusive scan* that creates array SA of the same size as A, such that $\forall_{k>0} SA[k] = \sum_{i=0}^{k-1} A[i] \wedge SA[0] = 0$. Similarly, we also define an *inclusive scan* operation. Inclusive scan, given any array A of numerical values, creates an array SA of the same size as A, such that $\forall_{k\geq 0} SA[k] = \sum_{i=0}^{k} A[i]$.

By a *device* we understand a graphics card hardware (GPU, memory). The computer hardware (CPU, memory, motherboard), which sends tasks and data to the device, will be called a *host*. By *kernel* we understand a function which is run concurrently in many threads on a device.

[1] This notation of concatenation might be somewhat misleading. For example, $01b + 10b = 1001b$. This stems from the fact, that the second operand of the operator $+$ consists of the bits that will be more significant in the resulting concatenated bitmap.

By *query* we understand a specification of a subset of data stored in a database. By *query processing* we understand a process of finding data specified in a query by means of compressed bitmap indexes.

3. Related work

3.1. Bitmap compression techniques

Multiple bitmap compression techniques have been proposed in the research literature, e.g., BBC (see Antoshenkov and Ziauddin, 1996), WAH (see Wu, Otoo and Shoshani, 2004a,b; Stockinger, 2007), PLWAH (see Deliège, 2009), RL-Huffman (see Nourani and Tehranipour, 2005), and RLH (see Stabno and Wrembel, 2009). All of them are based on the so-called run-length encoding. The run-length encoding consists in encoding continuous vectors of bits having the same value (either "0" or "1") into: (1) a common value of all bits in the vector (i.e., either "0" for a vector composed of zeros or "1" for a vector composed of ones) and (2) the length of the vector (i.e., the number of bits having the same value). 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.

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

The compression techniques proposed in Nourani and Tehranipour (2005) and Stabno and Wrembel (2009) additionally apply the Huffman compression (see Huffman, 1952) to the run-length encoded bitmaps. The compression technique presented in Nourani and Tehranipour (2005) was originally developed in the area of electronics for compressing scan cells results for large circuits. The main differences between this technique and RLH are as follows. First, in Nourani and Tehranipour (2005) 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 exact values. Second, in Nourani and Tehranipour (2005) 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. As a consequence, RLH produces more symbols (distances) than Nourani and Tehranipour (2005). Next, the distances are encoded by the Huffman compression.

3.2. GPU techniques in databases

Utilizing GPUs in database applications is yet not a very well researched field of computer science. Most of the research is focused on advanced rendering, image and volume processing as well as scientific computations (e.g., numerical algorithms and simulation). Few papers on GPU techniques are related to databases and most of them are related to efficient sorting (see Govindaraju et al., 2006, Greß and Zachmann, 2006, and Chen et al., 2009) or optimization of typical database operations (see Govindaraju et al., 2004) (selections based on multiple predicates and aggregations). Some approaches to accelerating data mining techniques on GPUs have also been proposed (see Böhm et al., 2009; Cao, Tung and Zhou; 2006, Andrzejewski, 2007 and Shalom, Dash and Minh, 2008).

Compression techniques with the support of GPUs mainly focus on compressing images (see Erra, 2005). To the best of our knowledge, the only paper dedicated to the problem of accelerating compression and decompression of bitmap indexes with the support of GPUs is Andrzejewski and Wrembel (2010). It presents the implementation of the WAH compression technique on GPUs and discusses the efficiency of this implementation.

4. Algorithms

In this section we present four algorithms: (1) an algorithm for extending an input bitmap (Algorithm 1), (2) an algorithm for compressing an extended input bitmap (Algorithm 2), (3) an algorithm for decompressing of a compressed bitmap to its extended version (Algorithm 5) and (4) and algorithms for performing bitwise operations on compressed bitmaps (Algorithms 6 and 7).

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 seven stages. Executions of stages must be sequential, but 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 (where each word is either a tail word or a pre-fill word). The most significant bit in each word is utilized to store the word class information. If the word 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 (see 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 PLWAH algorithm. Pre-fill words also have correct two most significant bits, but they are not yet rolled into a single word and their 25 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 words may be calculated in parallel.

The second stage (lines 8–16) 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 it is a tail word, or the word number $i + 1$ is either a pre-fill word of a different class (the words differ on the two most significant bits) or a tail word which differs on more than a single bit from the preceding pre-fill word number i. This stage may also be easily parallelized, as each of the values in the array F may be calculated independently.

The third stage (line 17) performs an exclusive scan on array F and stores the result in the array SF. The result of this operation is used in the subsequent stages for calculating the sizes of each of the blocks. 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 another array ($T1$), which will contain a single value for each block in the input bitmap. Moreover, it is possible to obtain the size of this array by summing the last value stored in array SF with last value in array F (notice that the last value in F always equals 1). This size is stored in the variable m. Efficient, parallel algorithms for performing the scan operation have been proposed in the research literature (see Harris, Sengupta and Owens, 2007, and Sengupta et al., 2007).

The fourth stage (lines 19–24) prepares an array $T1$ of the size equal to m. For each word $E(i)$, for which $F[i]$ equals 1 (last words of the blocks) the algorithm stores, in the array $T1$ 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 $i+1$. Values stored in $T1$ are used by the subsequent stages for calculating the numbers of words in blocks, and 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 fifth stage (lines 25–31) prepares an array $T2$ of the size equal to m. For each position i of the array $T2$ the algorithm calculates the difference $T1[i] - T1[i-1]$, which gives the number of words in the corresponding block. The calculated difference is divided by $2^{25} - 1$ and rounded up, in order to calculate the number of fill words needed to encode the corresponding block. The calculated number of fill words is stored in $T2[i]$. As each value is calculated independently, this stage may be easily parallelized.

The sixth stage (line 33) performs an in-place exclusive scan on the array $T2$. The resulting array $T2$ contains, for each block, the starting position in the output bitmap, at which the compression of the corresponding block should start. Moreover, the sum of the last values in the array $T2$ before and after scan gives the number of words in the compressed bitmap.

The last, seventh stage (lines 35–61) generates the final compressed bitmap. 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 $T1$). Each of these words is a representative for its block. Moreover, the number of words in the block is calculated based on the values stored in the array $T1$ similarly as in the fifth stage. Finally, the starting position, at which the compressed words should be stored is retrieved from the array $T2$. Based on the retrieved preprocessed word and the number of words in a block, one or more compressed words are generated. If the retrieved word is a pre-fill word, a sequence of corresponding fill words is stored in the output bitmap, starting at position k, such that the overall number of words encoded in these fill words is equal to the number of words in a block. If the retrieved word is a tail word such that it contains only a single 1, or a single 0 bit, then similarly as in previous case, a sequence of empty or full fill words is generated. This time, however, the last fill word encodes the position of the aforementioned bit. In all other cases, the retrieved word is stored in the output bitmap C in its original state. 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 notice that compression of each of the blocks is sequential. Though it is possible to create a fully parallel algorithm, it would require more stages, and it would be more memory consuming. Moreover, in most cases, the number of words generated for each of the blocks should be very small, as there is only one fill word generated for each $2^{25} - 1$ pre-fill words in the input bitmap, which roughly corresponds to 128MB of data.

Let us now analyze **Algorithm 5** that implements decompression. It is composed of several stages, each of which must be completed before the next one is started, but each stage may process input data in parallel.

Algorithm 2 Parallel compression of extended data

Require: : Extended input bitmap E
Ensure: : Compressed Bitmap C
1: $n \leftarrow \|E\|/32$
2: Create an array F of size n {0 based indexing}
3: **for** $i \leftarrow 0$ to $n-1$ **in parallel do**
4: **if** $E(i) = 0_{32}$ **or** $E(i) = 1_{31} + 0_1$ **then**
5: $E(i)_{31} \leftarrow 1b$
6: **end if**
7: **end for**
8: **for** $i \leftarrow 0$ to $n-2$ **in parallel do**
9: {For definition of NextNotEqual see Algorithm 3}
10: **if** NextNotEqual$(E(i), E(i+1))$ **then**
11: $F[i] \leftarrow 1$
12: **else**
13: $F[i] \leftarrow 0$
14: **end if**
15: **end for**
16: $F[n-1] \leftarrow 1$
17: $SF \leftarrow$ exclusive scan on the array F
18: $m \leftarrow F[n-1] + SF[n-1]$ {m is the number of words in the compressed bitmap}
19: Create arrays $T1$ and $T2$ of size m {0 based indexing}
20: **for** $i \leftarrow 0$ to $n-1$ **in parallel do**
21: **if** $F[i] = 1$ **then**
22: $T1[SF[i]] \leftarrow i+1$
23: **end if**
24: **end for**
25: **for** $i \leftarrow 0$ to $m-1$ **in parallel do**
26: $count \leftarrow T1[i]$
27: **if** $i \neq 0$ **then**
28: $count \leftarrow count - T1[i-1]$
29: **end if**
30: $T2[i] = \lceil count/(2^{25}-1) \rceil$
31: **end for**
32: $rs \leftarrow T2[m-1]$
33: $T2 \leftarrow$ exclusive scan on the array $T2$ {Original $T2$ array is no longer needed}
34: $rs \leftarrow rs + T2[m-1]$
35: Create a bitmap C such, that $\|C\| = rs * 32$
36: **for** $i \leftarrow 0$ to $m-1$ **in parallel do**
37: $count \leftarrow T1[i]$
38: $j \leftarrow count - 1$
39: $k \leftarrow T2[i]$
40: $X \leftarrow E_{(}j)$
41: **if** $i \neq 0$ **then**
42: $count \leftarrow count - T1[i-1]$
43: **end if**
44: **if** $X_{31} = 1b$ **then**
45: $tempOut \leftarrow 1_{25} + 0_5 + X_{30 \rightarrow 31}$
46: storeWords($tempOut$,$count$,k,C) {For definition of storeWords function, see Algorithm 4}
47: $C(k) \leftarrow$ 25bit representation of $count + 0_5 + X_{30 \rightarrow 31}$
48: **else if** $|\{p = 0, \ldots, 30 : B_p = 1b\}| = 1$ **then**
49: $p \leftarrow$ position of 1 in B increased by 1.
50: $tempOut \leftarrow 1_{25} + 0_5 + 10b$
51: storeWords($tempOut$,$count$,k,C)
52: $C(k) \leftarrow$ 25bit representation of $count$+ 5bit representation of $p + 10b$
53: **else if** $|\{p = 0, \ldots, 30 : B_p = 0b\}| = 1$ **then**
54: $p \leftarrow$ position of 0 in B increased by 1.
55: $tempOut \leftarrow 1_{25} + 0_5 + 11b$
56: storeWords($tempOut$,$count$,k,C)
57: $C(k) \leftarrow$ 25bit representation of $count$+ 5bit representation of $p + 11b$
58: **else**
59: $C(K) \leftarrow X$
60: **end if**
61: **end for**
62: C contains the compressed bitmap E

Algorithm 3 NextNotEqual function

Require: : Current word A and next word B
Ensure: : True, if A and B should be considered as not equal, false otherwise
if $A = 0_{31} + 1b$ and $|\{i = 0, \ldots, 30 : B_i = 1b\}| = 1$ **then**
  Return FALSE
else if $A = 1_{32}$ and $|\{i = 0, \ldots, 30 : B_i = 0b\}| = 1$ **then**
  Return FALSE
else
  Return logical value of expression: $A_{30\to31} \neq B_{30\to31}$ **or** $A_{31} = 0b$
end if

Algorithm 4 StoreWords function

Require: $tempOut$ word to be stored, $count$ the word counter, k store position, C output bitmap
Ensure: C contains appropriate number of tempOut words
while $count > 2^{25} - 1$ **do**
  $count \leftarrow count - (2^{25} - 1)$
  $C(k) \leftarrow tempOut$
  $k \leftarrow k + 1$
end while

Algorithm 5 Parallel decompression of compressed data

Require: : Compressed Bitmap C
Ensure: : Extended input bitmap E
$m \leftarrow \|C\|/32$
Create an array S of size m
for $i \leftarrow 0$ to $m - 1$ **in parallel do**
  if $C(i)_{31} = 0b$ **then**
    $S[i] \leftarrow 1$
  else
    $S[i] \leftarrow$ the value of $count$ encoded on bits $C(i)_{0\to24}$ increased by 1 if $C(i)_{25\to29} \neq (0)_5$
  end if
end for
$SS \leftarrow$ exclusive scan on the array S
$n \leftarrow SS[m-1] + S[m-1]$ {n contains the number of words in a decompressed bitmap}
Create an array F of size n filled with zeroes {0 based indexing}
for $i \leftarrow 1$ to $m - 1$ **in parallel do**
  $F[SS[i] - 1] \leftarrow 1$
end for
$SF \leftarrow$ exclusive scan on the array F
Create a bitmap E of length $\|E\| = n * 32$
for $i \leftarrow 0$ to $n - 1$ **in parallel do**
  $D \leftarrow C(SF[i])$
  if $D_{31} = 0b$ **then**
    $E(i) \leftarrow D$
  else
    if $D_{30} = 0b$ **then**
      $E(i) \leftarrow 0_{32}$
    else
      $E(i) \leftarrow 1_{31} + 0_1$
    end if
    if $(SF[i] \neq SF[i+1]$ or $i = n - 1)$ and $D_{25\to29} \neq 0_5$ **then**
      $p \leftarrow$ value encoded on $D_{25\to29}$ decreased by 1
      Negate bit $E(i)_p$
    end if
  end if
end for
E contains a decompressed bitmap C

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 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 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 17–33) 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. If the fill word encodes a bit position at which the last encoded word differs from the previous words, and the currently decompressed word is the last word of the block ($SF[i] \neq SF[i+1]$), the bit of the decompressed word, at the encoded position is negated. 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 compared to the internal device memory bandwidth (see NVidia1, 2010). This problem can be

partially eliminated by processing all queries 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 transferred 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). It is also possible to perform bitwise operations on compressed bitmaps in order to eliminate the need for bitmap decompression step. Our algorithm for parallel bitwise operations on compressed bitmaps is dedicated primarily to the WAH algorithm, though the efficient parallel conversion of the PLWAH compressed bitmap to the WAH compressed one is simple.

Let us consider **Algorithm 6** which performs such conversion. The algorithm is composed of three stages, each of which must be completed before the next one is started. Notice that each stage may process input data in parallel. The main difference between WAH and PLWAH compression algorithms is that fill words in PLWAH may encode additional words at the end of the compressed run. Such words therefore need to be decomposed into two WAH words: one fill word and one tail word.

The first stage (lines 2–5) creates array S of the size equal to the number of words in the PLWAH compressed bitmap. Consequently, each entry in the array corresponds to one word in the input, compressed bitmap. Each entry in this array stores the number of words that the corresponding word in the PLWAH compressed bitmap needs to be decomposed to. If the PLWAH word is a tail word or a fill word which does not encode an additional word at the end of the run, it does not need to be decomposed and therefore the value stored in the array should be equal to 1. On the other hand, if the PLWAH word is a fill word, which encodes an additional word at the end of the run, the value in the array should be equal to 2 as it needs to be decomposed into two WAH words.

The second stage (line 6) performs an exclusive scan on array S and stores the result in array SS. The result of this operation is directly tied to storing the conversion 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 converted bitmap at which decomposition of the word $C(i)$ should start. Moreover, the sum of the last values of the arrays S and SS is equal to the number of words the resulting bitmap will have (see line 7).

Algorithm 6 Parallel conversion of PLWAH to WAH algorithm

Require: : PLWAH Compressed Bitmap C
Ensure: : WAH Compressed Bitmap W
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: $S[i] \leftarrow$ the number of words the word $C(i)$ needs to be decomposed to
5: **end for**
6: $SS \leftarrow$ exclusive scan on the array S
7: $n \leftarrow SS[m-1] + S[m-1]$ {n contains the number of words in a converted bitmap}
8: Create a bitmap W of length $\|W\| = n * 32$
9: **for** $i \leftarrow 0$ to $m-1$ **in parallel do**
10: $W(SS[i]) \leftarrow$ the first word, the word $C(i)$ decomposes to
11: **if** $S[i] = 2$ **then**
12: $W(SS[i]+1) \leftarrow$ the second word, the word $C(i)$ decomposes to
13: **end if**
14: **end for**
15: W contains a converted bitmap C

The third stage (lines 9–14) performs the final conversion. In parallel, each of the words in the input bitmap is decomposed into one or two WAH words and stored in the output bitmap at the position determined by the values stored in array SS. This stage finalizes the PLWAH to WAH conversion.

Let us now consider how bitwise operations on WAH compressed bitmaps may be performed. Each compressed word in WAH compression scheme represents one or more words which are always the same, i.e. no compressed word may be decompressed into several distinct words. Consequently, given any two compressed words one may always determine a single word that is a result of the bitwise operation between them. There is, of course, the problem of the value of the counter in the case when both of the compressed words are fill words, but we will consider it later. Let us now consider an exemplary situation, where the first compressed bitmap $C1$ contains two words $C1(1)$ and $C1(2)$, which decompress into 3 and 4 words, respectively, whereas the second compressed bitmap $C2$ contains three words $C2(1)$, $C2(2)$ and $C2(3)$, which decompress into 2, 2 and 3 words respectively. In such a case one needs to determine the results of bitwise operations between *pairs* of words: $C1(1)$ and $C2(1)$, $C1(1)$ and $C2(2)$, $C1(2)$ and $C2(2)$, and finally $C1(2)$ and $C2(3)$. The counters of the obtained fill words may be determined easily based on values of counters of previously generated results of bitwise operations. In the case of the above example, the first generated word will have counter equal to 2. The second word will have counter equal to 1, as 2 (decompressed) words from the $C1(1)$ are already covered by the first generated word. The remaining counters may be determined similarly. The whole process is presented in Fig. 1.

Performing of the operations described above is relatively simple when done in sequence, compressed word after compressed word. Obtaining the same results in parallel is much more difficult. Our parallel algorithm for performing bitwise operations on WAH compressed bitmaps is presented in **Algorithm 7**. Similarly, as in all previous algorithms, this algorithm is also composed of seve-

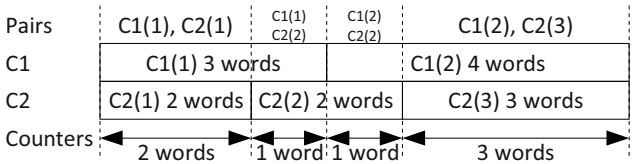

Figure 1. Finding counters in the result of compressed bitwise operation

ral stages, which must be executed in sequence. Notice that each of these stages is composed of tasks that may be executed in parallel.

The first stage (lines 1–7) creates two arrays, $S1$ and $S2$, which correspond to the input bitmaps $C1$ and $C2$, respectively. The number of entries in each of the arrays corresponds to the number of words in its corresponding bitmap. Consequently, each entry corresponds to a single word in the input bitmap. During this stage each compressed word is analysed. If it is a fill word, its counter (i.e. the number of words it may be decompressed to) is stored in the corresponding entry in the $S1$ or $S2$ array. If it is a tail word, then 1 is stored in the corresponding entry in the $S1$ or $S2$ array, as tail words do not change during decompression. For example, if $S1[2] = 4$, then $C1(2)$ is a fill word that may be decompressed into 4 words. This stage is a prerequisite for the next stage.

The second stage (lines 8–10) calculates an inclusive scan of arrays $S1$ and $S2$, obtained in the previous stage and stores the results in arrays $SS1$ and $SS2$, respectively. These arrays now store, for each of the words in the input arrays $C1(i)$ and $C2(i)$, the number of words that may be obtained by decompressing all of the words from arrays $C1$ and $C2$ up to and including $C1(i)$ or $C2(i)$. For example, if $SS2[10] = 20$ then by decompressing words $C2(i)$, where $i = 0, \ldots, 10$, one would obtain 20 decompressed words. Notice that arrays $SS1$ and $SS2$ are sorted in an ascending order and contain unique values.

The third stage (lines 11–16) stores in the least significant bit of each of the values in arrays $SS1$ and $SS2$ the information whether the value comes from array $SS1$ or $SS2$. Notice that after finishing this stage no value stored in one of the arrays appears in the second one (and vice versa). This stage is a preliminary step for the fourth stage that merges these two arrays. The third stage is required because of two reasons. Firstly, because of the limitation of the merge algorithm which works correctly only for arrays of distinct values. Secondly, because in the subsequent stages, the information about the source of value in the merged array will be required.

The fourth stage (line 17) merges the two arrays obtained in the previous stage. The result of the merging is stored in an array MS. The algorithm for efficient parallel merge on GPUs of two sorted arrays is described in Satish, Harris and Garlang(2009). The result of merging helps to determine in the

subsequent stages which pairs of compressed words will need to be processed to obtain the result. Recall the pairs of compressed words that were considered in the example of Fig. 1. Each consecutive pair contained one word from the previous pair and one new word (though there are cases in which both words may change). The least significant bits in the values in the merged array show for consecutive pairs which compressed word ($C1$ or $C2$) is different from the previous pair. The cases where both words in a pair change may be detected by comparing all but the least significant bits in the two consecutive values in the merged array. If they are equal, then both of these values represent a simultaneous change of both words in a pair. The fifth stage (lines 19–27) detects whether the merged array contains duplicate values, if the information appended in the third stage is ignored. The result of this stage is array F that for each of the values in array MS stores 0 if the next value in this array is equal to the current value and 1 otherwise. The number of ones in this array is equal to the number of words in the output bitmap.

The sixth stage (line 28) performs an exclusive scan of array F and stores the result in array SF. Array SF stores, for each of the values in array MS, an information to which word in the output they apply to. For example, if $SF[6] = 4$, then the fourth word of the output bitmap will be calculated based on the word from the one of the input arrays ($C1$ or $C2$) that is determined by the least significant bit in $MS[6]$ (recall the third and fourth stage). Moreover, if $SF[6] = 4$ then the counter of the fourth word in the output bitmap (if it is a fill word) is equal to ($MS[7]$ shift right by 1) − ($MS[6]$ shift right by 1) (as long as the difference is not equal to zero). Finally, as the sum of the last values in arrays F and SF is equal to the number of ones in F, then it gives the number of words in the output bitmap.

The seventh stage (lines 30–48) creates three arrays, $F1$, $F2$ and R. To explain the meaning of these arrays let us once again recall the pairs of compressed words that were considered in the example of Fig. 1 and the fact that each consecutive pair differed from the previous one by only one word (though it may happen that both words in a pair may change). Each of the arrays created in the seventh stage has as many entries as there are pairs that must be processed to obtain the result. Entries in the arrays $F1$ and $F2$ indicate for each pair which compressed word (from bitmap $C1$ or bitmap $C2$ respectively) is different with respect to the previous pair. For example, if $F1[5] = 1$ and $F2[5] = 0$, then to calculate the output word number 5 we need to use a next word from bitmap $C1$ and the word from bitmap $C2$ used to calculate the previous (fourth) output word. Array R contains the values of counters for the output words.

The eighth stage (lines 49–51) performs an inclusive scan on arrays $F1$ and $F2$ and stores results in arrays $SF1$ and $SF2$, respectively. These arrays, at each position i store the numbers of words in arrays $C1$ and $C2$ based on which should the word number i in the output be calculated.

The final, ninth stage (lines 30–48) generates the output bitmap. In parallel, for each of the output words, based on the information stored in the arrays $SF1$

Algorithm 7 Bitwise operation on two compressed bitmaps

Require: : WAH Compressed Bitmaps $C1$ and $C2$
Ensure: : PLWAH Compressed Bitmap CC containing result of bitwise operation of input bitmaps
1: **for** $j \leftarrow 1$ to 2 **in parallel do**
2: $mj \leftarrow \|Cj\|/32$
3: Create an array Sj of size mj
4: **for** $i \leftarrow 0$ to $mj - 1$ **in parallel do**
5: $Sj[i] \leftarrow$ the number of words the word $Cj(i)$ may be decompressed to
6: **end for**
7: **end for**
8: **for** $j \leftarrow 1$ to 2 **in parallel do**
9: $SSj \leftarrow$ inclusive scan on the array Sj
10: **end for**
11: **for** $j \leftarrow 1$ to 2 **in parallel do**
12: $m \leftarrow \|Cj\|/32$
13: **for** $i \leftarrow 0$ to $m - 1$ **in parallel do**
14: $SSj[i] \leftarrow (SSj[i]$ **shift left by 1 bit** $)$ **bitwise and** $(j - 1)$
15: **end for**
16: **end for**
17: $MS \leftarrow$ merged arrays $SS1$ and $SS2$
18: $m \leftarrow \|MS\|/32$
19: Create an array F of the size equal to m
20: **for** $i \leftarrow 0$ to $m - 2$ **in parallel do**
21: **if** $(MS[i]$ **shift right by 1** $) \neq (MS[i + 1]$ **shift right by 1** $)$ **then**
22: $F[i] \leftarrow 1$
23: **else**
24: $F[i] \leftarrow 0$
25: **end if**
26: **end for**
27: $F[m - 1] \leftarrow 1$
28: $SF \leftarrow$ an exclusive scan of the array F
29: $r \leftarrow F[m - 1] + SF[m - 1]$
30: Create two arrays $F1$ and $F2$ of the size equal to r filled with zeroes
31: Create an array R of the size equal to r
32: Create a bitmap CC of the length equal to $r * 32$
33: **for** $i \leftarrow 1$ to $m - 1$ **in parallel do**
34: **if** $MS[i]$ **bitwise and** $1 = 0$ **then**
35: $F1[SF[i]] \leftarrow 1$
36: **else**
37: $F2[SF[i]] \leftarrow 1$
38: **end if**
39: **if** $(MS[i]$ **shift right by 1**$) \neq (MS[i - 1]$**shift right by 1**$)$ **then**
40: $R[SF[i]] \leftarrow (MS[i]$ **shift right by 1**$) - (MS[i - 1]$**shift right by 1**$)$
41: **end if**
42: **end for**
43: **if** $MS[0]$ **bitwise and** $1 = 0$ **then**
44: $F1[0] \leftarrow 1$
45: **else**
46: $F2[0] \leftarrow 1$
47: **end if**
48: $R[0] \leftarrow MS[0]$ **shift right by 1**;
49: **for** $j \leftarrow 1$ to 2 **in parallel do**
50: $SFj \leftarrow$ exclusive scan on the array Fj
51: **end for**
52: **for** $i \leftarrow 0$ to $r - 1$ **in parallel do**
53: $b1 \leftarrow C(SF1[i])$
54: $b2 \leftarrow C(SF2[i])$
55: **if** $b1$ or $b2$ encodes a WAH tail word **then**
56: $CC[i] \leftarrow$ bitwise operation between $EX(b1)$ and $EX(b2)$ {for description of the EX function see the appropriate paragraph in the section 4}
57: **else**
58: $CC[i] \leftarrow$ a PLWAH fill word of the fill bit value calculated based on $b1$ and $b2$ and the counter equal to $R[i]$
59: **end if**
60: **end for**
61: CC contains the result of bitwise operation between $C1$ and $C2$

and $SF2$, appropriate words from $C1$ and $C2$ are retrieved. If both of the retrieved words are fill words, then a new fill word with a counter retrieved from array R is created. The information whether it is an empty or full fill word is determined based on the type of bitwise operation and the retrieved fill words. If one of the retrieved words is a tail word then the retrieved pair of compressed words must be processed differently. We use function $EX(x)$, which transforms word x in the following way. If word x is a tail word, then it remains unchanged. If word x is a fill word, it is changed into either (1) a word composed of 32 bits equal to 0 or (2) 31 bits equal to 1 and the most significant bit equal to 0. Next, an appropriate bitwise operation is performed and an output word is generated. The output words are stored in output array CC that finalizes the algorithm.

5. Experimental evaluation

In order to evaluate the performance of the developed GPU versions of PLWAH, we ran multiple experiments on a Core i7 2.8GHz CPU and NVIDIA Geforce 285 GTX graphics card. In the experiments we measured:

- CPU compression, decompression, and recompression times (applied to an extended bitmap, see Algorithm 1),
- GPU compression, decompression, and recompression times,
- time of uploading the input bitmap to the graphics cards memory (separately for each type of operation),
- time of downloading the output bitmap from the graphics cards memory (separately for each type of operation),
- time of calculating bitwise operation between two bitmaps using the following variants:

  Variant 1. (1) sending two compressed bitmaps to the device, (2) decompressing them, (3) performing bitwise operation on decompressed bitmaps, (4) sending the result back to the host;

  Variant 2. (1) sending two compressed bitmaps to the device, (2) performing bitwise operation on the compressed bitmaps, (3) decompressing bitwise operation result, (4) sending decompressed result back to the host;

  Variant 3. (1) sending two compressed bitmaps to the device, (2) performing bitwise operation on the compressed bitmaps, (3) sending compressed result of bitwise operation back to the host, (4) decompressing result of bitwise operation result on the host;

  Variant 4. (1) decompressing two compressed bitmaps on the host, (2) performing bitwise operation on the decompressed bitmaps;

  Variant 5. (1) performing bitwise operation on two compressed bitmaps on the host, (2) decompressing the result of bitwise operation result.

Additionally, as a reference, we included performance characteristics of the previously developed GPU version (see Andrzejewski and Wrembel, 2010) of the Word Aligned Hybrid (WAH) bitmap compression algorithm that has been reported to provide the shortest query execution time (see Stockinger, 2007, and Wu, Otoo and Shoshani, 2002).

Each of the tested input bitmaps for the first four measurements was composed of $1,677,721,600$ bits (200 MB). Bitwise operations between two bitmaps were tested on 120 MB bitmaps because of the large memory requirements of Algorithm 7. The density of bits equal to one was parameterized and ranged from 0.5 to 1/65536 ($1/2^i$ where $i = 1, 2, \ldots, 16$). The bits whose values were set to one were randomly selected. We generated three instances of each of the bitmaps for each experiment. The execution times discussed below represent averages of three experiments. The results of the experiments are shown in Figs. 2, 3, and 4.

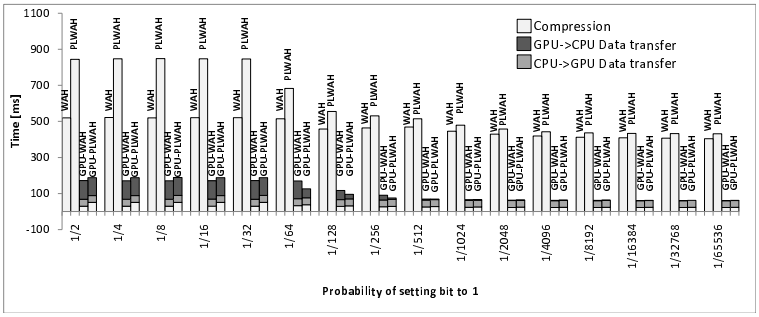

Figure 2. Compression and data transfer times for bitmaps of varying density

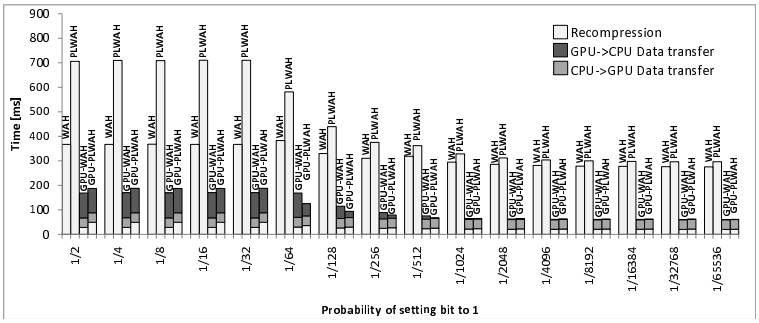

Figure 3. Recompression and data transfer times for bitmaps of varying density

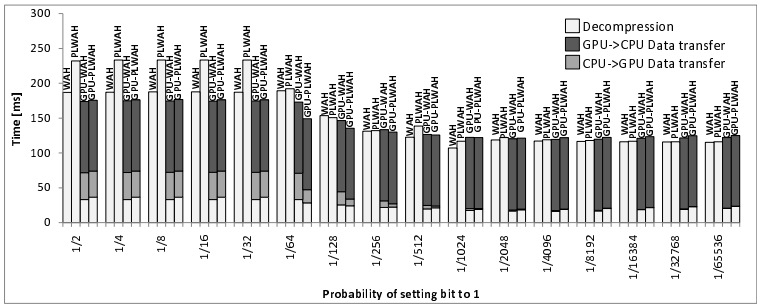

Figure 4. Decompression and data transfer times for bitmaps of varying density

Let us first compare the performance of our implementations of the WAH and PLWAH algorithms. First, we may notice that PLWAH implementation is slower than the WAH implementation (see Figs. 2, 3 and 4). This is expected, as PLWAH is more complex than WAH and several optimizations utilized in WAH cannot be used in PLWAH. Moreover, the GPU version of PLWAH incorporates a stage which detects whether a counter in the fill word may be bigger than 25 bits. On the contrary, there is no such stage in the GPU version of the WAH algorithm as 30bit counters are enough to compress the largest bitmaps given the modern graphics card memory constraints. Another interesting observation is that the difference between performance of PLWAH and WAH compression and recompression algorithms diminishes for sparse bitmaps. Notice that, for bitmap density equal to 0.5, the CPU version of the PLWAH compression algorithm is by 66% slower than the CPU version of the WAH compression algorithm. The GPU version of the PLWAH compression algorithm is by 68% slower than the GPU version of the WAH compression algorithm. For bitmaps of density 1/65535, the CPU PLWAH compression algorithm is only by 6% slower than the CPU WAH compression algorithm. The GPU PLWAH compression algorithm is only by 4% slower than the GPU WAH compression algorithm. Similar observations may be made for the recompression algorithm.

The fact that the difference in compression times diminishes with the decrease in bitmap density is probably caused by the fact that the lower the density, the longer are the runs that may be compressed. For CPU this means that more iterations of the main loop use the same code branch, which may be more consistent with the CPU branch prediction algorithms. This, in turn, causes that there is less influence of the increased complexity of the PLWAH algorithm on processing time. In the case of GPU, this means that in most cases each thread in a warp executes the same code branch. Execution of these threads, therefore, does not need to be serialized. Consequently, such execution is more similar to the WAH implementation that does not have as many code branches as the PLWAH implementation.

The results of the experiments for compression and recompression are presented in Figs. 2 and 3, respectively. Both of these figures are very similar, as essentially both of them present results of 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 120ms on host for the tested bitmaps, and indeed the difference between compression and recompression time is about 102ms in the best case and 156ms in the worst case. The same difference on the device is much smaller, i.e., 2.14ms in the best case and 2.49ms in the worst case.

Let us now analyse the speedups achieved by our PLWAH compression, recompression, and decompression implementations that utilize computer graphics cards. Table 1 presents these speedups calculated for the best, worst, and average cases. Each speedup is calculated in two variants, namely: (1) based on the pure processing times without the data transfer times between the host and the device, and (2) with the data transfer times included.

Table 1. Speedups achieved by GPU PLWAH compression scheme implementation

	Memory transfers included		
Case	Compress	Recompress	Decompress
Best	7.409	5.316	1.324
Worst	4.490	3.774	**0.928**
Average	5.495	4.272	1.141
	Without memory transfers		
Best	18.946	16.168	6.766
Worst	16.513	13.951	4.937
Average	17.508	14.479	6.146

Notice that the GPU-PLWAH implementation is faster than the CPU one in almost all cases. Compression on the device is 5.5 times faster on average even if the data transfer times are included, and 17.5 times if not. Recompression is 4.2 times faster on average with the data transfers included, and almost 14.5 times without them. The speedups achieved for decompression are unfortunately not as good. Though we still achieve speedup on average, the decompression on a GPU in the worst case may be slower than on a CPU. Notice that the decompression itself (without memory transfer), even in the worst case, is 4.9 times faster than on the host. Low decompression performance is caused by data transfer to and from graphics card (see Fig. 4). The host→device transfers do not take much time as compressed bitmap is transferred. The device→host transfers are much slower as we need to 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 than back. This feature results in substantial transfer times. As a consequence, the benefit of performing the decompression on the device is diminished by low transfer time.

Regarding the data transfer times, one may also notice that for compression and recompression (see Figs. 2 and 3) the device→host transfer times monotonically depend on the bitmap density, whereas host→device transfer times are constant. This is a consistent behaviour as the less dense the bitmap is, the smaller the compression (recompression) result (less data is transferred from the device to the host). The host→device transfer times are constant because the uncompressed bitmap of constant size is transferred to the device in each experiment. For decompression one may notice a reversed situation (see Fig. 4), i.e., the host→device transfer times monotonically depend on the bitmap density, whereas device→host transfer times are constant. This is a similar case as described before, with a difference that a compressed bitmap is transferred to the device, and a decompressed bitmap (of constant size) is transferred back to the host.

Let us now consider times of performing bitwise operations between two bitmaps (see Figs. 5 and 6). Fig. 5 presents times obtained for each of the variants (see beginning of this section), for each of tested bitmap densities while using WAH compression scheme.

Fig. 6 presents results of the same experiment but for the PLWAH compression scheme. Notice that both charts are very similar and the only difference is that the times shown in Fig. 6 are slightly longer than the times in Fig. 5. This is of course expected, as PLWAH bitwise operation algorithm is essentially the same as WAH operation algorithm, with only one additional stage included. We may also notice, that for dense bitmaps, the best variant is the first one, followed by the second, fourth, third and fifth variant, though there is not much difference between third and fourth variant. For more parse bitmaps the fourth variant performs worse than the third one. The decrease in performance of the fourth variant may be attributed to the previously observed phenomena of increased decompression times on CPUs for bitmaps of this density (see Andrzejewski and Wrembel, 2010). For the most sparse bitmaps fourth variant is the worst. All of the other variants are comparable, though the first variant seems to be a bit slower than the rest. Nevertheless, the first variant, which decompresses bitmaps and then performs bitwise operations on them, both using the GPU, seems to be the best one. The variants using algorithms for bitwise operations on compressed bitmaps are slower.

We attribute this to the fact that Algorithm 7 is complex and requires a large amount of global memory. As the cost of accessing the global device memory is high (see NVidia1, 2010), this unfortunately lowers performance of compressed bitwise operations. A little improvement of the second variant over the first one may be observed for the most sparse bitmaps. This is because the

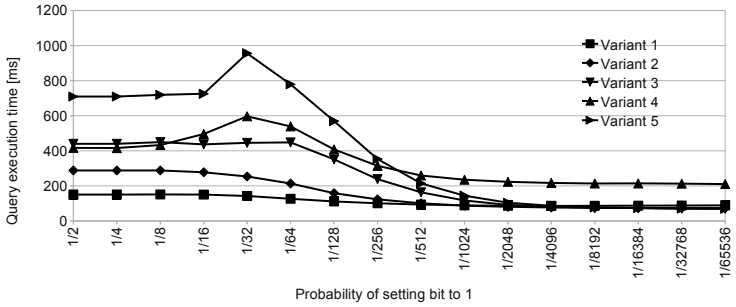

Figure 5. Query execution times for several variants of query processing procedures using WAH

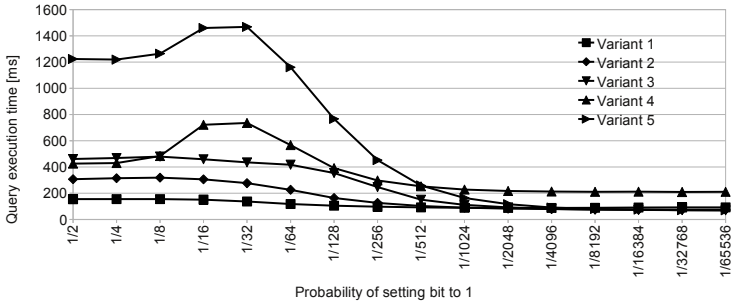

Figure 6. Query execution times for several variants of query processing procedures using PLWAH

first variant needs to decompress the input bitmaps and the second does not. For sparse bitmaps that compress well, this means that the number of global memory accesses is smaller for the second variant than for the second one and therefore its performance is better.

6. Summary

In this paper we presented the GPU-based implementation, called GPU-PLWAH, of the PLWAH compression/decompression technique. GPU-PLWAH allows to parallelize compressing and decompressing steps and to execute them with the graphics processing units on the CUDA platform. We also presented experimental evaluation of GPU-PLWAH, and compared it to (1) the standard CPU-based PLWAH, (2) the standard CPU-based WAH, and (3) the GPU-based WAH.

As the experiments show, the GPU-PLWAH compression performs on average 5.5 times faster than the CPU-based one. The experiments have also shown that while decompression itself is several times faster on a GPU than on a CPU, the data transfer between GPU memory and computer memory is a bottleneck. Still, we may alleviate this problem to a certain degree by utilizing the fact that data transfers and computations on the device may be performed in parallel. However, as was suggested in 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.

Finally, we have presented a parallel algorithm for performing bitwise operations on the WAH and PLWAH compressed bitmaps. It turned out that the parallel algorithm did not improve the performance but it showed that parallel bitwise operations on compressed bitmaps are possible. The implementation of the algorithm outlined in this paper was a naïve one, without any optimization. We argue that the algorithm could be better optimized, thus offering a better performance.

Future work will focus on: (1) lowering the number of stages and lowering memory requirements of Algorithm 7, (2) implementing on the CUDA platform other compression techniques including BBC and RLH-n, and comparing their efficiency, (3) applying several optimizations dedicated to given graphics cards computing capabilities, including the upcoming FERMI architecture (see NVidia2, 2010).

References

Andrzejewski, W. (2007) Fast K-Medoids Clustering on PCs. *Proc. of the ADMKD Workshop.* Technical University of Varna, 29–44.

Andrzejewski, W. and Wrembel, R. (2010) GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. *Proc. of Int. Conference on Database and Expert Systems Applications (DEXA).* **LNCS 6262**, Springer, 315–329.

Antoshenkov, G. and Ziauddin, M. (1996) Query processing and optimization in Oracle RDB. *VLDB Journal* **5**(4), 229–237.

Böhm, C., Noll, R., Plant, C. and Wackersreuther, B. (2009) Density-based clustering using graphics processors. *Proc. of ACM SIGMOD Int. Conference on Information and Knowledge Management (CIKM).* ACM Press, 661–670.

Cao, F., Tung, A.K.H. and Zhou, A. (2006) Scalable clustering using graphics processors. *Advances in Web-Age Information Management.* **LNCS 4016**, Springer, 372–384.

Chen, S., Zhao, J., Qin, J. and Heng, P.-A. (2009) An efficient sorting algorithm with CUDA. *Journal of the Chinese Institute of Engineers* **32** (7), 915–921.

Deliège, F. (2009) Concepts and Techniques for Flexible and Effective Music Data Management. PhD Thesis, Aalborg University, Denmark.

Erra, U. (2005) Toward real time fractal image compression using graphics hardware. *Proc. of Advances in Visual Computing*. **LNCS 3804**, Springer, 723–728.

Govindaraju, N., Gray, J., Kumar, R. and Manocha, D. (2006) GPUTeraSort: high performance graphics co-processor sorting for large database management. *Proc. of ACM SIGMOD Int. Conf. on Management of Data*. ACM Press, 325–336.

Govindaraju, N., Lloyd, B., Wang, W., Lin, M., and Manocha, D. (2004) Fast computation of database operations using graphic processors. *Proc. of ACM SIGMOD Int. Conference on Management of Data*. ACM Press, 215–226.

Gress, A. and Zachmann, G. (2006) GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society.

Harris, M., Sengupta, S. and Owens, J.D. (2007) Parallel prefix sum (scan) with CUDA. *GPU Gems 3*. Addison Wesley, 851–875.

Huffman, D.A. (1952) A method for the construction of minimum-redundancy codes. *Proc. of the Institute of Radio Engineers*. The Institute of Radio Engineers Inc., 1098–1101.

Lehman, T.J. and Carey, M.J. (1986) A study of index structures for main memory database management systems. *Proc. of Int. Conference on Very Large Databases (VLDB)*. Morgan Kaufmann, 294–303.

Nourani, M. and Tehranipour, M.H. (2005) RL-Huffman encoding for test compression and power reduction in scan applications. *ACM Transactions on Design Automation of Electronic Systems* **10** (1), 91–115.

NVidia1 (2010) NVIDIA CUDA C Programming Best Practices Guide. *NVIDIA CUDA C Toolkit 2.3*.

NVidia2 (2010) NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *White Paper*, NVIDIA.

O'Neil, P. and Quass, D. (1997) Improved query performance with variant indexes. *Proc. of ACM SIGMOD Int. Conference on Management of Data*. ACM Press, 38–49.

O'Neil, E., O'Neil, P. and Wu, K. (2007) Bitmap index design choices and their performance implications. Research Report No. 62756, Lawrence Berkeley National Laboratory.

Sathish, N., Harris, M. and Garlang, M. (2009) Designing efficient sorting algorithms for manycore GPUs. *Proc. of the IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, 1–10.

Sengupta, S., Harris, M., Zhang, Y. and Owens, J.D. (2007) Scan primitives for GPU computing. *Proc. of the Graphics Hardware 2007 Conference*. ACM Press, 97–106.

Shalom, S.A.A., Dash, M. and Minh, T. (2008) Efficient K-Means Clustering Using Accelerated Graphics Processors. *Proc. of Int. Conference on Data Warehousing and Knowledge Discovery (DaWaK)* **LNCS 5182**, Springer, 166–175.

Stabno, M. and Wrembel, R. (2009) RLH: Bitmap Compression technique based on run-length and Huffman encoding. *Information Systems* **34** (4–5), 400–414.

Stockinger, K. and Wu, K. (2007) Bitmap indices for data warehouses In: R. Wrembel and C. Koncilla, eds., *Data warehouses and OLAP: Concepts, Architectures and Solutions*. Idea Group Inc., 157–178.

Wu, K., Otoo, E.J., and Shoshani, A. (2002) Compressing bitmap indexes for faster search operations. *Proc. of the Int. Conference on Scientific and Statistical Database Management (SSDBM)*. IEEE Computer Society, 99–108.

Wu, K., Otoo, E.J. and Shoshani, A. (2004a) An efficient compression scheme for bitmap indices. Research Report No. 49626, Lawrence Berkeley National Laboratory.

Wu, K., Otoo, E.J. and Shoshani, A. (2004b) On the performance of bitmap indices for high cardinality attributes. *Proc. of Int. Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 24–35.

Wu, K., Otoo, E.J. and Shoshani, A. (2006) Optimizing bitmap indices with efficient compresion. *ACM Transactions on Database Systems (TODS)* **31** (1), 1–38.

Wu, M. and Buchmann, A. (1998) Encoded Bitmap indexing for data warehouses. *Proc. of Int. Conference on Data Engineering (ICDE)*. IEEE Computer Society, 220–230.

Zaker, M., Phon-Amnuaisuk, S. and Haw, S. (2008) An adequate design for large data warehouse systems: Bitmap index versus b-tree index. *International Journal of Computers and Communications* **2** (2), 39–46.