DECOMPRESSING INVERTED INDEXES USING A GPU-BASED

PARALLEL-QS METHOD

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE

(Electrical Engineering)

at the

POLYTECHNIC INSTITUTE OF NEW YORK UNIVERSITY

by

Shi Li

January 2014

Approved:

_____
Advisor Signature

_____
Date

_____
Department Head Signature

_____
Date

Copy No. **# 1**

Student ID#: **0464604**

Vita

Shi Li was born on December 14th, 1988. In 2007, he entered South China University of Technology, Guangzhou, China. He completed the Bachelor of Science in Electrical Engineering in June, 2011. In September 2011, he entered Polytechnic Institute of New York University, New York, U.S.A. to continue his master study.

Acknowledgement

The author wants to give special thanks to his supervisor Prof. Torsten Suel, who brought him to the world of research. Thanks to Torsten Suel's guidence that opened the door of web search engines and developed great interest in the area to the author. In addition, The author learned a lot from Torsten Suel's academic attitude. At last, thanks to Torsten Suel's patience help in correcting the mistakes in this thesis.

Dedicated to my parents Ruiqian Li and Yucai Jiang, who support me all the time.

ABSTRACT

DECOMPRESSING INVERTED INDEXES USING A GPU-BASED PARALLEL-QS METHOD

by

Shi Li

Advisor: Prof. Torsten Suel, Ph.D.
Submitted in Partial Fulfillment of the Requirements for
the Degree of Master of Science (Electrical Engineering)

January 2014

An inverted index is a data structure that is widely used by all major web search engines to perform query processing. During query processing, the inverted indexes are first fully or partial loaded to main memory and then decompressed for further lists intersection and ranking. Inverted indexes are always compressed using suitable compression algorithms and then stored on disk. Thus the higher the compression ratio is, the more compressed data can be kept in main memory, which benefits the performance of query processing. In addition, the decompression speed of inverted indexes is also very important since decompression of inverted indexes is performed during query processing; thus it is essential to accelerate the decompression speed as well. In this thesis, we provide a GPU-based method called Parallel-QS to accelerate the decompression speed of inverted indexes. GPUs are well-known for their high degree of parallelism, which is suitable

for workloads that can be preformed in parallel. In this thesis, we decompress the inverted indexes in parallel using the GPU-based Parallel-QS method. The Parallel-QS algorithm is based on the lower bits/upper bits representation of monotone sequences created by P. Elias, and the Quasi-Succinct indices introduced by S. Vigna. This thesis parallelizes the decompression of inverted indexes on the GPU using Parallel-QS. The experiment shows that after parallelizing workloads on the GPU, decompression speed is greatly increased.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Inverted indexes are used in major web search engines to perform query processing. Due to the rapid growth of data in recent years, search engines are facing the inevitable challenge that the size of the inverted index is increasing constantly, thus consuming more and more disk and memory space. Major web search engines have to process many queries per second over billions of web pages, and decompressing inverted index structures takes a lot of time. To cope with this heavy workload, a lot of efforts have been made to reduce the size of the inverted index and to increase decompression speed.

Index compression helps to store more data on disk or in memory, which saves disk and memory space and accelerates query processing. There are many algorithms in the area of index compression. The basic idea in index compression is to use fewer bits to represent document Ids (docIDs) and other index data, which are unique numbers assigned to the documents. There are two main measures to evaluate a compression algorithm, compression ratio and decompression speed. (Compression speed is not essential since the compression of inverted indexes only needs to be performed once before queries are processed, but the decompression of inverted indexes is performed while queries are being processed.) Intuitively, if the compression ratio of one compression algorithm is better than that of another, this means that the better algorithm uses fewer bits to represent integers than the other one. However, a better compression ratio may likely

have to perform more complex computations; this may result in a slower decompression speed. Thus we need to care about both compression ratio and decompression speed when choosing a compression algorithm. A well known idea when compressing sorted sequences of integers is differencing, which means computing the gap between two successive docIDs and storing this gap instead. Later, to decompress the gaps, a prefix sum operation, which computes the sum of all integers before the current index, is performed to get the original docIDs back. One popular algorithm in list compression method is PforDelta [13, 21], which has many different versions such us OptPFD [17], NewPFD [17] and Para-PFor [14].

In this thesis, we investigate the Quasi-Succinct (QS) algorithm [18], which provides a very good theoretical upper bound on the compression ratio, and parallelize this algorithm on Graphical Processing Units (GPUs). GPUs are specialized processors designed to accelerate graphics applications, and in particular to enhance these applications' performance using large numbers of parallel computing units. However due to the high parallelism of the GPU, researchers also tend to apply general-purpose heavy computational workloads to the GPU in order to achieve better performance, for example in the areas of databases and scientific computing [8, 9, 12]. This is also known as GPGPU (General Purpose Graphical Processing Units) computing.

We modify the QS algorithm and apply the modified version to the GPU. We focus only on the docIDs in the inverted index. There is also some other useful information stored in inverted indexes, such us frequencies and scores. The frequency is the number of times that the term appears in a documents, and the score is a number that shows the importance of the terms in the document. Such information can be also indexed by our Parallel-QS or by many other existing algorithms such us PForDelta [13, 21], or Rice coding [20], etc. We implement this modified QS

and test it using the TREC-GOV2 dataset. Our experimental results on the GPU show that by parallelizing the QS algorithm, we can greatly increase the decompression speed while still achieving better compression ratio than PForDelta.

The remainder of this thesis is organized as follows. In the next chapter, we provide some background and discuss related work. Chapter 3 summarize the contributions of this thesis. In Chapter 4, we introduce the idea to parallelize the QS algorithm on the GPU. In Chapter 5, we show more details about how we implement the system. In Chapter 6, we show our experimental results and compare the results to other algorithms. Chapter 7 provides some concluding remarks.

Chapter 2 Background and Related Work

We now provide some background on index compression in search engines and discuss related work.

2.1 Background

For a basic overview of IR query systems, see [4, 15]. For a survey of the work on index compression see [16].

Inverted Index: All major search engines use a data structure called inverted index to index and store information over hundreds of billions of web pages. We are given a collection of N documents, where each document is assigned a unique number, also called docID, from 0 to N-1. For all distinct terms (words) that appear in the collection, inverted lists are then constructed, where each inverted list I_w contains the docIDs of all documents in the collection that contain w. Usually, we first sort all docIDs that appear in the inverted list in an increasing order and store this sorted list so that it is more convenient for later query processing operations such us index decompression, lists intersection, and ranking. After we construct the inverted index, we usually

compress the whole index using appropriate compression algorithms, and store the compressed inverted index on disk or in memory. A typical inverted index structure consisting of three inverted lists can be seen as follows.

dog {1, 3, 16, 35}

cat {2, 3, 13, 16, 30, 66}

monkey {3, 12, 13, 15, 18, 20, 25, 30}

In this example, the term "dog" appears in the documents whose docIDs are {1, 3, 16, 35}.

Index Compression: It is always important to compress the inverted indexes using an appropriate compression algorithm. Recall that there are two main measures to evaluate a compression algorithm, compression ratio and decompression speed. An algorithm with a better compression ratio helps us cache more compressed data in cache, or load more data into main memory, which should make query processing faster. However, we need to care about the decompression speed at the same time, since algorithms with higher compression ratio tend to have slower decompression speed. There are many ideas to compress inverted indexes. Following the basic idea of differencing, we can create an array containing smaller numbers by computing the gaps between each pair of successive numbers in each inverted lists, For example, in the above list for dog, after computing the gaps, we would have an array {1, 2, 13, 17}, where we can represent 17 using five bits[1] in binary, and store it instead of the original 35 which consumes 6 bits. In this case we would save one bit. Realistically, in some long inverted lists (i.e., common terms), the docIDs can be very large, but the gap between two successive numbers would be very small. Thus we could greatly reduce the size of the inverted list by using this approach. To

1. Note that in fact the number 17 needs more than five bits to store because some meta information about the length of the code also needs to be stored, but it is an easy way to think about the problem like this.

decompress the inverted index, we perform a prefix sums operation. We will introduce two gap-oriented compression algorithms later in this chapter. There are of course many other compression ideas, but we will not introduce them here since this is not out main focus.

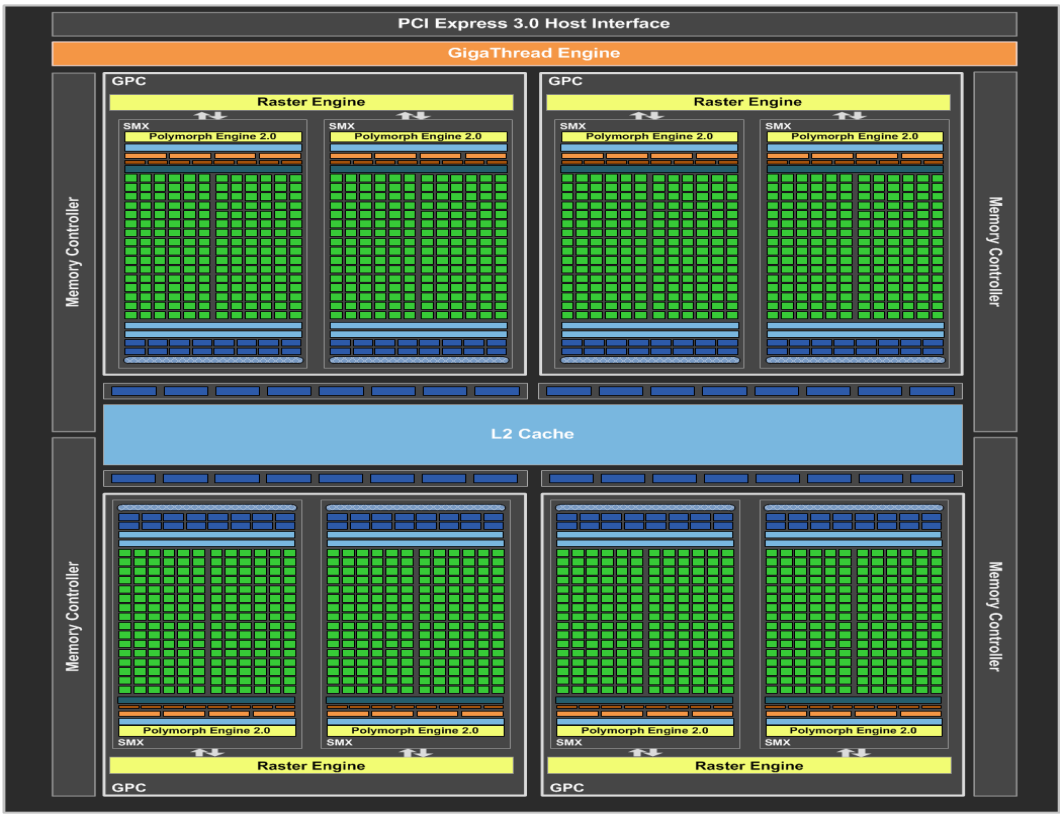


Figure 1: Basic structure of a GPU (from NVIDIA.com)

Graphical Processing Units (GPUs): GPUs were initially designed to achieve better performance in graphical applications such us 3D images and videos. Because of the high degree of parallelism in GPUs, in recent years, researchers have begun to transfer complex but highly parallel computational work from traditional CPUs to GPUs; this is the so-called General-Purpose GPU (GPGPU) computing. Some typical scenarios of this can be found in [2].

GPUs are carefully designed to allow a high degree of parallelism. A typical GPU structure can be seen in Figure 1. This is the structure of the GeForce GTX 680, codenamed Kepler, designed

by NVIDIA and released in 2012. On this chip, there are four Graphics Processing Clusters (GPCs), eight next-generation Streaming Multiprocessors (SMs), and four memory controllers. GPCs are the largest hardware unit on GPUs, which have their own raster engine, texture memory, and some other units. Each GPC contains two SMs, which perform most of the computational work on GPUs. All SMs are running the computation simultaneously, with a large number of so called cuda cores residing on each one of them, ensuring the high parallelism and efficiency of GPUs. For more details of the structure of GPUs and SMs, please refer to [3].

Single Instruction, Multiple Data (SIMD) is an introduction that is widely in parallel programming on CPUs. Using SIMD, all concurrent threads run exactly the same instructions at the same clock but over different data. Most modern CPUs include the SIMD instruction set to impove their parallel computing efficiency. Due to the high degree of parallelism design of the GPUs, the idea of SIMD can also be useful on the GPUs. One major GPU vendor, NVIDIA, introduced a SIMT (Single Instruction, Multiple Threads, which is one form of SIMD) parallel computing and programming platform called Compute Unified Device Architecture (CUDA) [2] in 2006. CUDA not only made programming on GPUs to perform general computing easier, but also often decreased running times compared to previous GPU APIs such us Open GL.

There are two main types of functions in CUDA programming, host functions and kernel functions. Host functions run on the CPU to control data transfer from the CPU to the GPU. Later to compute with this data, we use kernel functions. After the kernel functions complete the computation, we can copy the results from GPU back to CPU. On GPUs, kernel functions define how many threads need to be allocated, corresponding to the workload. The number of threads is defined by two parameters: threads per block and number of blocks. On state-of-the-art GPUs, every 32 threads are handled by one warp, which is the smallest unit of parallelism. When calling

the kernel function, blocks are assigned to different SMs, then warp schedulers within the block dispatch warps to different cuda cores. For more details of programming on GPUs using CUDA, see [1]. In the GTX 680, we have four warp schedulers per SM and we have eight SMs; this means that we can have a high degree of parallelism, of 1024 concurrent threads at a time.

2.2 Related Work

Rice Coding: Rice Coding [20] is a well known encoding algorithm invented by Robert F. Rice. After taking gaps between each pair of integers to be encoded, Rice coding chooses an constant integer *B,* where *B* is a power of two and is close to the average of all gaps. Then each gap is represented by two parts: a unary part and a binary part. The binary part stores the value of *v mod B* in no more than $log_2 B$ binary bits. The unary part stores the value of *floor(v/B)* in a unary code in another array. However, though standard Rice coding is often considered to be slow, there has been some optimization work (see [19]) which can make the algorithm much faster. A GPU-based optimization is performed in [6]. [6] keeps the compression the same but changes the decompression by performing two parallel prefix sums, one on the unary array and the other one on the binary array. After these two parallel prefix sums are complete, the final results can be computed by two simple operations, one shift on the unary number, and then adding the shifted unary number to the corresponding binary number. The experimental results in [6] show that the decompression speed is slightly faster than the speed on the CPU.

PForDelta: The PForDelta method was proposed by S. Heman in [13, 21]. When compressing gaps using PForDelta, a number *B* is first computed, such that *B* is bigger than most, say 90%, of

the gaps. This means that we can represent most of the gaps within log_2B binary bits. For the remaining gaps that are bigger than B, we consider them to be exceptions and store them in a separate linked list. Due to this special design, very few operations are needed to decode the gaps; thus the PForDelta method is very fast. There are some additional optimizations implemented in [17], which make PForDelta even faster. A GPU-based version of PForDelta is implemented in [6] by S. Ding et al. In [6], exceptions are stored in two separate arrays rather than storing them in a linked list. More precisely, given an exception, the lower log_2B bits are stored with all the other non-exceptions, while the other bits and the corresponding index are stored in two separate arrays. Then PForDelta is recursively performed on these two arrays until there are no exceptions. For more details of the GPU-based PForDelta, see [6]. The results provided in [6] show that the GPU version is can achieve better performance in decoding speed than the CPU version. Recently, Lemire et al. provided a CPU implementation called Fast-PForDelta in [14] using SSE2 (an extension of SSE). SSE (Streaming SIMD Extension), is an SIMD instruction set extension to x86 architectures, introduced by INTEL. The results in [14] show that the top speed to decompress the inverted indexes can be as fast as two billion integers per second. This thesis will not discuss all the optimized versions of PForDelta. Please see [14] for more details and a comparison between different compression algorithms.

Quasi-Succinct: The high bits/low bits representation for monotone sequences is proposed by Peter Elias in [7]. Very recently S. Vigna optimized this algorithm and applied it to inverted index structures and called it the Quasi-Succinct (QS) algorithm. Rather than computing the gaps first, QS algorithm first selects a number *b (also called bit-width)* within the inverted list, such that $b = max\{0, floor[log_2(c*u/n)]\}$, where u is the maximum value of the docIDs in the inverted list, n is the length of the inverted list, and c is a parameter (usually c is set to 1). Then the original

numbers in the inverted list are coded in two parts, with each part stored in an independent array, the lower bits array (LBA) and the upper bits array (UBA). LBA stores the rightmost *b* bits of all integers in binary; the value of the *b*-bits integers in the LBA are called lower values. In the UBA, all integers are first shifted to the right by *b* bits; then gaps between each two successive remaining values are computed; these numbers are called upper values. Finally, the gaps are stored in unary form in the UBA. As shown in [18], the QS algorithm provides a high compression ratio and can compress the inverted indexes close to its information-theoretic lower bound. Also, the QS provides constant-time access to all information stored in the index. When decompressing an array, the UBA and the LBA can be decoded independently. To decode the UBA, a prefix sum operation is performed over the decoded unary numbers. The final resulting numbers can be computed by shifting the summed up unary numbers *b* bits to the left and adding these shifted results to the corresponding binary numbers. An example can be seen in Figure 2. On our GPU-based system, we modify the QS representation based on S. Vigna's work. We parallelize the decoding of both UBA and LBA. More details are provided in the further chapter.

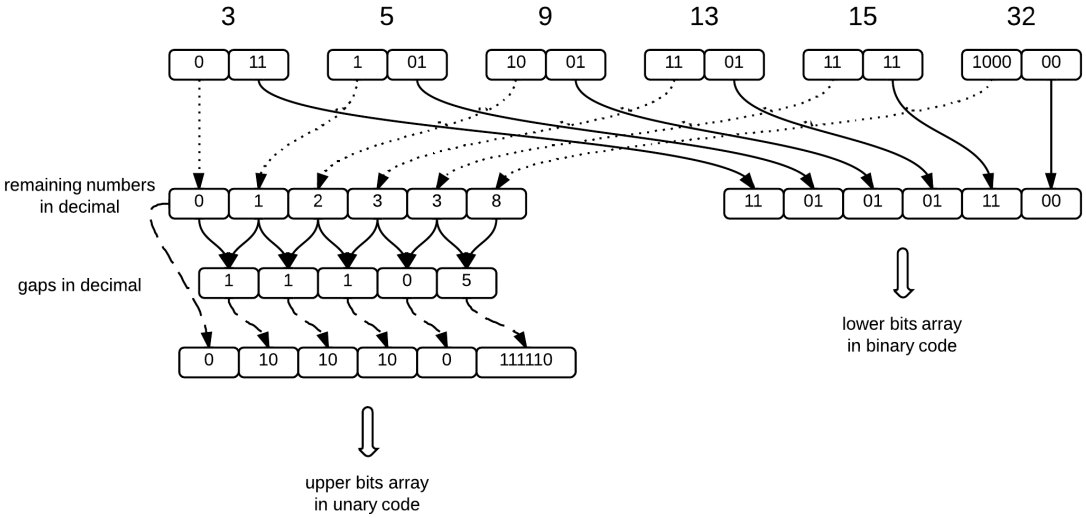


Figure 2: An example of compressing the array using QS. Here $b = floor[log_2(32/6)] = 2$, so the last two

bits of each integer are stored in the LBA. In the UBA, the gaps between each pair of successive values are computed, and the gaps are stored in unary form.

Chapter 3 Contributions of this Thesis

The main contributions of this thesis are as follows.

1. We carefully modify the QS compression algorithm. In order to parallelize the QS algorithm on the GPU, we create several mechanisms to ensure that all threads are performing exactly the same sequences of instructions over different data. We create a block-based parallel prefix sum rather than the traditional prefix sum.

2. We evaluate our Parallel-QS algorithm. We first find the parameters that are suitable for our implementation, such us thread alignment, and block size. Later we compare Parallel-QS to other compression algorithms.

Chapter 4 GPUs and Parallel-QS Algorithm

4.1 Programming on the GPU

The first API for programming on GPUs was introduced by the Khronos Group, and was called Open GL. Later, Microsoft introduced another API called DirectX. Due to the fact that these languages were originally designed only for graphics applications, researchers had to store data in texture memory and call the Open GL or DirectX functions to execute the computation. This was too complex for researchers, particularly for researchers that do not have any graphics background. In 2006, the introduction of the CUDA API by NVIDIA solved this problem. CUDA not only makes programming on a GPU almost as easy as it is on a CPU, but it often accelerates the computational speed greatly.

The idea of the CUDA API is to perform SIMD on a large scale, which is called SIMT by NVIDIA. Recall the idea of SIMD, we aim to perform strictly the same instructions for all concurrent threads but over different data. For a simple example, we have two arrays:

A { a_1, a_2, a_3, a_4};

B {b_1, b_2, b_3, b_4};

We want to add each corresponding pair of elements from these two arrays, and write the result into a third array, C, to get the following result:

C $\{a_1+b_1, a_2+b_2, a_3+b_3, a_4+b_4\}$;

On a CPU, a "for" loop or a "while" loop can be performed in order to accomplish such requirement. However on a GPU, following the idea of SIMT, the computational work (a_1+b_1) can be performed by one thread, and since the other three pairs of additions require the same instructions, they can be performed by three other threads concurrently.

Recall that there are two types of functions defined in CUDA, host functions and kernel functions. Host functions run before kernel functions to arrange memory space on the GPU, and after kernel functions to copy the results from the GPU back to the CPU. Kernel functions perform the instructions we discussed above in parallel. For more details of kernel and host functions, we provide the following codes:

```
__global__ add ( int* A, int* B, int* C)
{
    int tid = threadIdx.x;
   if (tid < 3)
   {
      C[tid] = A[tid] + B[tid];
   }
}
```

When executing the above codes, each thread will run the codes corresponding to its thread index. Thus, the actual code for each threads is as following.

```
__global__ add ( int* A, int* B, int* C)
{
    int tid = 0;
    C[0] = A[0] + B[0];
}
```

```
__global__ add ( int* A, int* B, int* C)
{
    int tid = 1;
    C[1] = A[1] + B[1];
}
```

```
__global__ add ( int* A, int* B, int* C)
```

```
__global__ add ( int* A, int* B, int* C)
```

```
{
    int tid = 2;
    C[2] = A[2] + B[2];
}
```

```
{
    int tid = 3;
    C[3] = A[3] + B[3];
}
```

Kernel functions are called by host functions. In the above example, we can call the kernel function (add()) by using following host function,

add<<<1,4>>>(dev_A, dev_B, dev_C);

where the first number in the angle brackets is the number of blocks, meaning how many blocks are defined (the number of parallel blocks when calling kernel functions), and the second number is the number of threads within each block. Thus we have a total number of 1*4 threads to execute the add function. (Also, “dev_A”, “dev_B”, and “dev_C” are arrays allocated on the GPU). For more details on how to allocate memory and copy data between the CPU and the GPU, please refer to [1] and [2].

4.2 Parallelizing the QS Algorithm

In [6], S. Ding implements a GPU-based PForDelta and a GPU-based Rice coding. Both of the algorithms are first modified to guarantee all threads proceed by using the same sequence of instructions over different data in the same clock. This is also the same with our parallel-QS implementation on the GPU. Before deciding to implement the QS on the GPU, we carefully studied the original high bits/low bits representation for monotone sequences discussed in [7] and the QS implementation on inverted index structures [18]. We then concluded that we could have high parallelism by modifying the QS algorithm to make it suitable for the GPU.

Bit-packing: Recall that the QS representation for a monotone array consists of two parts, the

unary part UBA and the binary part LBA. The LBA stores the packed lower *b* bits of all integers from a list, where *b* is the bit-width of the list. Considering that integers are stored using *32 bits*, however, it is unlikely to have a bit-width that is 32. This means that there exists some bits that are unused, which waste a lot of space on disk or in memory. Here an operation called bit-packing is involved to pack all *b-bit* lower value together to avoid this waste. After bit-packing, each integer can store *32/b b-bit* values. If *32* is divisible by *b*, it means that there contains exactly *32/b* lower values in each integer, such a value of *b* is called *special b*. Otherwise, it is called *common b* . An example of bit-packing for *special b* can be seen in Figure 3.

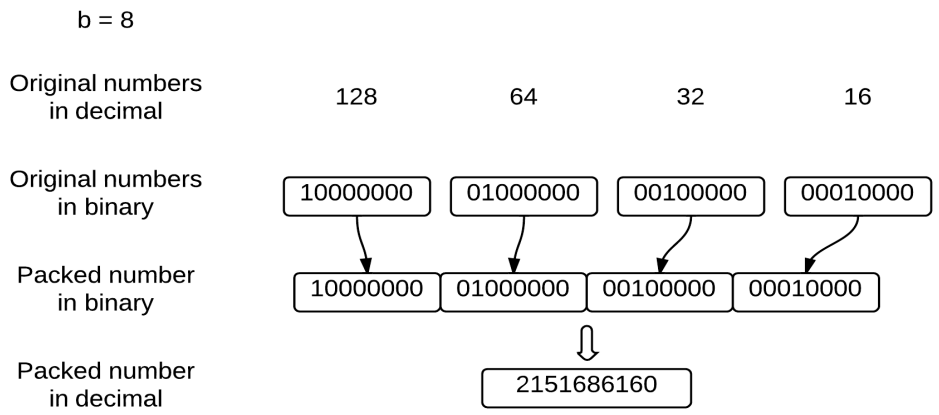


Figure 3: An example of bit-packing. In this figure, an array of {128, 64, 32, 16} is to be packed, where each integer can be represented using 8 binary bits. Before packing, it consumes 4*32 bits = 128 bits = 16 bytes to store the array. However, after packing the integers, it only cost 4 bytes.

From Figure 3 we can see that bit-packing is very important for compressing the LBA. The same mechanism holds true for the upper bits array. By packing the unary codes of the UBA, a lot of space can be saved as well.

Decompressing the LBA: The decompression of the LBA from packed integers is trivial. On the CPU, it only consumes two operations to recover a value in the LBA, shift and mask. Given

bit-width *b*, to recover the first integer from the encoded integer, the encoded integer is first shifted by *(32-b)* bits, then we mask the shifted result with (*(1<<b) -1)*. Now the first value in the lower bits array is already decoded. Such operations can be performed on a coded integer several times until there is no next *b* bits in this encoded integer. There are two options after the last *b* bits of the encoded integers be decoded. If the *b* is *special*, it means that the last *b* bits in the encoded integer are exactly the representation of the last integer to be decoded. However, if the *b* is *common*, there should be an offset with value *offset = 32 – floor[32/b]*b*, where *offset* < *b*. This means that the next lower value is represented by the *offset* and the *(b – offset)* bits in the next encoded integer, so this lower value should be decoded using these two encoded integers together. Then the next encoded integer is considered to start from its *(32 – (b – offset))* bit, and the rest of all encoded integers are decoded in the same way until we recover all lower values back.

Recall that on the GPU, all concurrent threads should execute the same sequences of instructions in lockstep. For the problem of decompressing the lower bits, each thread can be used to decode one lower value, which performs exactly the same shift and mask operations. However, the offsets should be carefully tailored to avoid collision between the boundaries of two integers. More details of the implementation are provided in the next chapter.

Decompressing the UBA: The UBA contains the gaps between each part of upper values and is represented in unary coded format. On the CPU, it is easy to transfer the unary codes to binary and perform a prefix sum operation to decode all upper values. However, this serial workload seems unsuitable for the GPU. After some consideration, we found that parallel prefix sum algorithms can solve this problem, see [10]. We use the CUDPP library [11] to implement the prefix sum. In addition, we implement a block-based structure on prefix sum that speeds up the decompression of the upper bits array. Later, we provide some details of how we implement this

block structure.

From the above descriptions, we are confident that it is possible to perform the decompression in parallel using the QS algorithm. In the next chapter, we show details of how we parallelize the decompression using the Parallel-QS algorithm on the GPU.

Chapter 5 Implementation of Parallel-QS

In this chapter, full details of the implementation are given. In the actual implementation, we merge some functions together to optimize the decompression performance. However, here we show all functions separately in order to let readers understand the details clearly.

5.1 GPU-based decompression for the LBA

Recall that in the decompression of the lower bits array on the CPU, two main operations are performed, shift and mask. In addition, we need to manage the offsets if the *b* is *common*. These operations are almost the same to the GPU, but they should be allocated to different threads. Each thread works on *b* bits within one or two encoded integers, depending on the offset of the thread. This means that to decompress the LBA, the numbers of threads that are needed is the length of the inverted list *l*. An example $b = 8$ is shown in Figure 4, which means that each encoded integer contains *four* lower values and each *eight* bits represent a lower value. *Four* threads are allocated; all threads perform the same instructions as shown in Figure 5. Each threads here has a unique ID called threadID, also know as tid; using the value of its tid, a thread retrieves one specific encoded integer and computs how many bits the encoded integer should be shifted. However, if *b*

is *special*, the instruction performed by threads would be much more complicated since the offset between two integers should be carefully tailored. An example is given in Figure 5, for $b = 7$.

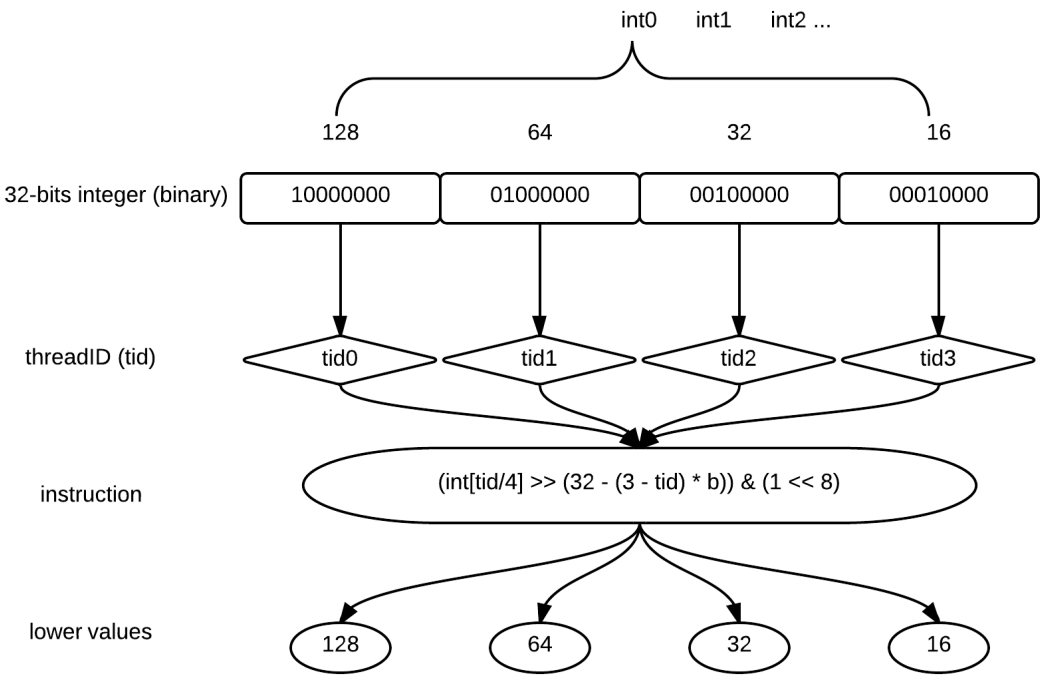


Figure 4: An example of GPU-based LBA decompression for a *special* b. In this figure, *b* is *special,* int0 contains four integers; and thus four threads are needed to decompression the integer. tid0, tid1, tid2, and tid3 are the four threads, where each thread shifts certain bits corresponding to its tid and then mask the result with (1 << 8).

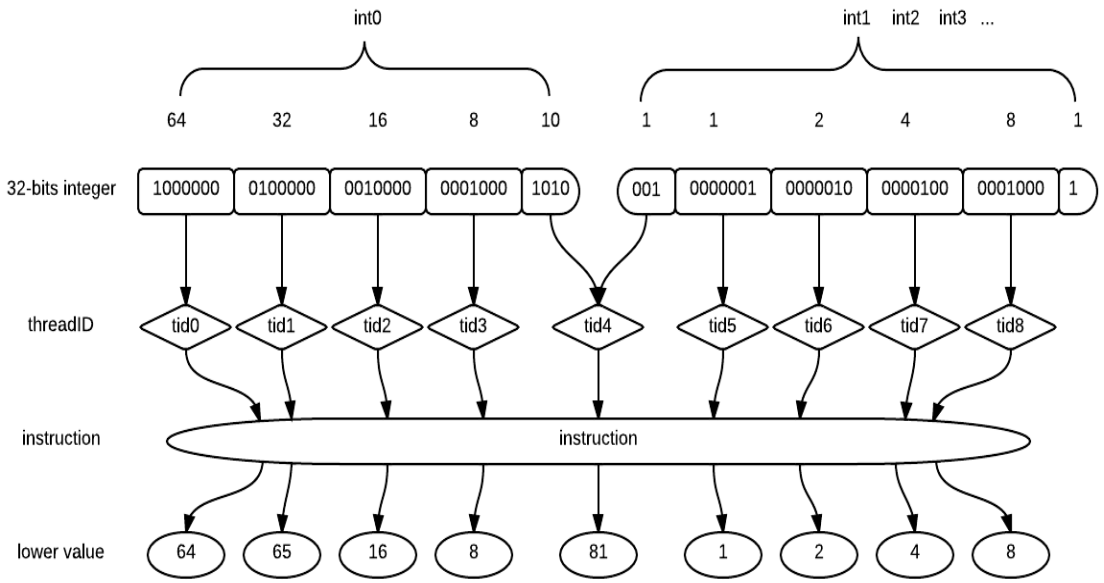


Figure 5: An example of GPU-based LBA decompression for a *common* b. In this figure, *b = 7*; thus in each encoded integer, there are four lower values and several offsets that belong to another lower value. In *int0*, the first *28* bits containing four integers, while the last *(32 – 28) = 4* bits belong to the next integer whose first *four* bits are stored in *int0* and last *three* bits stored in are *int1*. After decompressing this value that resides between two integers, *int2* is considered to start from its *29th* bit. As with *int0*, there are *four* lower values consuming *28* bits in *int2*, then the last *one* bit in *int2* belongs to the next integer with *one* bit stored in int2 and *six* bits stored in int3 (not shown).

For all *common b,* there must be offsets in all encoded integers, either the first few bits or the last few bits, or both. Given a *tid*, the corresponding *b* bits need to be found to decompress the lower bits value. The pseudocode of how to find the corresponding *b* bits is provided below.

DECOMPRESS-LOWER-BITS-ARRAY(A)

for all *tid* ◄– 0 **to** *length (A)*
    mask ◄– *(1 << bw) -1*
    index ◄–*(bw * tid) >> 5*
    position ◄– *32 - bw - ((tid * bw) & 31)*
        if (*offset >= 0*)

```
    output[tid] ← (A[index] >> position) & mask
else
    offset ← -position
    output[tid] ← ((A[index] << offset)|(A[index +1] >> (32 - offset))) & mask
```

By applying such codes on all threads, all threads are guaranteed to run the same sequence of operations on their corresponding *b* bits simultaneously. However, the operations for a *common b* are much more complicated than those for a *special b*. This means that the decompression speed for *common b* might be slightly slow than the decompression speed for a *special b*. The comparison for the decompression speed of different bit-widths can be seen in Figure 6, which shows that in fact the slow down for *common b* is minor.

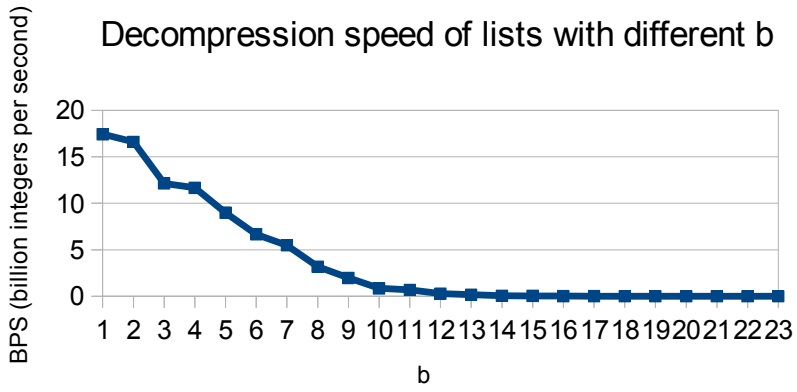


Figure 6: Decompression Speed for Lists with Different b

In real datasets, longer lists have a relatively smaller *b*. Thus if the list is longer, a higher degree of parallelism can be achieved on the GPU. In Figure 6, it is apparent that the decompression speed for longer lists is much faster.

5.2 GPU-based decompression for the UBA

Recall that the UBA is packed before storing it. Thus it seems necessary to unpack the integers

in UBA into unary bits. A naive method to decompress the UBA can be performed in three steps:

1. Unpack the UBA. The UBA can be unpacked by assigning only one bit to each thread. This means that each thread needs to perform a shift operation based on the value of its *tid* and a mask operation. The above result is stored using another array, called unary bits array[2]. In this step, for every integer in UBA we need 32 threads to perform the unpack operation. In addition, since every integer is split into 32 bits (note that every bit is now a 32-bit integer), the final unary bits array is 32 times bigger than the original UBA. For example, given a UBA of length l_u *words*, *32** l_u threads are allocated to unpack the UBA, and the final unpacked result consumes 32*4* l_u bytes on the disk.

2. Parallel prefix sum over the unpacked unary bits array from Step 1. There are two types of prefix sum: inclusive prefix sum and exclusive prefix sum [5]. In an inclusive prefix sum, given an index, all numbers before the current index including the number of the current index are summed up as the result of current index. In an exclusive prefix sum, the number of the current index is not included and the first number in the result array is set to 0. In this thesis, all prefix sum operations are exclusive. In this step, the CUDPP library is called to perform the parallel prefix sum. The CUDPP automatically allocates threads and optimize the running process according to the size of input array. More details can be found in [11].

3. Gather the specific index from the summed-up array in step 2 to fetch the final result. This step is performed based on the result of the previous two steps. In the unary code, "1" means that we should increase the value of the number by one, and "0" means that this is the end of a number, called stop bit. Thus the indexes of "0" in the unary bits array are the same as the indexes of original unary values in the prefix sum results. This means the final unary results can be gathered by looking up the index of "0" bits in the unary bits array, and reading the corresponding

2 Here unary bits array is not the same with UBA. In unary bits array, every bit in the UBA is an integer.

index in the prefix sum results. When implementing this on the GPU, every thread works on only one bit. Threads read the unary bits array, and if the value in the unary bits array is “0”, they write the corresponding number in the prefix sum array to the result array corresponding to the index and their *tid;* else, if the value is “1”, the further work is canceled. The GPU-based pseudocode for the gather operation can be seen in Figure 7.

```
GATHER(unary_bits_array, prefix_sum, result)
for all tid 0 ← length[unary_bits_array]
  if (tid == 0)
    result[prefix_sum[tid] – tid] ← prefix_sum[tid]
  else
    return null
```

An example of prefix sum and gather can be seen in Figure 9.

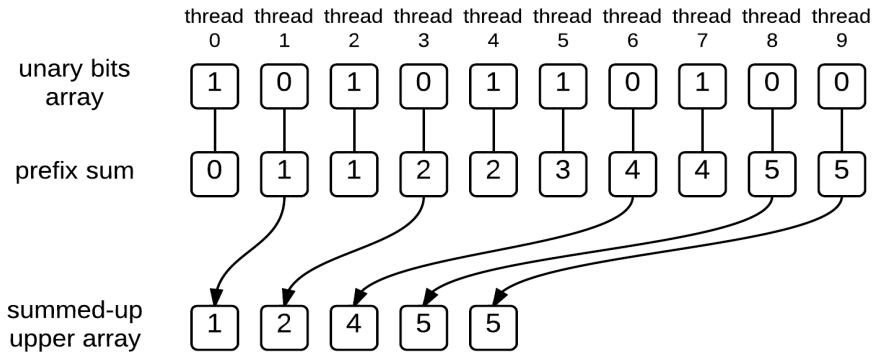


Figure 7: An example of prefix sum and gather. In this example, the unary bit values of *thread1, thread3, thread6, thread8, and thread9* are *0*. Thus, these threads write the corresponding values in the prefix sum array to the summed-up UBA with indexes equal to their threadIDs minus the corresponding values in the prefix sum array.

The cost of each step in this naive decompress method can be seen in Figure 8.

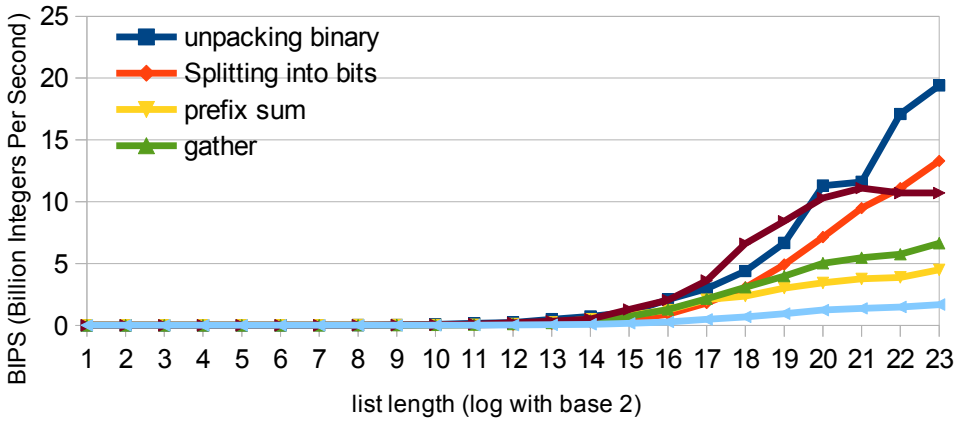


Figure 8: The Decompression Speed of the Naive Method.

In the naive approach, the UBA is expanded into a unary bits array that is 32 times larger than the UBA; as a result, the cost of parallel prefix sum over this large array is expensive. The performance of this naive method is not so attractive. We now try to optimize the naive method by decreasing the size of array that we need to perform prefix sum operations on. In the optimized method, a block structure is applied to achieve this goal. In the block structure, a fixed block size *m* is applied to the UBA. Then for each block, the number of unary "1"s is counted and stored in another array. Later, a prefix sum operation is performed over this array. Recall that in an exclusive prefix sum is that the current summed up value is the summation of all previous initial values. This means that after this prefix sum operation, the start value of each block can be abtained. Finally, another prefix sum operation is performed within each block to retrieve the final prefix sum result, which is the same as the result after step 2 in the naive method. In this optimized method, two prefix sum operations are performed, but each time the size of the array is much smaller than in the naive method. Thus, the overall decompression time is reduced. An example of this optimized method can be seen in Figure 11.

In Figure 9, the size of the block is 32, which means that each integer in the UBA is considered

to be one block. After counting how many unary "1"s are existing within each block, a prefix sum operation is performed over this result. The result after this prefix sum operation is the start value of each block. Then the second prefix sum operation is performed within each 32-bits integer. In our implementation, we choose 32 as the block size, but other block sizes also work.

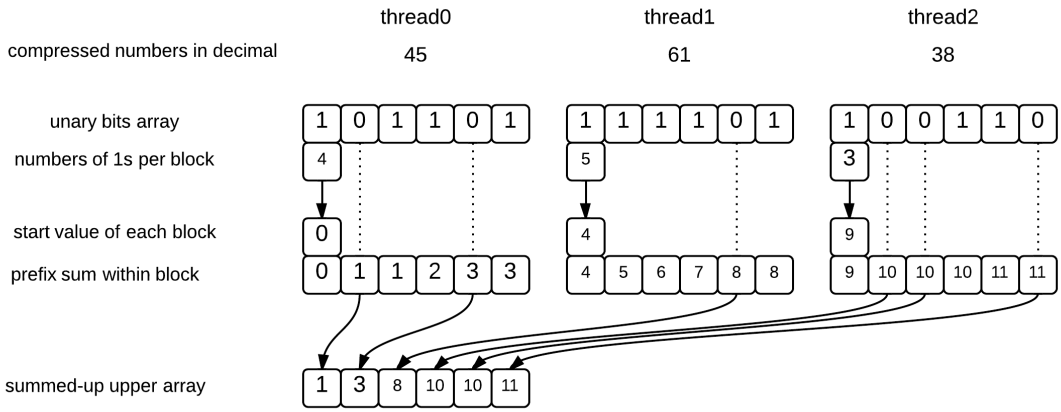
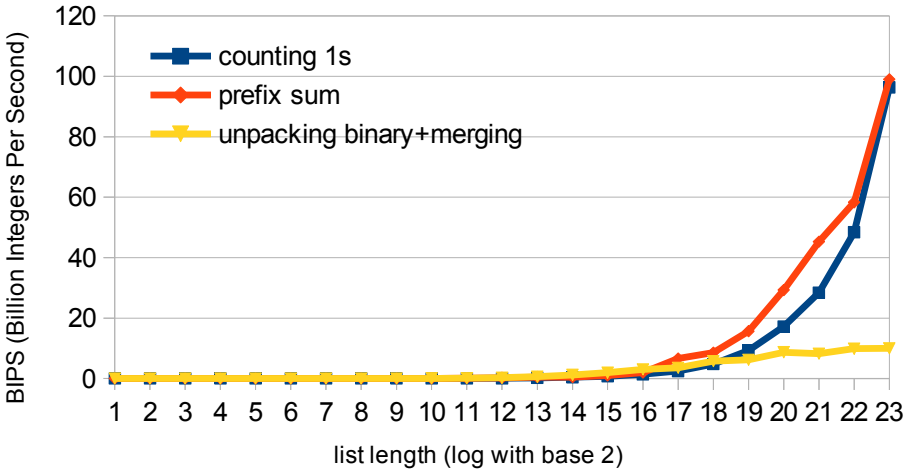


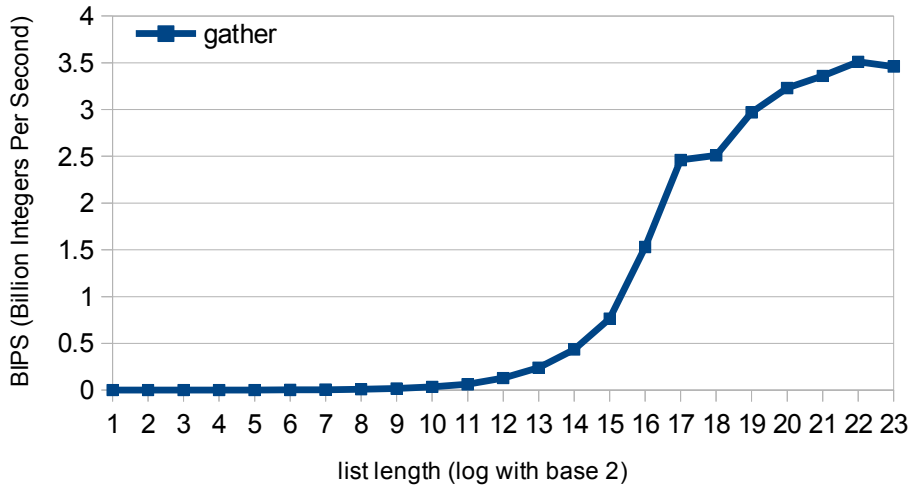
Figure 9: An example of block-based UBA decompression. For simplicity, the block size in this example is *six* rather than *32*. During the decompression, *thread0, thread1, and thread2* first count how many "*1"*s there are in their blocks, then the CUDPP library is called to perform a prefix sum operation over these counts. The result of prefix sum *(0, 4, 9)* represents the start value of each block. After prefix sum, each thread preform a local prefix sum operation within its block; afterwards, the thread checks the corresponding value in the unary bits array; if the value is *0,* the prefix sum result is written to the summed-up upper array.

To optimize the implementation and threads alignment, for all steps in this method, each thread is allocated one 32-bit integer, instead of one bit as in the naive method. This means that the numbers of threads we need in this step is the same as the number of elements in the UBA. Threads first count how many unary "1"s are existing within their allocated integers. Then the CUDPP library is called to perform the parallel prefix sum operation. The next prefix sum is directly performed over the arrays of length 32 each, and the CUDPP library is not needed here any more. We choose to perform this prefix sum using one thread per integer. The final gather

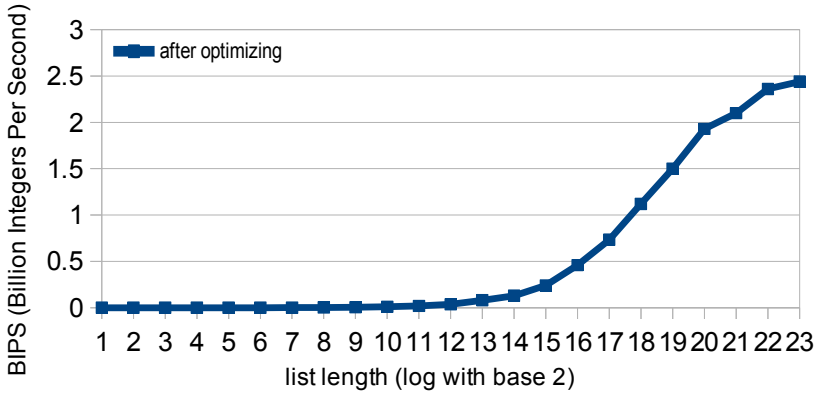
operation is still the same as the one in the naive method. After this optimization, the cost of each step in this optimized method can be found in Figure 10.



10.1 Decompression speed for counting 1s, prefix sum, and unpacking binary plus merging the binary and unary result.



10.2 Decompression speed for the gather operation.



10.3 Overall decompression speed after optimizing

Figure 10: The decompression of each steps after optimizing.

After the above optimization, we further optimized the overall performance by merging different steps that use same numbers of threads together. This optimization can decrease the times that kernel functions are called. Since the calling of kernel functions consume some time; and some intermediate results do not need to be created and write to separate arrays. This further optimization makes the overall performance even better. In the implementation, the unary bits array is not generated; it is performed within the gather step. This means that each thread needs to perform 32 times shift and mask operations; and if the masked result is "1", it performs the prefix sum operation; else if the result is "0", it performs the prefix sum and the gather operation as well.

Finally, after decompressing the LBA and the UBA. The initial values can be computed by shift the numbers in the UBA b bits and add the numbers in the LBA to their corresponding shifted numbers in the UBA. Since the length of both LBA and UBA are the length of the inverted list l; the final step needs l threads to accomplish. Recall that l threads are needed to decompress the LBA where each thread work on one specific lower value. Thus the decompression of the LBA and the computation of the final results can be further merged in one

function where each thread first computes its lower value and later access the corresponding number in the UBA to compute the final result. The performance of before and after this merge can be seen in Figure 11.

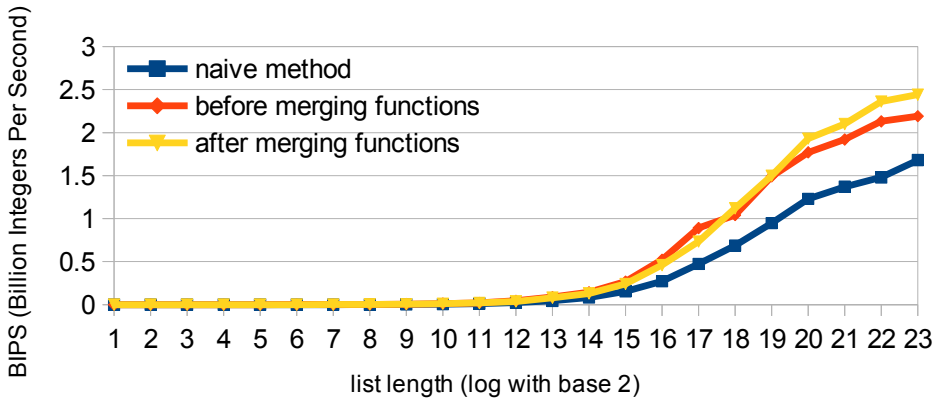


Figure 11: Comparison before and after optimizing.

Chapter 6 Experimental Results

In this chapter, we compare the Parallel-QS algorithm to other compression algorithms. We provide the experimental results from different aspects. We find that the decompression with the Parallel-QS algorithm is faster than with standard PForDelta algorithm, and it also achieves a better compression ratio.

In our experiment, the dataset we use is TREC-GOV2, which contains 25 million documents. The queries we use are 100K random queries for efficiency test for 2006 TREC-GOVE2 dataset. The GPU we use is the NVIDIA GTX 680 graphics processor, which has 2047MB global memory and a 1.12GHz GPU clock rate. The CPU we use is the Intel (R) Xeon (R) E5-2620, which has a 2.0GHz clock rate.

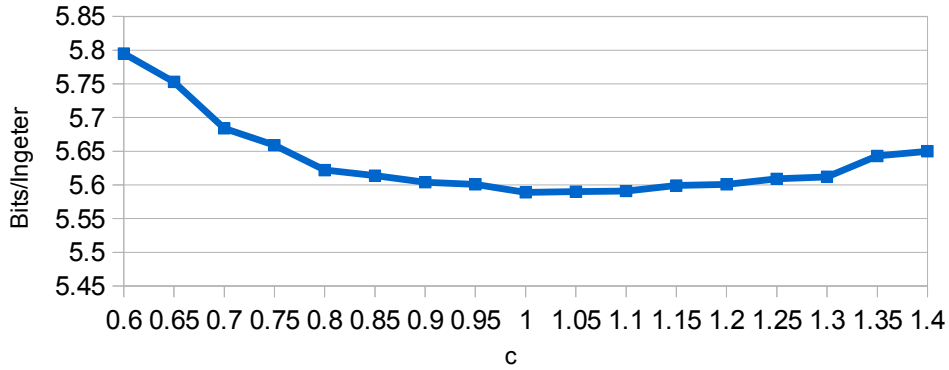


Figure 12: Compressed size in bits per integer for different values of the parameter *c* used to

determine number of bits in binary part.

6.1 Comparison of Compression Ratio

Figure 12 shows the compressed size of the QS with different values of the parameter *c* (*c* is the parameter to determine the value of *b,* which can be found in the previous chapter). Different values of the parameter *c* will lead to different values of *b,* and thus change the length of the UBA and the LBA. This will then change the compressed size and decompression speed. We know that theoretically the compression ratio is the best when *c* equals to *1.0.* As we can see in Figure 12, this is in fact the case on our data. The compression ratio is worse when *c* is larger or smaller than *1.0.* However, the number of bits used to represent a integer increases much more rapidly when *c* gets smaller than when *c* gets larger, but not much in general. This is because of the trade-off between unary code and binary code, it might consumes more bits to represent a number using unary codes than binary codes. Figure 13 shows the decompression speed related to different values of the parameter *c.*

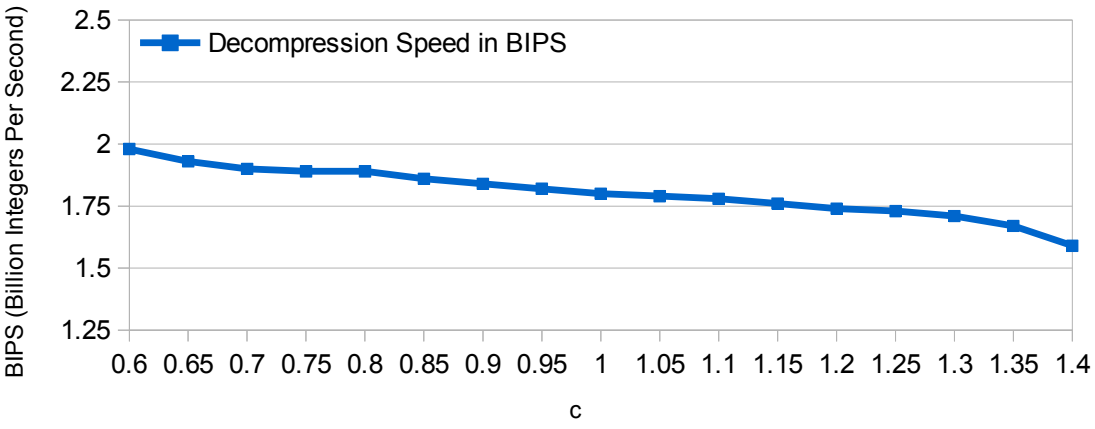


Figure 13: Decompression speed in billion integers per second for different values of the parameter.

Please note that for all other experiments in this thesis, we use *1* as the value of the parameter *c*. Table 1 shows the compressed size between different compression algorithms. We can see that out Parallel-QS has the same compression ratio with Rice, and it is much better than the PForDelta algorithm.

Algorithm	Rice	PForDelta	Parallel-QS
Bits/Integer	3.87	7.63	3.13

Table 1: Compressed size in bits per integer for different compression algorithms.

6.2 Comparison of Decompression Speed

The decompression speeds for different list length under Parallel-QS were already provided in the previous chapter. Here we provide a comparison of speeds between Parallel-QS and other methods. In Table 2 we can see that our Parallel-QS algorithm is much faster than the PForDelta algorithm and the Rice algorithm. This means that our Parallel-QS has a compression ratio that almost the same with Rice algorithm without sacrificing the decompression speed.

Algorithm	Rice (CPU)	PForDelta (CPU)	QS (CPU)	Parallel-QS (GPU)
Speed	0.3	1.3	0.9[3]	1.8

Table 2: Average decompression speed of different algorithms in billion integers per second, based on a 100K query log. The PforDelta algorithm is the version in [23]. And for the implementation of Rice and PForDelta, please see [24].

6.3 Impact of Thread Alignment

3. We did not implement the decompression for QS on the CPU. This number is cited from [18]; for the hardware information and other details, please see [18].

The GTX-680 we use in the experiments has a maximum block size (number of threads per block) of 1024. However, we want to find the best block size. We tried three different block sizes: 256, 512, and 1024. Figure 15 shows the decompression speed for these different thread alignments. In Figure 15, we can see that for shorter lists, say lists have docIDs fewer than 2^{15}, the decompression speeds with block size of 256 are faster than those with block sizes of 512 or 1024. This is because when the lists are shorter, a larger block size would have more wasted threads, which consume time to synchronize. However, as the lists become longer, decompressing the lists using a larger block is faster than using a smaller block.

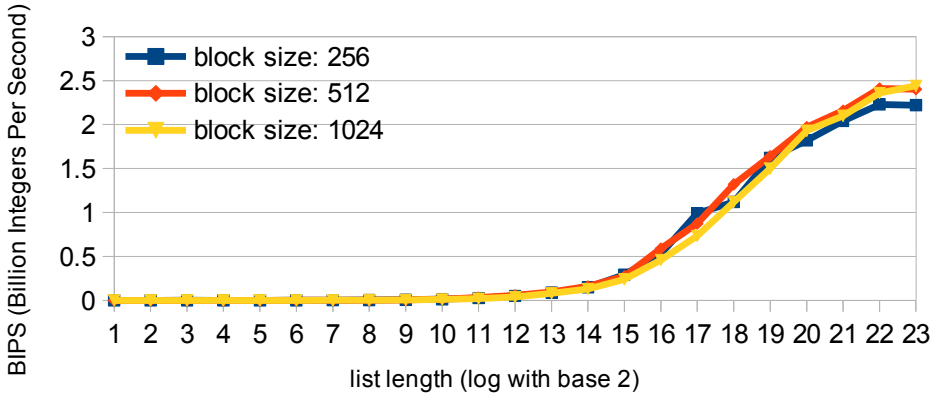


Figure 14: Decompression speed for different block size (number of threads per block).

Chapter 7 Conclusions

In this thesis, we studied several compression algorithms for inverted index structures. We noticed that the QS algorithm by S. Vigna has a relative high compression ratio, and has the potential to be efficiently parallelized on the GPU. To perform this decompression through QS using the GPU, we carefully modified the original decompression procedure, and proposed a naive GPU-based Parallel-QS decompression method. To further optimize the performance, we created a block-based prefix sum structure. This block-based structure allows us to perform the prefix sum operation over a smaller array than in the naive method over a larger array, which accelerates the decompression. In addition, we merged some kernel functions to decrease the calling times of kernel functions which also speeds up the overall decompression.

For the future improvements, some optimizations could be made for the gather operation, which consumes 70% of the decompression time. Other possible future work including query processing on the GPU could also be interesting. Although some of them have already been implemented by researchers in [6, 22], we believe that there is still a lot of space for the performance to be improved.

Biblioaraphy

[1] Nvidia CUDA programming guide

http://www.nvidia.com/object/cuda_home_new.html

[2] Nvidia CUDA homepage.

http://www.nvidia.com/object/cuda_home_new.html

[3] Nvidia GTX 680 Kepler Whitepaper, 2012

[4] R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval. *New York: ACM Press,* 1999.

[5] G. Blelloch. *Prefix sums and their applications,* In John H. Reif (Ed.), *Synthesis of Parallel Algorithms,* Morgan Kaufmann,1990.

[6] S. Ding, J. He, H. Yan and T. Suel. Using Graphics Processors for High Performance IR query Processing. In *Proc. of the 18th International World Wide Web Conference,* 2009.

[7] P. Elias. Efficient storage and retrieval by content and address of static files. *J. Assoc. Compu. Mach.,* 21(2):246-260, 1974.

[8] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasor: high performance graphics co-processor sorting for large database management. In *Proc. of the ACM SIGMOD,* 2006.

[9] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of databse operations using praphics processors. In *Proc. of the International Conference On Computer Graphics and Interactive Techniques,* 2005.

[10] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 2007.

[11] M. Harris et al. CUDPP: CUDA data parallel primitives library. *http://www.gpgpu, org/developer/cudapp,* 2007.

[12] B. He, K. Yang, R. Fang, M.Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proc. of the ACM SIGMOD International Conference,* 2008.

[13] S. Heman. Super-scalar database compression between RAM and CPU-cache. *MS Thesis*, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands, 2005

[14] D. Lemire, L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience,* 2013.

[15] C. Manning, P. Raghavan, H. Schuetze. Introduction to Information Retrieval. *Cambridge University Press,* 2008.

[16] A. Moffat, and T. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. *Morgan Kaufmann.* 1999.

[17] H. Yan, S. Ding, T. Suel Inverted index compression and query processing with optimized document ordering. In *Proc. of the 18th International World Wide Web Conference,* April 2009.

[18] Sebastiano Vigna. Quasi-Succinct Indices. In *Proc. of the Sixth ACM International Conference on Web Search and Data Mining*, 2013.

[19] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines.In *Proc. of the 17th International World Wide Web Conference,* April 2008.

[20] J. Zobel and A. Moffat. Inverted files for text serach engines. *ACM Computing Surveys, 38(2)*, 2006.

[21] M. Zukowski, S. Heman, N, New, and P. Boncz. Super-scalar RAM-CPU cache

compression. In *Proc. Of the Int. Conf. On Data Engineering*, 2006.

[22] N. Ao, et al. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *In Proc. of the VLDB Endowment 4.8 (2011): 470-481*, 2011.

[23] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. *In Proc. of the 17th international conference on World Wide Web. ACM,* 2008.

[24] R. Khmelichek. Developer of a high performance search engine toolkit. *Master thesis*, Polytechnic Institute of New York University, 2010.