Published in final edited form as:

ACM SIGAPP Appl Comput Rev. 2017 June ; 17(2): 27–35. doi:10.1145/3131080.3131083.

Large Scale Document Inversion using a Multi-threaded Computing System

Sungbo Jung,

University of Louisville, Computer Engineering and Computer Science, Louisville, KY 40292,
1-502-852-0467

Dar-Jen Chang, and

University of Louisville, Computer Engineering and Computer Science, Louisville, KY 40292,
1-502-852-0472

Juw Won Park

University of Louisville, Computer Engineering and Computer Science, KBRIN Bioinformatics
Core Louisville, KY 40292, 1-502-852-6307

Abstract

Current microprocessor architecture is moving towards multi-core/multi-threaded systems. This trend has led to a surge of interest in using multi-threaded computing devices, such as the Graphics Processing Unit (GPU), for general purpose computing. We can utilize the GPU in computation as a massive parallel coprocessor because the GPU consists of multiple cores. The GPU is also an affordable, attractive, and user-programmable commodity. Nowadays a lot of information has been flooded into the digital domain around the world. Huge volume of data, such as digital libraries, social networking services, e-commerce product data, and reviews, etc., is produced or collected every moment with dramatic growth in size. Although the inverted index is a useful data structure that can be used for full text searches or document retrieval, a large number of documents will require a tremendous amount of time to create the index. The performance of document inversion can be improved by multi-thread or multi-core GPU. Our approach is to implement a linear-time, hash-based, single program multiple data (SPMD), document inversion algorithm on the NVIDIA GPU/CUDA programming platform utilizing the huge computational power of the GPU, to develop high performance solutions for document indexing. Our proposed parallel document inversion system shows 2-3 times faster performance than a sequential system on two different test datasets from PubMed abstract and e-commerce product reviews.

CCS Concepts—Information systems→Information retrieval • Computing methodologies→Massively parallel and high-performance simulations.

Keywords

Graphics Processing Unit; GPU; high-performance computing; document inversion

1. Introduction

The rapid growth of network and computer technology leads to the immense production of information globally. This has resulted in a large number of text data in the digital domain,

leading to the need for more efficient methods for searching databases. For example, PubMed, one of the largest collections of bioinformatics and biomedical publications, currently has over 26 million articles, and over 1,000,000 articles are added annually. With databases of this size, researchers must rely on automatic information retrieval systems that can quickly find articles relevant to their work. The demand for fast information retrieval has motivated the development of efficient indexing and searching techniques for any kind of data expressed as text.

An index is a list of important terms, including the locations where those terms appear in a document. If the document is divided into pages, the index also provides page numbers and line numbers for the terms. An index facilitates content-based searching by locating the topic word or phrase in the index and returning the corresponding locations. Thus, indexing plays a very important role in searching and retrieving documents. During the past decade, indexing is regarded an excellent text processing technique. An index links each term in a document to the list of all its occurrences. It allows efficient retrieval of all the occurrences of a term. This method usually requires a large amount of space, which can be up to several times the size of the original text. It also requires space for storing the text itself, because the text is hardly reproduced by the index. The process of generating an index of a document is called document inversion to emphasize that an index is the inverse function of a document. It transforms the sequence of symbols in the document into an index. A document can be construed as a mapping from the positions in the document to the terms that appear there.

Recently a trend in microprocessor design is to include multiple cores in one chip. Because the heat generated in a microprocessor is roughly quadratic of the clock rate, it becomes a barrier to increasing processor speed. With improvement of manufacturing technology, a processor can be made with smaller size circuits, from 90nm to 65nm down to 14nm, so a processor chipset can hold more circuits in the same size die. For these reasons, most of microprocessor vendors, such as Intel, AMD, and IBM, turned to multi-core architecture instead of high clock-speed single-core processors. The multi-core trend affects not only CPU chipset, but also graphics processing units. NVIDIA, one of the major GPU chipset vendors, launched a multi-core GPU chipset called G80 about 10 years ago. It is equipped with 128 streaming processors with 768MB DDR3 memory. Currently, the top of the line NVIDIA chipsets (GP100) have 3584 streaming processors with 16GB HBM2 memory [5]. These NVIDIA GPU chipsets support Computing Unified Device Architecture (CUDA), which is an extended C language environment, to implement general purpose applications in the GPU. CUDA simplifies the development process to such a degree that many researchers now convert massive computation problems to the GPU platform. Many applications, such as molecular dynamics, Nbody from physics, and the DNA-folding in bioinformatics, show a dramatic speed-up in GPU solutions.

2. Related Work

Some research in the literature demonstrates that multi-core platforms achieve good performance for document indexing. A. Narang, *et al.* [12], uses a distributed indexing algorithm on IBM BlueGene/L supercomputer platform. Their system focused on high-throughput data handling, real-time scalability with increased data size, indexing latency,

and distributed search across 8 to 512 nodes with 2GB-8GB size data. It shows 3-7 times improvement in indexing throughput and 10 times better indexing latency. D. P. Scarpazza [17] implemented a document inversion algorithm on the IBM cell broadband engine blade, which has 18 processing cores (16 synergistic processor elements and 2 power processor elements). He adopted a single instruction multiple data (SIMD) blocked hash-based inversion (BHBI) algorithm for the IBM cell blade. It is 200 times faster than the single pass in-memory indexing algorithm. Another scalable index construction approach was performed on multi-core CPUs by H. Yamada and M. Toyama [21]. They implemented multiple in-memory index and on-disk merging methods on two quad-core Xeon CPU equipped system with 30GB web-based document collection. N. Sophoclis, *et al.* proposed an Arabic language indexing approach on a GPU with OpenCL [20]. Their experiment shows that a GPU Arabic indexer is 2.5 times faster than CPU one on overall performance. M. Frumkin presented a real time indexing method on a GPU [4]. The presenter uses Tokenizer(Sequential), Splitter, BucketSort, and Reduce implementation on a GPU. The system was tested with 4200 documents in a literature collection and 7M documents of Wikipedia web data, and was 3.1 times faster than a CPU searching the literature collection and 2.2 times faster in searching the Wikipedia data.

3. GPU and CUDA

3.1 CUDA Programming

NVIDIA CUDA is a general purpose scalable parallelized programming model for highly parallel processing applications [13]. It is an abstract high-level language that has distinctive abstractions, such as a hierarchy of thread blocks, barrier synchronization, and shared memory structures. CUDA is well-suited for programming on multiple threaded multi-core GPUs. Many researchers and developers attempt to use CUDA for demanding computational applications in order to achieve dramatic improvements in speed. In earlier GPGPU designs, general purpose applications on the GPU must be mapped through the graphics Application Programming Interface (API) because traditional GPUs had highly specialized pipeline designs. This structural property made it necessary for a programmer to write the programs to fit the graphics API. Sometimes the programmer needed to rework all the programs. The GPU has a global memory that can be addressed directly from the multiple sets of processor cores. The global memory makes the GPU architecture a more flexible and general programming model than previous GPGPU models. The global memory also allows programmers to implement data-parallelized kernels for the GPU easily. A processor core in the GPU can share the same traits with other processor cores. Thus, multiple processor cores can run independent threads in parallel at the same time.

3.2 SPMD Design

The GPGPU application systems use the GPU as a group of fast multiple coprocessors that execute data-parallelized kernel code, allowing the programmers to access the GPU cores via a single source code encompassing both CPU and GPU code. A kernel function operates in a Single Program Multiple Data (SPMD) fashion [1]. The SPMD concept extends Single Instruction Multiple Data (SIMD) by executing several instructions for each piece of data. A kernel function can be executed by the threads in order to run the data-parallelized

operations. It is very efficient to utilize many threads in one operation. For full utilization of the GPU, finegrained decomposition of work is required, which might cause redundant instructions in the threads. However, there are several restrictions in using the kernel functions. A CUDA kernel function cannot be recursive and it cannot use static variables. Kernel functions also need a non-variable type of parameters. The host (CPU) code can copy data between the CPUs memory and the GPUs global memory via API calls.

3.3 Thread, Block, and Grid

Thread execution on the GPU architecture consists of a three-level hierarchy, grid, block, and thread. The grid is the highest level. A block is a part of the grid, and there can be a maximum of $216 - 1$ blocks in the grid, organized in a one or two dimensional array. A thread is a part of a block, and there can be up to 512 threads in a block, organized in a one, two, or three dimensional array. Threads and blocks have their unique location numbers as threadID and blockID. The threads in the same block share the data through the shared memory. In CUDA, the function syncthreads() performs the barrier synchronization, which is the only synchronization method in CUDA. Additionally, the threads can be grouped into warps of up to 32 threads.

3.4 CUDA Memory Architecture

The GPU has several specially designed types of memory that have different latencies and different limitations [8]. The registers are fast and very limited size read-write per-thread memory in each SP. The local memory is slow, not cached, limited size read-write per-thread memory. The shared memory is a low-latency, fast, very limited size, read-write per-block memory in each SM. Shared memory is useful for data among the threads in a block. The global memory is a large, long-latency, slow, non-cached, read/write per-grid memory. It is used for communication between CPU and GPU as a default storage location. Constant and texture memory is used for only one-sided communication. The constant memory is slow, cached, limited size, read-only per-grid memory. The texture memory is slow, cached, large size, read-only per-grid memory. Table 1 indicates the CUDA memory types [16].

3.5 CUDA Toolkit and SDK

NVIDIA provides the CUDA toolkit and CUDA SDK as the interface and the examples for the CUDA programming environment. They are supported on Windows XP, Windows Vista, Mac OS X, and Linux platforms. The CUDA toolkit contains several libraries and the CUDA compiler NVCC. A CUDA program is written as a C/C++ program. NVCC separates the code for the host (a regular C/C++ code) and the device (a CUDA native code). The CUDA toolkit also supports a simulation mode of CUDA which allows for debugging. The CUDA toolkit and SDK also provide GPU management functions, memory management functions, external graphical API supported functions, and some useful sample codes.

4. Document Inversion

4.1 Documents as Data

Documents come in a variety of forms. They may contain not only texts and graphics, but also audio, video, and multi-media data. In information retrieval, traditionally, information

of a document can be represented by words or phrases. A word is the minimum component of a document, and it can be used for query and retrieval. A list of words is the simplistic representation of a document.

4.2 Text Preprocessing

Currently, various document file formats are used in the digital environment. From a simple text file with an extension .txt to a more complicated binary file with an extension .pdf, several document formats are used by different applications: such as simple text editors, Microsoft Word, Adobe Acrobat, web browser, and so on. These document formats have their own structures. In order to create the index from a document file, extracting text contents is required. Each document format requires a different extraction method. Before indexing, the extracted texts are processed to isolate the words or the terms. The term extraction process includes four main stages: tokenization, term generation, stopword removal, and stemming (See Figure 1).

4.3 Inverted Index Construction

In order to efficiently search and retrieve documents matching a query from the document collection, the list of terms and their postings are converted to an inverted index. There are many algorithms that construct an inverted index, such as sort-based inversion, memory-based inversion, and block-based inversion. Their performance varies by memory usage, storage types, processor types, and the number of passes through the data. Conventional document inversion algorithms process a document collection in the system memory. Both the document collection and the resulting dictionary are stored in the system memory. Thus the size of the document collection that can be processed by a conventional algorithm is restricted by the size of the system memory. If the document collection is too large to fit into the memory at once, document inversion cannot be performed with these methods. To deal with large document collections, contemporary algorithms process the documents in blocks of manageable sizes.

4.3.1 Blocked Sort-based Inversion—Blocked sort-based inversion (BSBI) addresses the issue of insufficient system memory for a large document collection. It maps a term to a unique identifier (termID) so that the data to be sorted later have uniform sizes. Furthermore, it uses external storage to store the intermediate results. This inversion algorithm is a variant of sort-based inversion. To invert a large document collection, the collection is divided into blocks of equal sizes. The blocks are inverted individually. After the termIDs and the associated postings are extracted from the block, the algorithm sorts the list of termIDs and postings in the system memory. Then, it outputs the sorted list of the block to the external storage. After all blocks are individually inverted, the sorted lists of all the blocks are merged and sorted into a final inverted index. The algorithm shows excellent scaling properties for increasing sizes of the document collections, and the time complexity is $O(n \log n)$ [7]. However, blocked sort-based inversion is not suitable for a multi-core system with small local memory for each core, because the algorithm needs to maintain a large table that maps terms to termIDs.

4.3.2 Single-pass in Memory Inversion—For a very large document collection, the data structure, used in term-termID mapping, of the BSBI algorithm does not fit the system memory. Single-pass in memory inversion (SPIMI) is a more scalable algorithm to solve this problem by storing the terms directly rather than storing the termIDs and maintaining a huge table of the term-termID mapping [7]. Because it uses the dynamic allocation of memory when processing a block of documents, the sizes of terms and the associated posting lists can be expanded dynamically. When terms and postings are extracted from a document, the SPIMI algorithm adds the term and posting into the memory directly without sorting. The first occurrence of a term is added to the dictionary, and a new posting list is created. If multiple occurrences of a term appear, only the postings are added to the posting list. Often a hash table is used to store the terms. Specifically, a term is associated with a dynamic linked list of postings. Because the space for posting lists is dynamic, if the space is full, SPIMI can double the space of the posting lists. If the memory for the hash table is full, SPIMI saves its output block to an external storage and starts a new empty block. Once all the blocks are processed, they are merged and sorted into a final inverted index. Because sorting is not required during the first pass, this algorithm is faster than the BSBI algorithm.

4.3.3 Blocked Hash-based Inversion—Scarpazza [17] proposed the blocked hash-based inversion (BHBI) algorithm on the multi-core processor, the IBM cell broadband engine, for document inversion. The BHBI algorithm is proposed to solve several issues of the BSBI and the SPIMI with multi-core processor platforms. In a multi-core processor, local memory and registers of each processor are very useful for computation because they have very low access latency. Although they are very fast, the sizes of local memory and registers are too small to store large document collection. Thus, the BHBI is optimized for using small local memory in a multi-core processor system. To avoid dynamic allocation and overflows, input and output buffers for a block are set to the size of local memory. After reading a term from the input stream, instead of using the term directly in a dictionary, the BHBI uses a hash value of the term as its identifier. The term itself is not stored at all. Thus BHBI hashes the term, and adds to the output block an entry of the hash value, a document identifier, and a location. Multiple occurrences of a term will result in multiple entries in the output block. If the output block is full, all the entries in the block are sorted by their hash values. Then BHBI writes the output block to the global memory. After finishing all the blocks, BHBI performs a global merge sort that combines all the index blocks together in a global index. It is important to note that there is no hash table in BHBI. The hash function is used to generate identifiers of a uniform width. The width of the hash values must be chosen carefully. Because the hash values are used as the unique identifiers in the output block, a collision of hash values will lead to mistaking two different terms as the same term.

4.4 Sequential Document Inversion

The document inversion system consists of five components: input module, tokenizer, stopword remover, stemmer, and index maker. First, there are several file formats determined by the applications that created the files. Pure text files are preferred in text processing due to easy handling. However, most of the journal papers and articles are available in the PDF, PS, RTF, and DOC formats, or various marked-up file formats. In addition, these document files contain not only texts but also figures, tables, and additional structure information. So

the function of the input module is to extract the texts in a document. At this stage, all the documents are converted into texts using file converters, and unnecessary structures and additional information are ignored. Second, after obtaining pure texts from the documents, the system performs tokenization. The tokenizer reads a document sequentially, and it separates a word, the minimum unit of the document, as a token from the input stream. If the input stream contains a blank space or a line separator, the tokenizer creates a new token after that point. In addition, the system keeps track of the location where the token occurs, and this information will be used later in document inversion. Third, after obtaining a token, the tokenizer passes it to the stopword remover. The stopword remover eliminates all stopwords in the stopword list. The stopword list can have only one word per line with no leading or trailing spaces. Any character after the first word on a line is ignored. The apostrophe characters (') are stripped out, and letters before and after an apostrophe are concatenated. The underscore (_) and pound sign (#) characters are treated as normal characters. Hyphen (-), semicolon (;), and colon (:) characters are not allowed. These characters have the effect of breaking words, which is equivalent to two words on the same line. A minimal perfect hash function for stopwords is generated by the minimal perfect hash function generator, implemented by Zbigniew J. Czech [3]. It produces a minimal perfect hash function from the selected 641 stopwords. After a token is found by the tokenizer, the token is hashed with the minimal perfect hash function of stopwords. If a token is matched to a stopword, the token is discarded. If the token is not a stopword, it is passed to the stemming code. Fourth, the stemming algorithm creates the base (or stem) of each term. The system uses the Porter stemming algorithm [15] for a stemmer. In this stage, a stemmer removes the prefixes or suffixes by the defined rules in order to restore the bases. Lastly, after passing all the steps above, the refined terms, which will be used in indexing, are stored into the hash table. The system uses the separate chaining type hash table. The base of a term is used as a key in the hash table, and the value of docID and location, called a posting, is stored in the hash table. When the memory space for the index is full, the index is stored as a block file. Then the indexing system starts a new block. After finishing all the documents, all the index block files are merged and sorted using merge sort by the keys. The finalized index can be stored as the bag of words format, which contains a pair of the dictionary file and the index file.

4.5 GPU Document Inversion

There are two main design issues of GPU computing. One issue is to design the thread-blocks and how to feed the data to the thread-blocks efficiently. Each document can be processed with one thread, one block of threads, or one grid of blocks of threads by the GPU. The thread-block design follows the SPMD programming. The other issue is to use the GPU memory (global memory, constant memory, and shared memory) efficiently.

4.5.1 Thread Design—The sizes of documents, the numbers of terms in them, and the lengths of terms, are all different. The numbers of stopwords in the documents are also different. These variations make it difficult to predict the number of operations and the size of storage, and to divide the data in the SPMD programming design. After reading the documents from external storage, the documents are copied to the GPU global memory, which is large but slow. The shared memory has fast access cycles and all the threads in the

same block can access the shared memory. The CUDA architecture gives a programmer the full flexibility to control the use of shared memory. However, the size of shared memory in each streaming multiprocessor (SM) of the CUDA device is 16KB, which is fairly small to store common data structures. Therefore, adequate distribution of data is required to use full benefits from the limited size of shared memory. During the preliminary study, the abstracts/reviews as document are found have fewer than 4,000 characters, which is less than 4KB using the ASCII code. Thus it is proposed to store one document at a time in the shared memory, and to use one block of threads to process the document. Initially, one block will contain 128 threads, and up to 512 threads can be used. The optimal number of threads in a block will be chosen experimentally. There will be 65,535 thread blocks in a 1-dimensional grid. That is, the GPU may process 65,355 documents with one launch, and each document will be stored in the shared memory to be processed by one block of threads. For the thread design and memory usage, see Figure 2.

4.5.2 Tokenization—The document is copied from the global memory to the shared memory as an array of characters. As shown in Figure 3, we use 128 threads in a block. These threads will examine the first 128 characters in the array, and mark the locations of token breaking characters (blank spaces, or non-alphabet symbols). The beginning and the length of a token are stored. Then the next 128 characters are examined in the same fashion until the end of the document. After finding the locations of the tokens, the information is stored in the shared memory as an array, the token array.

4.5.3 Stopword Removal—To remove stopwords, each token in the token array must be compared to the list of 641 stopwords. To reduce the computation time, the 641 stopwords are stored in a hash table, and a minimal perfect hash function [14] is used. That is, these 641 stopwords are stored in an array of 641 entries without collision. For every token in the token array, the minimal perfect hash function () is used to compute its hash value, and the token is compared to the corresponding entry in the hash table. Thus these stopwords can be removed from the tokens. The GPU device has 64KB of constant memory, which is cached and is much faster than the global memory. The stopword hash table will be constructed by the CPU and copied to the GPU constant memory, which cannot be modified by the GPU threads. The tokens will be hashed by the minimal perfect hash function. If the hashed value of the token is between 0 and 640, the token is matched to the corresponding stopword in the hash table. Then the token is the stopword, it is removed from the token array.

4.5.4 Stemming—The Porter stemming algorithm will also be implemented in CUDA. The original implementation of the algorithm had highly branched code. Most of the branches in the algorithm are checking the presence of a suffix or prefix for removal. However, the CUDA programming platform does not support branch prediction, and any branching will lead to divergent execution, which is serialized on the GPU device. Thus, the implementation of the Porter stemmer will use a finite state automata [11]. The state transition table is used for lookup for each state and each input character. During the stemming, a state of each token is determined by the location of the suffix or prefix. The number of stemming states is small enough to be stored in the GPU constant memory. Then a token is stemmed in place; that is, the location and the length are modified to reflect the

stemming. The code does not use additional storage because it overwrites its input with output in the shared memory directly (Figure 4). Using the constant memory, the state transition table, and overwriting, the Porter stemming algorithm can be performed on the GPU.

4.5.5 Generating the Index of One Document—As shown in Figure 5, the document inversion follows the BHBI algorithm. First, the hash value of a token will be computed. The choice of a hash function is important. The clock cycles needed for operations in the GPU are different from those for the CPU architecture. For example, the integer multiplication and division take 16 cycles in the GPU. It is four times longer than the floating point multiplication, which takes only four cycles. The operations of addition and subtraction take four cycles in both cases. Therefore, elimination of integer multiplication/division or conversion from integer to floating point calculation are recommended to reduce the computing time. The SDBM hash function, which uses addition, subtraction, and bit shift, is used in the CUDA code. SDBM hash function is an extended version of NDBM, a part of the Berkely DB Library [18]. The hash values of terms are stored in the shared memory of a block of the threads.

4.5.6 Merging the Indices of All Documents—After a thread block has finished indexing a document, the indices in the index block are sorted by the hash values of terms using a parallel bubble sort method. Generally, the index block from a document is fewer than the number of the original tokens. Thus, if the number of block indices is fewer than the number of threads, it will be sorted with the threads. After sorting, the index blocks are transferred into GPU global memory. Then, the next documents are indexed in the same fashion until the end of the document in the current work block. When all of the available memory is full, the index blocks in the memory need to be merged together. In a global merge stage, American flag sort [10] is used to finalize the index in the work blocks. American flag sort consists of three steps. In the first step, the number of indices in each bucket is counted. In the second step, the starting position in the bucket array is computed. Lastly, indices are cyclically permuted to their proper bucket. There is no collection stage because the buckets are already stored in order. If the work block is sorted, the system copies the sorted block to the CPU and makes an index file. Then the GPU will process the next work block in the same fashion until the end of the work blocks. The finalized index is generated by merging and sorting the work block files in the CPU.

5. Result and Discussion

5.1 Dataset

Abstracts of PubMed and e-commerce product reviews were used for this research. The machine learning repository at University of California at Irvine (UCI) provides PubMed abstracts of 8,200,000 articles published before 2008 in the bag of word format as well as texts [6]. J. McAuley, *et al.* at University of California at San Diego (UCSD) used Amazon product data with 142.8 million reviews during May 1996 - July 2014 [9]. They released smaller dataset of certain categories as JSON data type. For the dataset, abstracts of PubMed articles and product reviews of e-commerce on-line stores were directly collected from each

web site. A total of 143,201 abstracts of PubMed articles and a total of 229,115 reviews in digital camera category were downloaded. Most abstracts and product reviews had less than 3,000 words. Therefore, most data files are less than 4KB in size.

5.1.1 Prediction of Term Size—For the experiment, preprocessing was performed with 8 sets of different numbers of abstracts from the collected abstract data. The numbers of abstracts in the sets are 1,000, 2,000, 4,000, 8,000, 16,000, 32,000, 64,000, and 128,000. After preprocessing, the results show the number of terms used in each set. 10,343 terms are used in 1,000 abstracts, and 146,062 terms are used in 128,000 abstracts. When the number of abstracts is doubled, the number of terms is increased by an average 1.46 times.

In order to reduce the collision in hash values of the terms, the width of hash values should be wide enough. Thus, the expected number of unique terms from a certain number of documents is needed to choose the width of hash values. The expected number of unique terms can be estimated by Heaps' law, which describes the portion of a vocabulary which is represented by certain documents consisting of terms chosen from the vocabulary [2]. Let M be the expected number of unique terms, and let n be the number of documents. K and β are parameters to be determined by empirical data. Heaps' law states

$$M = Kn^\beta$$

For English text, the range of K is between 10 and 100, and β is between 0.4 and 0.6. For PubMed abstracts, the parameters $K = 19$ and $\beta = 0.55$ are the best-fit for the preliminary data. Calculating with Heaps' law, if PubMed contains 16 million abstracts, the expected number of unique terms is around 2.2 million. Figure 6 shows that the preliminary data follows Heap's law.

A width W (the number of bits) of the hash values must be chosen so that the expected number of collision among M unique terms is less than one. Under this constraint, W and M are related by the following inequality:

$$M < \frac{1 + \sqrt{1 + 8 \cdot (2^W - 1)}}{2} \approx \frac{1}{2} + 2^{\left(\frac{W}{2} + \frac{1}{2}\right)}$$

A 26-bit hash function is expected to produce no collision for up to 11,585 terms. The abstracts from 1,000 articles have 10,343 unique terms. Thus, a 26-bit hash function is wide enough for 1,000 abstracts. A 41-bit hash function can safely hold up to 2.09 million terms, which can cover the 2.07 million terms from the estimated number of terms in 16 million PubMed abstracts. Currently, the PubMed has over 20 million articles and around 26% of the articles do not have abstracts. Therefore, a 41-bit hash function is enough for all PubMed abstracts without producing a collision. Since the average length of product reviews is shorter than one of abstracts, product reviews have a smaller number of terms. 8,931 terms are used in 1,000 product reviews and 81,590 terms are used in 128,000 product reviews.

Thus, a hash width of abstracts is enough to hold the terms, used in the same number of product reviews, without any collision.

5.2 Discussion of GPU Document Inversion

The CPU used for the experiments was a 3.4 GHz AMD Phenom II X4 965 processor with 4 cores, 8 GB total memory and 512 KB cache. The GPU used for the implementation was NVIDIA Tesla C2050 device with 14 multiprocessors and 32 cores per multiprocessor [19]. It has a GPU clock speed of 1.15 GHz and 3 GB GDDR5 global memory. The CUDA Toolkit 5.0 based on CentOS 6.5 was used.

The performance of document inversion using GPU is affected by the number of thread and the data transfer time between the host (CPU) and the device (GPU). For the comparison of speedup, 4 sets of abstracts, 16,000, 32,000, 64,000, and 128,000 are selected from PubMed abstract collection and 4 sets of reviews, 50,000, 100,000, 150,000 and 200,000 are selected from product review collection. Each document is less than 4KB size and at least 800MB in global memory is required to hold up to 200,000 documents.

Table 2 shows the variation in total run time for CPU and GPU for each set of abstract collection from PubMed and product review. Using 128 threads per one block design, the GPU performed 1.97-2.97 times as fast as CPU on average. Speed was dependent on the number of documents and on the number of threads. The data transfer from host to device and from device to host took a significant amount of time. Thus, the small number of abstracts showed less speedup than large size of data.

6. Conclusions

With the advent of CUDA and GPU, several attempts have been made to parallelize the existing algorithms as well as to develop other new algorithms that work best with CUDA architecture. In this work, we implement the parallel document inversion on high throughput document data using massively parallel computational device. We conducted preliminary experiments and found that the parallel document inversion on the GPU is 1.97 times to 2.97 times faster than the same method on the CPU. The performance of implementation is limited by the amount of global memory and shared memory available on the GPU, as it requires storing all the documents in the global memory. It is likely that even better performance may be achieved by employing a more sophisticated arrangement of threads, blocks, and the grid. With newer generation GPU devices and the latest version of CUDA toolkit, it could lead to improvements in speed. These findings indicate that this approach has potential benefits for large-scale document collection, and could be easily applied to other similar problems.

Acknowledgments

We would like to thank Julia Chariker for insightful discussions and comments. Funding was provided by a grant from the National Institutes of Health (NIH), National Institute for General Medical Science (NIGMS) grant P20GM103436 (Nigel Cooper, PI). The contents of this manuscript are solely the responsibility of the authors and do not represent the official views of NIH and NIGMS.

References

1. Atallah, MJ., Fox, S. Algorithms and theory of computation handbook. CRC Press, Inc.; Boca Raton, FL, USA: 1998.
2. Baeza-Yates, R., Ribeiro-Neto, B. Modern information retrieval. Addison Wesley; 1999.
3. Czech ZJ, Havas G, Majewski BS. Perfect Hashing. Theor Comput Sci. 182(1-2):1–143.
4. Frumkin M. Indexing text documents on gpu - Can you index the web in real time? NVIDIA, GPU Technology Conference. 2014
5. Harris, M. Nvidia; 2016. Inside Pascal: NVIDIA's Newest Computing Platform. <https://devblogs.nvidia.com/paralleforall/inside-pascal>
6. Lichman, M. University of California, Irvine, School of Information and Computer Sciences; Irvine, CA: 2013. UCI machine learning repository. <http://archive.ics.uci.edu/ml>
7. Manning, CD., Raghavan, P., Schütze, H. Introduction to information retrieval. Cambridge University Press; 2008.
8. Marziale, L., Richard, GG., III, Roussev, V. Proceedings of the 7th Annual Digital Forensics Research Workshop (DFRWS). Vol. 1. Pittsburgh, PA, USA: 2007. Massive threading: using GPUs to increase the performance of digital forensics tools; p. 73p. 81
9. McAuley, J., Targett, C., Shi, Q., van den Hengel, A. Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM; Santiago, Chile: 2015. Image-based recommendations on styles and substitutes; p. 43-52.
10. McIlroy PM, Bostic K, Mcilroy MD. Engineering radix sort. COMPUTING SYSTEMS. 6:5–27.
11. Naghmouchi, J., Scarpazza, DP., Berekovic, M. Proceedings of the 24th ACM International Conference on Supercomputing. ACM; Tsukuba, Ibaraki, Japan: 2010. Small-ruleset regular expression matching on GPGPUs: quantitative performance analysis and optimization; p. 337-348.
12. Narang, A., Agarwal, V., Kedia, M., Garg, VK. IEEE International Conference on High Performance Computing (HiPC). IEEE; Kochi, India; 2009. Highly scalable algorithm for distributed real-time text indexing; p. 332-341.
13. NVIDIA. NVIDIA Corporation; 2007. NVIDIA CUDA compute unified device architecture programming guide. http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf
14. Pearson PK. Fast hashing of variable-length text strings. Commun ACM. 33(6):677–680.
15. Porter MF. An algorithm for suffix stripping. Program. 14(3):130–137.
16. Ryoo, S. Ph D Thesis. University of Illinois; Urbana, IL: 2008. Program optimization strategies for data-parallel many-core processors.
17. Scarpazza, DP., Braudaway, GW. IEEE International Symposium on Workload Characterization (IISWC). IEEE; Austin, TX, USA: 2009. Workload characterization and optimization of high-performance text indexing on the Cell Broadband Engine; p. 13-23.
18. Seltzer, M., Yigit, O. Proceedings of the USENIX Winter 1991 Conference. USENIX; Dallas, TX, USA: 1991. A new hashing package for UNIX; p. 173-184.
19. Shimpi, AL., Wilson, D. Purch; New York, NY: 2008. NVIDIA's 1.4 billion transistor GPU. <http://www.anandtech.com/show/2549/21>
20. Sophoclis, NN., Abdeen, M., El-Horbaty, ESM., Yagoub, M. 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE). IEEE; Montreal, QC, Canada: 2012. A novel approach for indexing Arabic documents through GPU computing; p. 1-4.
21. Yamada, H., Toyama, M. Proceedings of the Twenty-First Australasian Conference on Database Technologies. Australian Computer Society, Inc.; Brisbane, Australia; 2010. Scalable online index construction with multi-core CPUs; p. 29-36.

Biographies

Mr. Sungbo Jung received his Bachelor's degree from the school of Journalism and Mass Communication at the Korea University, Seoul, South Korea and Master's degree from

Computer Engineering and Computer Science at University of Louisville, Louisville, KY, USA. Currently, he is a Ph.D. candidate in Computer Engineering and Computer Science at University of Louisville. He is broadly interested in the research area of High Performance Computing(HPC), Bioinformatics, and Information Retrieval. Contact him at sungbo.jung@louisville.edu.

Dr. Dar-jen Chang is an Associate Professor of the Department of Computer Engineering and Computer Science (CECS) at the University of Louisville. He received an M.S. in Computer Engineering and a Ph.D. in Mathematics from the University of Michigan, Ann Arbor. His current research interests include computer graphics, 3D modelling, computer games, GPU computing, and compiler design with LLVM. He can be contacted at djchan01@louisville.edu.

Dr. Juw Won Park is an Assistant Professor at the Department of Computer Engineering and Computer Science, University of Louisville. His research interests include the analysis of alternative mRNA splicing and its regulation in eukaryotic cells using high-throughput sequencing along with related genomic technologies. He has a Ph.D. in computer science from the University of Iowa. He can be contacted at juw.park@louisville.edu.

Figure 1.
Document inversion process

Figure 2.
Thread design and memory usage of the proposed GPU document inversion system

Figure 3. Each thread checks the blank-space or non-alphabet symbol to find a start and a stop position of tokens in Shared memory

Figure 4.
GPU stemmer overwrites its input with the output in Shared memory

Figure 5.
Every token will be hashed as its Term Hash

Figure 6.
Estimated number of unique terms

Table 1**Various CUDA memory type**

Memory	Location	Cached	Access	Scope
Register	On-Chip	No	Read/Write	One thread
Local	Off-Chip	No	Read/Write	One thread
Shared	On-Chip	-	Read/Write	One block
Global	Off-Chip	No	Read/Write	All grid
Constant	Off-Chip	Yes	Read only	All grid
Texture	Off-Chip	Yes	Read only	All grid

Table 2
Document inversion of CPU vs GPU

Document		CPU (sec)	GPU (sec)	Speedup
PubMed Abstract	16000	23.59	11.98	1.97 ×
	32000	45.98	15.44	2.97 ×
	64000	89.21	31.18	2.86 ×
	128000	178.33	62.43	2.85 ×
Product Review	50000	63.41	22.48	2.82 ×
	100000	112.67	40.38	2.79 ×
	150000	183.85	64.96	2.83 ×
	200000	242.22	84.99	2.84 ×