A Fast Algorithm for Constructing Inverted Files on Heterogeneous Platforms

Zheng Wei and Joseph JaJa
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies, University of Maryland
College Park, U. S. A
{zwei, joseph}@umiacs.umd.edu

Abstract—**Given a collection of documents residing on a disk, we develop a new strategy for processing these documents and building the inverted files extremely fast. Our approach is tailored for a heterogeneous platform consisting of a multicore CPU and a highly multithreaded GPU. Our algorithm is based on a number of novel techniques including: (i) a high-throughput pipelined strategy that produces parallel parsed streams that are consumed at the same rate by parallel indexers; (ii) a hybrid trie and B-tree dictionary data structure in which the trie is represented by a table for fast look-up and each B-tree node contains string caches; (iii) allocation of parsed streams with frequent terms to CPU threads and the rest to GPU threads so as to match the throughput of parsed streams; and (iv) optimized CUDA indexer implementation that ensures coalesced memory accesses and effective use of shared memory. We have performed extensive tests of our algorithm on a single node (two Intel Xeon X5560 Quad-core) with two NVIDIA Tesla C1060 attached to it, and were able to achieve a throughput of more than 262 MB/s on the ClueWeb09 dataset. Similar results were obtained for widely different datasets. The throughput of our algorithm is superior to the best known algorithms reported in the literature even when compared to those run on large clusters.**

Keywords- indexer; inverted files; multicore; GPU; pipelined and parallel parsing and indexing

I. INTRODUCTION

A critical component of all information retrieval systems including web search engines is the set of inverted files generated typically from a very large collection of documents. A considerable amount of research has been conducted to deal with various aspects related to inverted files. In this paper, we are primarily concerned with methods to generate the inverted files as quickly as possible. All the recent fast indexers use the simple MapReduce framework on large clusters, which enables quick development of parallel algorithms dealing with internet scale datasets without having to deal with the complexities of parallel programming. Such framework leaves the details of scheduling, processor allocation, and communication to the underlying run time system, and hence relieves programmers from all the extra work related to these details. However such an abstraction comes at a significant price in terms of performance, especially when using the emerging multicore processors. In this paper, we take the different approach that does try to exploit the common features present on current processors, both general CPUs and GPUs, to obtain a very fast algorithm for generating the inverted files.

Current trend in CPUs or GPUs increasingly includes more core processors on a single chip. It is expected that the number of cores will double every 18 to 24 months, and such trend is likely to continue in the foreseeable future. While the architectures of current and emerging multiprocessors vary significantly, they all include several levels of memory hierarchy, SIMD or vector type operations, and multithreaded cores. These processors offer unprecedented opportunities for speeding up demanding computations on a single processor if the available resources can be effectively used.

In this paper we consider a heterogeneous processor consisting of a CPU and two GPUs. The CPU consists of two Quad-Core Intel Xeon X5560 with 24 GB of main memory and each quad core shares a 8MB L3 cache. Two NVIDIA Tesla C1060 GPUs are attached to our processor. The CPU offers a multithreaded environment with a shared memory programming model. In this model, communication is carried out through the shared memory, and hence a careful management of the shared memory coupled with load balancing among the cores is critical to achieve good performance.

On the other hand, The basic architecture of our GPU consists of a set of Streaming Multiprocessors (SMs), each of which containing eight Streaming Processors (SPs or cores) executing in a SIMD fashion, 16,384 registers, and a 16KB of shared memory. All the SMs have access to a very high bandwidth Device Memory; such a bandwidth is achieved only when simultaneous accesses are coalesced into contiguous 16-word lines. However the latency to access the device memory is quite high and is around 400-600 cycles. In our work, we have used the NVIDIA Tesla C1060 that has 30 SMs coupled to a 4GB device memory with a peak bandwidth of 102 GB/s.

The CUDA programming model [1] of the NVIDIA GPUs envisions phases of computations running on a host CPU and a massively data parallel GPU acting as a co-processor. The GPU executes data parallel functions called

kernels using thousands of threads. Each GPU phase is defined by a grid consisting of all the threads that execute some kernel function. Each grid consists of a number of thread blocks such that all the threads in a thread block are assigned to the same SM. Several thread blocks can be executed on the same SM, but this will limit the number of threads per thread block since they all have to compete for the resources (registers and shared memory) available on the SM. Programmers need to optimize the use of shared memory and registers among the thread blocks executing on the same SM.

Each SM schedules the execution of its threads into warps, each of which consists of 32 parallel threads. Half-warp (16 threads), either the first or second half of a warp, is introduced to match the 16 banks of shared memory. When all the warp's operands are available in the shared memory, the SM issues a single instruction for the 16 threads in a half-warp. The eight cores will be fully utilized as long as operands in the shared memory reside in different banks of the shared memory (or access the same location from a bank). If a warp stalls, the SM switches to another warp resident in the same SM. Optimizing performance of multithreaded computations on CUDA requires careful consideration of global memory accesses (as few as possible and should be coalesced into multiple of contiguous 16-word lines); shared memory accesses (threads in a warp should access different banks); and partitioning of thread blocks among SMs; in addition to carefully designing highly data parallel implementations for all the kernels involved in the computation.

The rest of the paper is organized as follows. In the next section, we provide a brief background about the typical strategy used to build inverted files and a summary of the work that is most related to our paper. Section III provides a detailed description of our algorithm, including our new dictionary data structure, the organization of the parallel parsers and parallel indexers, and a high performance CUDA implementation of the indexer. Section IV provides a summary of our test results on three very different benchmarks, and we conclude in Section V.

II. Background and Previous Related Work

We start by giving a brief overview of the process of building inverted files given a collection of documents residing on a disk. The overview will be followed by a summary of previous work on parallel and distributed implementations of this strategy.

The overall process essentially converts a collection of documents into inverted files consisting of a postings list for each of the terms appearing in the collection as follows. The strategy starts by parsing each document into a "bag of words" of the form <term, document ID> tuples, followed by constructing a postings list for each term such that each postings contains the ID of the document containing the term, term frequency, and possibly other information. Parsing consists of a sequence of simple steps: tokenization, stemming, and removal of stop words. *Tokenization* splits a document into individual tokens; *stemming* converts different forms of a root term into a single common one (e.g. parallelize, parallelization, parallelism are all based on parallel); and *removal of stop words* consists of eliminating common terms, such as "the", "to", "and", etc. The overall parsing process is well understood, and follows more or less the same linguistic rules, even though there exist different stemming strategies.

The next phase consists of constructing the inverted index. All <term, document ID> tuples belonging to the same term are combined together to form the postings list of that term. During the construction, a dictionary is usually built to maintain the location of the postings list of each term and to collect some related statistics. Postings on the same list are usually organized in a sorted order of document IDs for faster look up. Indexing is a relatively simple operation—group tuples for the same term together and then carry out sorting by document IDs—but it is always by far the most time consuming part given the typical size of the collection to be indexed.

In [2], postings lists are written as singly linked lists to disk and the dictionary containing the locations of the linked lists remains in main memory; however, another run is required as post-processing to traverse all these linked lists to get the final contiguous postings lists for all terms. Moffat and Bell proposed sort-based indexing in [3] for limited memory. Their strategy builds temporary postings lists in memory until the memory space is exhausted, sorts them by term and document ID and then writes the result to disk for each run. When all runs are completed, it merges all these intermediate results into the final postings lists file. The dictionary is kept in memory; however as the size grows, there may be insufficient space for temporary postings lists. Heinz and Zoble [4] further improved this strategy to a single-pass in-memory indexing version by writing the temporary dictionary to disk as well at the end of each run. Dictionary is processed in lexicographical term order so adjacent terms are likely to share the same prefix and front-coding compression is employed to reduce the size.

We now turn to a review of the major parallel strategies that appeared in the literature. In [5], the indexing process is divided into loading, processing and flushing; these three stages are pipelined by software in such a way that loading and flushing are hidden by the processing stage. The *Remote-Buffer and Remote-Lists* algorithm in [6] is tailored for distributed systems. In the first run, the global vocabulary is computed and distributed to each processor and in the following runs, once a <term, document ID> tuple is generated, it is sent to a pre-assigned processor where it is inserted into the destination sorted postings list.

Today, MapReduce based algorithms are prevalent. First proposed in [7], the MapReduce paradigm provides a simplified programming model for distributed computing involving internet scale datasets on large clusters. The Map workers emit <key, value> pairs to Reduce workers defined by Master node, and the runtime would automatically group incoming <key, value> pairs received by a Reduce worker according to key field and pass <key, list of values associated with this key> to the Reduce function. A straightforward MapReduce algorithm for indexing is to use term as key and document ID as value, in which case the

Reduce workers can directly receive unsorted postings lists. Since there is no mechanism for different Map workers to communicate with each other, creating a global dictionary is not possible. McCreadie et.al let Map worker emit <term, partial postings list> instead to reduce the number of emits and the resultant total transfer size between Map and Reduce since duplicate term fields are less frequently sent. Their strategy has achieved a good speedup relative to the number of processors and cores [8]. Around the same time, Lin et.al [9] developed a scalable MapReduce Indexing algorithm by switching <term, posting{document ID, term frequency}> to <tuple{term, document ID}, term frequency>. By doing so, there is at most one value for each unique key, and moreover it is guaranteed by the MapReduce framework that postings arrive at Reduce worker in order. As a result, a posting can be immediately appended to the postings list without any post processing. Their algorithm seems to achieve the best known throughput rate for full text indexing.

We note that almost all the above strategies perform compression on the postings lists for otherwise the output file would be quite large. Because document IDs are stored in sorted order in each postings list, a basic idea used is to encode the gap between two neighbor document IDs instead of their absolute values combined with a compression strategy such as variable byte encoding, γ encoding and Golomb compression.

III. Description of our Algorithm

Our main goal in this paper is to present a very fast indexing algorithm for today's common platform – a multicore CPU augmented by a GPU accelerator. More specifically, we use the Intel Processor Xeon X5560 consisting of two quad-core processors and two NVIDIA Tesla processors each consisting of 240 streaming cores and 4GB of device memory. Our algorithm can easily be adapted to any other such heterogeneous configuration.

A. Overall Approach

The cores on the CPU offer opportunities for a limited amount of parallelism on highly irregular computations. On the other hand, the streaming cores on the GPU are ideally suited for a very high number of fine grain data parallel computations. Our approach attempts to exploit both capabilities simultaneously, carefully orchestrating which tasks are assigned to the CPU cores and which are assigned to the GPU streaming cores. This approach is illustrated in Fig. 1.

Briefly, a number of parsers run in parallel on the CPU, where each parser reads a fixed size block of the disk containing the documents, executes the parsing algorithm, and then writes the parsed results onto a buffer. A number of indexers, some running on CPU cores and the rest running on GPU cores, pull parsed results from the buffer as soon as they are available and jointly construct the postings lists, which are written into a disk as soon as they are generated. The dictionary remains in the CPU memory until the whole process is completed.

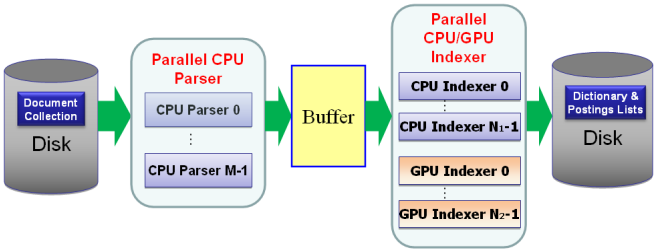

Figure 1. Dataflow of Pipelined and Parallel Indexing

There are many details that need to be carefully worked out for this approach to achieve optimal throughput. Before providing details about the parsing and indexing tasks and how they are allocated to the available cores, we describe the dictionary data structure used since it plays a central role through which the various tasks coordinate their work. This will be followed by a description of the parsing and indexing tasks allocated to the various cores available on our heterogeneous platform, and how coordination between the CPU and GPU cores is carried out.

B. Dictionary Data Structure

The structure of the dictionary plays a critical role in the performance of our indexing algorithm since multiple concurrent threads have to access the dictionary and hence conflicts among the corresponding parallel threads must be properly resolved in such a way to ensure correctness and achieve high performance. The B-Tree is the typical data structure used in many information retrieval systems due to its balanced structure and small height. In particular, the height of any n-key B-Tree is at most $\log_t \frac{n+1}{2}$ where t is the degree of the tree. Such a structure is not in general suited for operations such as multiple threads attempting to insert a new term into the same node or any other operations with similar conflicts. Locks can be used to prevent such hazards but the overhead is extremely high since many threads may have to wait until a thread completes its modification of the B-Tree. In our implementation, we introduce a combination of a trie at the top level and a B-Tree attached to each of the leaves of the trie. A similar data structure was used in [10] to achieve compact size and fast search; however in our case we will exploit this hybrid data structure to achieve a high degree of parallelism and load balancing among the heterogeneous processor cores.

1) A Hybrid Data Sructure

Our hybrid data structure for the dictionary is shown in Fig. 2. Essentially, terms are mapped into different groups, called trie collections, followed by building a B-tree for each trie collection.

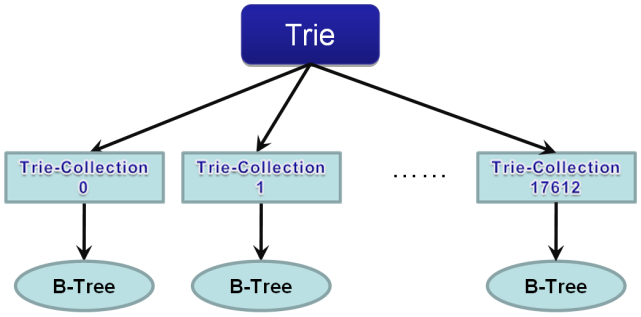

Figure 2. Hybrid of Trie and B-Tree Structure of Dictionary

The main reason we use a trie at the top level is to generate many independent B-trees instead of a single B-Tree. Each B-tree is then handled by a single thread, independently of the other B-trees. For our platform, we fix the height of the trie to three, which means that the first three letters in a term are used to determine the corresponding the index of the trie collection. In fact, the index category of each leaf of the trie is specified in Table I. Clearly, the number of terms belonging to different trie collections varies significant; for example, there are many words with prefix "the" and hardly any terms with prefix "zzz". The height of three for the trie seems to work best since a smaller height will lead to a wide variety of trie collections, some very large and some very small, which will be hard to allocate to the different core processors in such a way as to achieve a good load balance. A larger value for the trie height will generate many small trie collections, which will be again hard to manage. Since the trie height is constant here, we don't need to actually build the trie structure but we use a table to map a trie index directly into the root location of the corresponding B-Tree.

TABLE I. TRIE-COLLECTION INDEX DEFINITION

Index		Term Category	Example
Special	0	Terms can't fall into other categories	"-80", "3d", "Česky"
Pure Numbers (10 entries)	1	Numbers starts with '0'	"01","0195"
	…	…	…
	10	Numbers starts with '9'	"9", "954"
Terms with ≤3 letters or special letter in the first 3 letters (26 entries)	11	Terms start with 'a' and (1) with ≤3 letters and or (2) with special letter in the first 3 letters	"a", "at", "act", "añonuevo"
	…	…	…
	36	Terms start with 'z' and (1) with ≤3 letters and or (2) with special letter in the first 3 letters	"z", "zoo", "zoé"
Terms with >3 letters and no special letter in the first 3 letters (26*26*26=17576 entries)	37	Terms with >3 letters and starts with 'aaa'	"aaat", "aaaé"
	38	Terms with >3 letters and starts with 'aab'	"aabomycin"
	…	…	…
	17612	Terms with >3 letters and starts with 'zzz'	"zzzy"

In addition to allowing a high degree of parallelism through the independent B-trees, our hybrid data structure achieves two more benefits. Since we replace a big B-tree by many small B-trees, the heights of the B-trees are smaller, implying that the time to search or insert a new term is reduced as well. Another advantage of the trie lies in the fact that terms belonging to the same trie index share the same prefix (except trie index 0) and hence we can eliminate such common prefix, save memory space for term strings and reduce string comparison time in B-tree operations. The average length of a stemmed token is 6.6 in the ClueWeb09 dataset and hence removing the first three letters results in almost doubling the string comparison speed. An alternative option to the trie is to use a hash function, but a hash function will still require comparisons and searches on full strings and hence won't be as effective as the trie.

2) Special Node Structure in B-tree

The structure of a B-tree node is illustrated in Table II. The degree of B-tree is 16, that is, each node can hold up to 31 terms, and this number is selected to match the CUDA warp size. Since the length of a term string is not fixed but varies over a wide range, it is impossible to store the strings within a fixed B-Tree node; instead, pointers are used to indicate the memory location of the actual strings. During a search or insert operation into one of the B-trees, strings are accessed through these pointers, and such operations can be quite expensive on both the CPU and the GPU. To get around this problem, we include 31 four-byte caches in each node. These caches are used to store the first four bytes of the corresponding term strings. Consider for example the term "application"—the first 3-byte "app" is not needed since it is already captured by the trie, so we only have to store the term string "lication" into the B-Tree, and hence "lica" is stored in the cache, and the remaining string is stored in another memory location indicated by the term string pointer.

Occasionally some memory space will be wasted when caches are not fully occupied. However the advantages of our scheme are substantial because:

- *Short strings can be fully stored within the B-tree node;*
- *For long strings, even though only the first four bytes are stored, it is highly likely that the required comparison between two term strings can be done with only these four bytes since it is a rare case that two arbitrary terms share the same long prefix.*

TABLE II. DATA STRUCTURE OF ONE B-TREE NODE

Field	Number	Data Size (Byte)
Valid term number	1	4
Pointer to term string	31	124
Leaf indicator	1	4
Pointer to postings lists	31	124
Pointer to children	32	128
4-Byte Cache for term string	31	124
Padding	1	4
Total Size		512

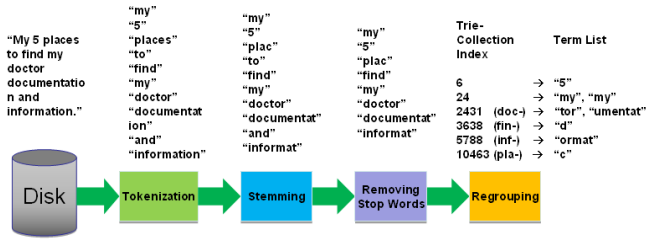

Figure 3. Data Flow of One Parser Thread

C. *Parsers*

As mentioned earlier, we will have several parsers running in parallel, the number of which depends on the number of CPU cores available and will be discussed further later. Here we describe the sequence of operations executed by each parser, illustrated in Fig. 3. Each such sequence will be executed by a single CPU thread. The corresponding steps are briefly described next.

- ***Step1*** *reads files from disk, decompresses them if necessary, assigns local document ID to each document, and builds a table containing <document ID, document location on disk> mapping.*
- ***Step2*** *performs tokenization, that is, parses each document into tokens and determines the trie index of each resulting term.*
- ***Step3*** *performs Porter stemmer.*
- ***Step4*** *removes stop words using a stop word list.*
- ***Step5*** *rearranges terms with the same trie index so that they are located contiguously. In addition, the prefix of each term captured by the trie index is removed.*

The first four steps are standard in most indexing systems. Step5 is special to our algorithm. Essentially, this step regroups the terms into a number of groups, a group for each trie collection index as defined by our dictionary data structure. We note that the overhead of this regrouping step is relatively small, about 5% of the total running time of the whole parser in all our experiments. This is due to the fact that tokenization scans input document character by character and hence a trie index can be calculated as a by-product using a minimal additional effort.

This regrouping is clearly needed for our parallel indexing algorithm. However, even in the case when indexing is carried out by a serial CPU thread, regrouping results in approximately 15-fold speedup based on our tests. The improved performance is due to improved cache performance caused by the additional temporal locality. Now we are processing a group of terms falling under the same trie collection index, which are inserted into the same small B-tree whose content stays in cache for a long time.

After processing a number of documents with a fixed total size, e.g. 1GB, the parsed results organized according to trie index values will be passed to the indexers. For trie collection index *i*, the parsed results will look like:

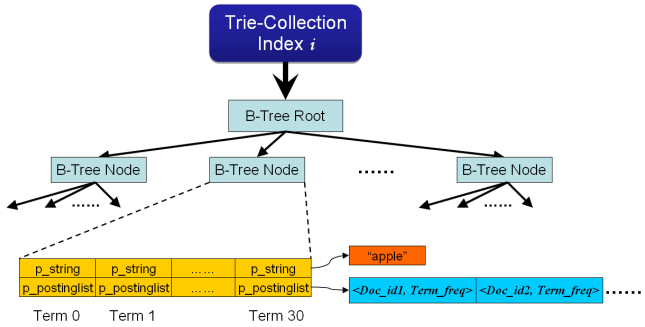

Figure 4. A B-tree Corresponding to a Single Trie Collection Index

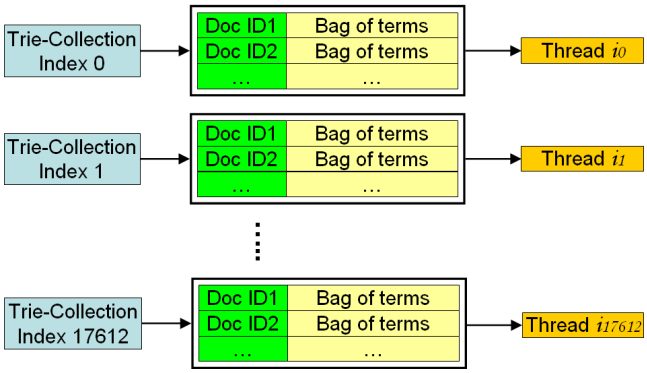

Figure 5. Work Assignments among Multiple Threads

Trie Collection corresponding to index i: (Doc_ID1, term1, term2, ...), (Doc_ID2, term1, term2, ...), ...

*Doc_ID*s in the lists are local ones within this parser. A global document ID offset will be calculated by the indexer; thus the global document ID can be obtained by adding *Doc_ID* and the global offset.

D. *Indexers*

As described in Section *A*, we will have a number of indexers running in parallel, some on the CPU cores and the rest on the GPU cores. In this section, we focus on describing the algorithm for an indexer either for a single CPU thread or for a single GPU kernel.

The purpose of an indexer is to construct all the B-Trees and the postings lists corresponding to each input term as shown in Fig. 4.

Clearly, the B-tree of each trie collection can be built independently of the rest of the B-trees. To ensure load balancing, a CPU thread or a GPU kernel will take care of the B-trees of several trie collections as we will describe later. However, we focus here on the algorithm used to build a single B-tree.

1) *CPU Indexer*

A CPU indexer is executed by a single CPU thread, which follows the commonly used procedures for building the B-tree and the corresponding postings lists. The only difference is to make use of the fact that a cache is included within each B-tree node. Hence, when a new term is inserted into a B-tree, the first 4-bytes of the string are stored in the

string cache field in the appropriate B-tree node. The remaining bytes, if any, are stored in another memory location, which can be reached via the string pointer for this term. Multiple CPU threads can execute concurrently without any modification to build B-trees corresponding to different trie collection indices.

2) GPU Indexer

We allocate the work to build a single B-tree and the related postings lists corresponding to a single trie collection to a thread block consisting of 32 threads (forming a warp). At this stage, we assume that our term strings are already moved into the device memory and are represented as indicated in Fig. 6. Without loss of generality, we also assume that no term is longer than 255 bytes and hence one byte will be sufficient to hold the length of the corresponding string. We read these term strings in contiguous chunks (512B) and store them into the shared memory corresponding to the thread block handling this particular trie collection. Hence we are making use of coalesced memory accesses to move the data into the streaming multiprocessor shared memory. The GPU threads will then access the shared memory to process the corresponding terms instead of accessing the device memory.

Each term is now inserted into the B-tree using the 32 threads as follows. Starting from the root and as we go down the B-tree, we move the next B-tree node to be examined into the shared memory using coalesced memory access. We use the available threads to perform a comparison between the term to be inserted and each of the terms stored in the node in parallel. This parallel comparison operation followed by a parallel reduction step [11] will enable us to identify the location of the new term as indicated in Fig. 7. If this term needs to be inserted in the current position of term ($i+1$), then term ($i+1$) up to the current last term in the node must be shifted to the right, which is achieved by a number of parallel threads.

During B-tree insertion, three major operations inside a B-Tree node can take place and they are all carried out in parallel using coalesced device memory accesses.

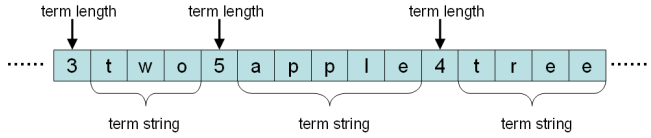

Figure 6. String Representation: Term Length in the First Byte

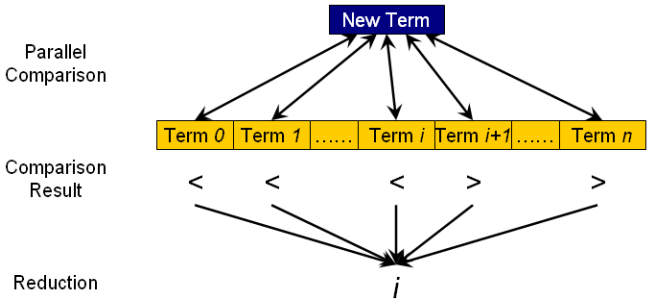

Figure 7. Parallel Comparison in One GPU Thread Block

- ***Searching***: the algorithm compares the new term with existing terms inside the current B-Tree node and then do one of the following: (1) if this term is already present, we update the postings lists; (2) if this term is not there and this node is not a leaf, we proceed to the corresponding child node for searching; (3) if this term is not there and this node is a leaf, we insert this term into this node.
- ***Inserting***: in order to insert a new term, we must first shift those existing terms which are "larger" than the new term so that a blank location is created to accommodate this new term.
- ***Splitting***: before accessing a B-Tree node, we check to determine whether this node is full or not and if yes, the node will be split into two nodes.

We now address the issue of how the trie collections assigned to the GPU will be handled. Since the trie collections are of different sizes and depend on the input documents, any static allocation of these collections to the available thread blocks is likely to incur a serious load imbalance. In our algorithm we use a dynamic round-robin scheduling strategy such as whenever a thread block completes the processing of a particular trie collection, it starts processing the next available trie collection.

E. Load Balancing between the CPU and GPU Indexers

In deciding how to allocate the trie collections among the CPU and GPU, we exploit the strength of each architecture—a large cache on the CPU and a high degree of data parallelism on the GPU. We divide the trie collections into two major groups. The first group, to be called *popular* trie collections, consists of the trie collections containing the most frequently occurring terms. In this group, a few common terms dominate the entries in each corresponding trie collection (by Zipf's law [12]). In this case, the B-tree nodes on the path from the root to these common terms will be accessed frequently and hence it makes sense to store such paths in cache, which would indeed happen if we process such collections on the CPU.

The second group consists of the remaining collections. Unlike the popular trie collections, this group contains primarily infrequent terms and, again according to Zipf's law, the differences in their frequencies are relatively very small. This means that every time we perform a B-tree operation for a new term, the path taken is likely to be very different than the previous one, and hence caching won't be so useful. However exploiting data parallelism in processing each node (to perform all the comparisons in parallel) speeds up the computation significantly, and this is exactly what we do using CUDA thread blocks.

Therefore, we assign the popular trie collections to a number of CPU indexers and unpopular ones to the GPU indexers. To determine which collections belong to which group, we extract a sample from the document collection, e.g. 1MB out of every 1GB, and run several tests on the sample to determine membership. Since there are many trie collections in this group and we have multiple GPUs, say N

(N=2 on our platform), we use a simple method of splitting the unpopular trie collections among the N_2 GPUs by assigning the trie collection TC_i with index i to the GPU whose index is given by i mod N_2. For example, if unpopular trie collections have indices (0, 13, 27, 175, 384, 5810, 10041, 17316) and there are two GPU, then (0, 384, 5810, 17316) are assigned to GPU indexer 0 and (13, 27, 175, 10041) to GPU indexer 1.

However there are relatively very few popular trie collections (around one hundred), and hence we split these trie collections into N_1 independent sets such that each contains almost the same number of tokens, where N_1 is the number of CPU threads used.

Once a trie collection is assigned to a particular indexer, it is bound with this indexer through the program lifetime. That is to say, every indexer keeps an independent and exclusive part of the global dictionary and will focus on this part only.

In addition to the main indexing step, pre-processing delivers input to the GPU device memory and post-processing combines postings lists from all indexers, compress them with variable bytes encoding and then write the compact results to disk. These two steps are serialized. We call such procedure beginning with input data in a parser buffer and ending in postings lists as a single run, as shown in Fig. 8.

F. Overall Pipelined Data Flow

In our setting, the input document data collection is stored on a disk which is processed through our heterogeneous platform to generate the postings lists residing on a disk. The dictionary is kept in main memory until the last batch of documents is processed, after which it is moved to the disk. To avoid several parsers from trying to read from the same disk at the same time, a scheduler is used to organize the reads of the different parsers, one at a time. On the other hand, an output buffer is allocated to each parser to store the parsed results. The CPU and GPU indexers in the next stage will read from these buffers in order, that is, (buffer of Parser 0, buffer of Parser 1, …, buffer of Parser M-1, buffer of Parser 0, …). Such read sequence is enforced to ensure that document first read from disk will also be indexed first so the postings lists are intrinsically in sorted order. The number of parsers, the number of CPU indexers and the number of GPU indexers devices are determined by the physical resources available. In the next section, we determine the best values of these parameters for our platform.

We note that a separate output file is created for the postings lists generated during a single run, whose header contains a mapping table indicating the location and length of each postings list. This mapping table is indexed by the pointers to postings lists stored in the dictionary as shown in Table I. To retrieve a postings list for a certain term string, we look it up in the dictionary and use the corresponding pointer to determine the location of the partial postings list in each of the output files. This output format has some additional benefits including:

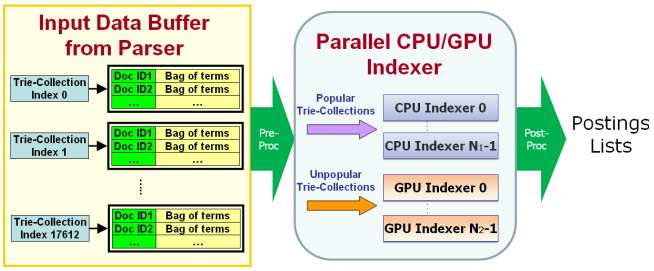

Figure 8. Data Flow of One Single Run on Parallel CPU and GPU Indexers

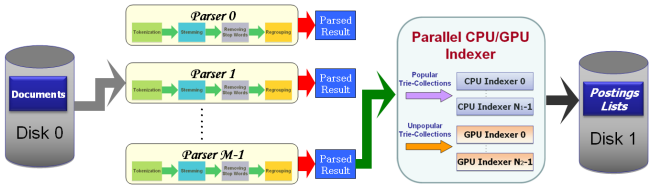

Figure 9. Pipelined Data Flow of Overall Indexing System

- *faster search when narrowed down to a range of document IDs since we can fetch only those partial postings lists that overlap with this range. This is possible since we include an auxiliary file containing the mapping of document IDs to output file names; and*
- *the possibility of parallel reading of the postings lists because the output files can be written onto multiple disks.*

Although a postings list is divided into partial lists and stored in separate files, the index is still monolithic for the entire document collection. If necessary, we can combine the partial postings lists of each term into a single list in a post-processing step, with an additional cost of less than 10% of the total running time.

IV. Experimental Evaluation of our Algorithm

Our parallel and pipelined indexing system is tested on a single machine that holds two Intel Xeon X5560 Quad-core CPUs and two NVIDIA Tesla C1060 GPUs each with a 4GB device memory. We use three document collections to test the performance of our algorithm. We start with the first English segment of the ClueWeb09 collection, which has been heavily utilized by the information retrieval community. Crawled between January and February 2009 by Language Technologies Institute at Carnegie Mellon University, this data set includes 50,220,423 web pages packed into 1,492 files with a total size of 230GB compressed and 1.389 TB uncompressed. The second data set is the Wikipedia01-07 data, which is derived from a publicly available XML dump of Wikipedia articles created on January 3th 2008 with 83 monthly snapshots between February 2001 and December 2007. The third is the Congressional data set from the Library of Congress, which includes weekly snapshots of selected news and government websites crawled between May 2004 and September 2005 by the Internet Archive.

Overall statistics about the three are given in Table III. These document collections are stored on a disk connected to our platform via a 1Gbs Ethernet. The generated output, postings lists and dictionary, are written to either a remote disk (with 1Gbs connection) or to a local disk (resulting performance differences are very small and insignificant). We report results averaged over three trials and in all our tests the differences between the fastest and slowest execution times have been less than 2%.

In what follows, we start by determining the best values on our platform for the following parameters: number of parallel parsers, number of CPU indexers, and number of thread blocks and number of threads per block for each of the two GPUs. This will be followed by summarizing the performance of our algorithm on the three document collections. We end by comparing our performance to the best reported results in the literature.

A. *Optimal Numbers of Parallel Parsers and Indexers*

In this section we focus on determining the best number of parallel parsers and indexers. Note that our goal is not only to speed up the parsing of the documents but also to match it with the speed at which indexers are able to consume the parsed data. Fig. 10 illustrates the performance of our algorithm on the ClueWeb09 data set as a function of the number of parsers under three scenarios: (1) M parsers and 8-M CPU indexers without GPU indexers; (2) M parsers and 8-M CPU indexers with 2 GPU indexers; and (3) M parsers without any CPU or GPU indexers. The value of M varies from 1 to 7 since there are only eight physical cores available.

TABLE III. STATISTICS OF DOCUMENT COLLECTIONS

	ClueWeb09 1st Eng Seg	Wikipedia 01-07	Library of Congress
Compressed Size	230GB	29GB	96GB
Uncompressed Size	1422GB	79GB	507GB
Crawl Time	01/09 to 02/09	02/01 to 12/07	05/04 to 09/05
Document Number	50,220,423	16,618,497	29,177,074
Number of Terms	84,799,475	9,404,723	7,457,742
Number of Tokens	32,644,508,255	9,375,229,726	16,865,180,093

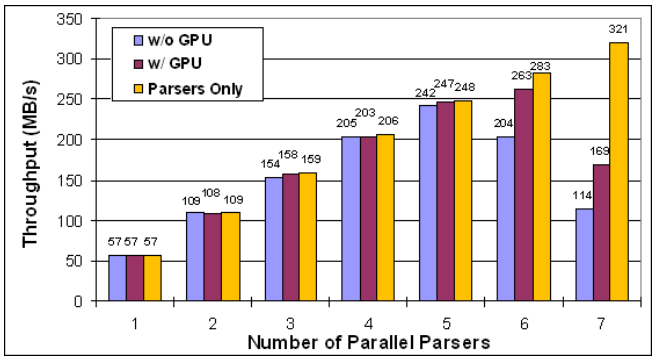

Figure 10. Optimal Number of Parallel Parsers and Indexers

When the number of parsers is within the range of 1 to 5, we observe similar performance among the three scenarios, including an almost linear scalability as a function of the number of parsers. This indicates that the indexers are keeping up with the data generated by the parsers and hence within this range the parsers constitute the slow stage of the pipeline. The major limitations to speeding up the parsers include the sequential access to our single disk and the contention on cache and memory bandwidth resources. Beyond 5 parsers, when the number of CPU indexers decreases, the indexing pipeline stage is not able to catch up with the parsing stage without the help of the GPU. On the other hand, the streams generated by 6 parsers can be consumed by the 2 CPU and 2 GPU indexers.

In other words, with acceleration from the GPU, when six parsers are running in parallel, the total parsing throughput achieved through the first pipeline stage approximately matches the indexing throughput of the second pipeline stage. When GPU is disabled, a ratio of 5:3 between parsers and indexers is the best possible.

We now clarify an issue related to the data format processed by the parsers. A typical file of the ClueWeb09 data set is about 160MB compressed and 1GB uncompressed. On average, it takes around 1.6 seconds to read such a compressed file and 3.2 seconds to decompress it. On the other hand, it takes about 10 seconds to read the uncompressed file via the 1Gbs Ethernet. Therefore we load the compressed files from disk and then decompress them in memory before parsing. Two choices are possible here: decompression can be folded into either the file read stage or as a separate step after reading. The advantage of the former is that decompression can be partially hidden by file access time if decompression starts whenever there is any data available in memory, so the overall time for reading and decompressing a file takes 3.8 seconds on average, which translates into 263MB/s intake bandwidth. The disadvantage of this method is that the file access right cannot be released to a parser until reading and decompression are both completed. This causes a mismatch between the data generated by the parsers and the data consumed by the indexers. Hence we choose the second scheme in which decompression starts after the file is fully transferred to memory. In this case, the average time to read a compressed file is $(1.6+3.2/p)$ seconds where p is the number of parallel parsers. When p=6 the intake bandwidth reaches as high as 469MB/s.

B. *Scalability of the Number of Parallel Indexers*

Given that we have already determined that the best overall performance is achieved by using six parsers, we now proceed to examine the scalability of CPU and GPU indexers in combination with the six parser threads. We test the performance of our algorithm using five configurations: (i) no CPU indexer and two GPUs; (ii) one CPU indexer and no GPUs; (iii) two CPU indexers and no GPUs; and (iv) two CPU indexers and two GPUs. However we start by discussing the best possible strategy for indexing on a single GPU (Tesla C1060). As mentioned earlier we use thread blocks each with 32 threads to match the number of keys in

each B-tree node and load each such node into shared memory using the 32 threads to achieve coalesced access. After extensive testing using a wide range of values for the number of thread blocks, it turns out that the best performance is achieved by using 480 thread blocks per GPU. From now on, whenever we refer to a GPU indexer we mean 480 thread blocks are running on a single GPU, each with 32 threads.

We now focus on the indexing time using the four configurations mentioned above. First, we notice a speedup by a factor of 1.77 when we use two CPU indexer threads compared to single CPU indexer thread and an extra 37.7% performance gain is achieved through the acceleration from the two GPU indexers. Second, if we calculate the indexing throughput by dividing uncompressed input data size by indexing time, we notice that the throughput from CPU+GPU combination in scenario (iv) is higher than throughput of CPU in scenario (iii) plus throughput of GPU in scenario (i) separately. This superlinear scalability comes from our specific task partitioning strategy between CPU and GPU so both processors are doing computation that they are good at. Note that the performance of multiple GPU indexers is limited by the time it takes to transfer the parsed input to the GPU device memory and the time it takes to move the output postings lists back to main memory at the end of each single run. Another issue is the possibility of load imbalance among the CUDA threads, which is hard to fully control.

If we calculate the difference between the total indexer time and the sum of the pre-processing time, the indexing time and the post-preprocessing time, we obtain the time during which the indexers are waiting for results from the parsers. This is due to the fluctuations between the two pipeline stages, which are very hard to fully control since they are input dependent. Note that this gap can be occasionally severe during some runs.

Detailed running times of various steps are shown in Table IV.

TABLE IV. RUNNING TIME OF INDEXERS

	Time (second)			
	6 Parsers 2 GPU Indexers	*6 Parsers 1 CPU Indexers*	*6 Parsers 2 CPU Indexers*	*6 Parsers 2 CPU Indexers &2 GPU Indexers*
Pre-Processing Time (second)	107.01	93.44	111.74	104.15
Indexing Time (second)	19313.6	11243.61	6357.67	4616.78
Post-Processing Time (second)	417.21	416.66	521.52	464.04
Sum of above Three (second)	19837.82	11753.71	6990.93	5184.97
Total Indexer Time (second)	19858.69	11758.81	7019.87	5408.25
Indexing Throughput (MB/s)	75.41	129.53	229.08	315.46
Total Indexer Throughput (MB/s)	73.34	123.86	207.47	269.29

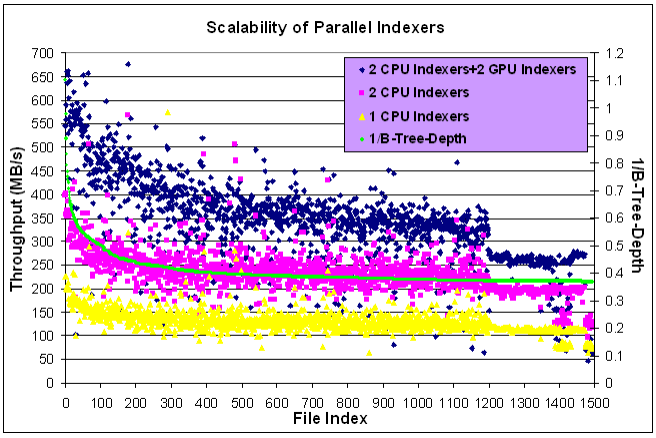

Figure 11. Scalability of Parallel Indexers

We now take a closer look at the indexing throughput (not including pre-processing and post-processing since they are serialized) of parallel indexers. We track the time of the parallel indexers spent on each file in the document corpus and compute the throughput for each file in scenario (ii), (iii) and (iv) as shown in Fig. 11. Note that starting with file index 1,200, we can see a significant drop in performance. This can be explained by the fact that the files with indices from 1,200 to 1,492 all belong to Wikipedia.org, and hence they exhibit a totally different behavior than previous documents. The combined CPU and GPU solution is especially affected with a significant drop in throughput after this point because our CPU and GPU parameters depend on sampling the whole collection prior to indexing and since the portion of the Wikipedia files is relatively small, the resulting parameters do not effectively reflect the characteristics of this small subset.

Note that overall slope is similar in all three scenarios: sharp decrease near the beginning followed by a trend that approaches a horizontal line. This pattern coincides with the inverse of the depth of B-tree because as the B-trees grow deeper, it takes more time to perform insert or search operations.

Now let's take a look at the contents processed by the CPU and the GPUs during indexing when the configuration consists of two CPU indexers and two GPU indexers. The GPU indexers process almost 80% the number of tokens compared to those processed by the CPU, while the numbers of terms and characters are respectively 2.5 times and 2.16 times as those performed by the CPU. This shows that the effectiveness of the way we split the work load between CPU and GPU.

TABLE V. WORK LOAD BETWEEN CPU AND GPU

	CPU Indexers	GPU Indexers
Token Number	14,465,084,050	18,179,424,205
Term Number	24,244,017	60,555,458
Character Number	239,433,858	513,640,554

TABLE VI. PERFORMANCE COMPARISON ON DIFFERENT DOCUMENT COLLECTION

Time Type	Time (second)			
	ClueWeb09	*ClueWeb09 w/o GPUs*	*Wikipedia 01-07*	*Library of Congress*
Sampling Time	59.53	57.53	7.27	29.01
Parallel Parsers	5410.89	7024.86	999.45	2437.79
Parallel Indexers	5408.25	7019.87	1023.96	2458.64
Dictionary Combine	2.46	2.54	0.26	0.21
Dictionary Write	59.21	54.92	0.57	0.80
Total Time	5541.62	7126.77	1033.34	2495.29
Throughput (MB/s)	262.76	204.32	78.29	208.06

C. *Performance of our Algorithm on Different Document Collections*

We show in Table VI the overall running times of our algorithm on the three different document collections. For all tests, six CPU parsers, two CPU indexers and two GPU indexers are used to achieve the best performance. The throughputs achieved on the ClueWeb09 and Library of Congress datasets are comparable. For the Wikipedia01-07 collection, the HTML tags were removed, and the remainder is just pure text. As we can see from Table III that although the uncompressed sized is only 1/18th of ClueWeb09, the numbers of documents and tokens are about one third to one fourth compared to the ClueWeb09 dataset. Hence the slower than 100MB/s throughput achieved on Wikipedia01-07 actually amounts to a very high processing speed.

D. *Comparison with Fastest Known Indexers*

In this section, we compare our algorithm to the best known algorithms that appeared in the literature, namely Ivory MapReduce [9] and Single-Pass MapReduce [8]. Both of these algorithms are implemented using the MapReduce framework, and hence the comparison is a bit unfair since these are high level algorithms that do not exploit the underlying architectures. The Ivory MapReduce tests are conducted on exactly the same ClueWeb09 collection using a cluster of 99 nodes each with two cores. Positional postings lists are generated by the Ivory MapReduce algorithm, which will add some extra cost but we don't believe this will alter the overall throughput numbers significantly. The Single-Pass MapReduce result uses a cluster of 8 nodes with a total of 24 cores on the .GOV2 collection. ClueWeb09 and .GOV2 are well-known collections in the information retrieval community. Detailed platforms comparison can be found in Table VII. The throughput numbers presented in Fig. 12 correspond to the uncompressed collection size divided by total running time.

It is clear that our pipelined and parallel indexing algorithm achieves the best raw performance with or without GPUs even when compared to much larger clusters. A number of factors contribute to the superior performance of our algorithm including:

TABLE VII. PLATFORM CONFIGURATION COMPARISON

	This Paper	Ivory MapReduce	SP MapReduce
Processors per Node	Two Intel Xeon 2.8GHz Quad-core CPUs Two NVIDIA Tesla C1060 GPUs	Two Intel Single-core 2.8GHz CPUs	One Intel Xeon 2.4GHz Quad-core CPUs (one core reserved for distributed file system)
Memory per Node	24GB	4GB	4GB
Node Number	1	99	8
Total CPU Cores Used	8	198	24
File System	Remote File System via 1Gb Ethernet	Hadoop Distributed File System	Hadoop Distributed File System

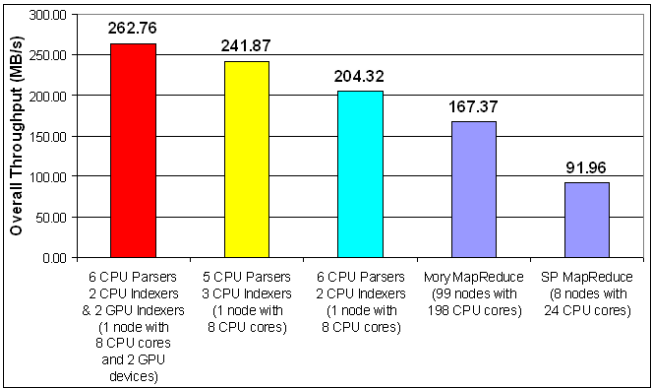

Figure 12. Comparison to Recent MapReduce Implementations

- *The pipelined and parallel strategy that matches maximum possible parsing throughput with parallel indexing on available resources.*
- *The hybrid trie and B-tree dictionary data structure, in which the logical trie is implemented as a table for fast look-up and each B-Tree includes character caches to expedite term string comparisons;*
- *Parallel parsers that scale well with the number of threads, in addition to the fact that the file reading and decompression are carefully optimized to boost the I/O bandwidth;*
- *The regrouping operation that is integrated into the parsing stage with little overhead but that noticeably increases CPU cache performance;*
- *The allocation of unpopular trie collections to the GPU where cache sensitive computations remain on the CPU; and*
- *The careful organization of memory accesses on the GPU in such a way as to exploit coalesced memory accesses and shared memory.*

V. CONCLUSION

In this paper, we presented a high performance pipelined and parallel indexing system based on a heterogeneous CPU and GPU architecture. We have shown how to optimize the performance of the pipeline by using parallel parsers and indexers in such a way that the streams produced by the parsers are consumed by the indexers at the same rate, and that rate is optimized. Several new techniques were introduced including a hybrid trie and B-trees data structure and optimized CUDA algorithm for indexing. The experimental tests reveal that our implementation on a single multicore processor shows scalable parallel performance in terms of the number of cores with a resulting throughput higher than most recent published algorithms on large clusters.

ACKNOWLEDGMENT

We would like to thank Jimmy Lin for providing us access to the ClueWeb09 dataset and discussing with us the details on his MapReduce implementation. We would also like to thank the Library of Congress and the Internet Archive for making the congressional dataset available to us. We would like to also give credit to Dr. Sangchul Song who developed the version of Wikipedia04-09 dataset which was used in our experimental evaluation. This research was supported by the NVIDIA Research Excellence Center at the University of Maryland and by an NSF Research Infrastructure award, grant number CNS 0403313.

REFERENCES

[1] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide Version 3.1.1", 2010.

[2] D. Harman and G. Candela, "Retrieving records from a gigabyte of text on a minicomputer using statistical ranking", Journal of the American Society for Information Science, vol. 41(8), pp. 581-589, Dec. 1990

[3] A. Moffat and T. A. H. Bell, "In situ generation of compressed inverted files", Journal of the American Society for Information Science, vol. 46(7), pp. 537-550, Aug. 1995.

[4] S. Heinz and J. Zobel, "Efficient single-pass index construction for text databases", Journal of the American Society for Information Science and Technology, vol. 54(8), pp. 713-729, June 2003.

[5] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina, "Building a distributed full-text index for the Web", ACM Transactions on Information Systems, Vol. 19(3), pp. 217-241, July 2001.

[6] B. Ribeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani, "Efficient distributed algorithms to build inverted files", SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, pp. 105-112, 1999.

[7] J. Dean and S. Ghemawat. "Mapreduce: Simplified data processing on large clusters", In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Dec. 2004.

[8] R. McCreadie, C. Mcdonald, and I. Ounis, "Comparing Distributed Indexing: To MapReduce or Not?", 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009.

[9] J. Lin, D. Metzler, T. Elsayed, and L. Wang. "Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search". Proceedings of the Eighteenth Text REtrieval Conference (TREC 2009), November 2009.

[10] S. Heinz, J. Zobel, and H. E. Williams, "Burst tries: A fast, efficient data structure for string keys", ACM Transactions on Information Systems, vol 20(2), pp. 192–223, 2002.

[11] M. Harris, "Optimizing Parallel Reduction in CUDA", available at http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf. Access date: 09/01/2010.

[12] G. K. Zipf, Human Behavior and the Principle of Least Effort : An Introduction to Human Ecology. Addison Wesley, Cambridge, Mass., 1949.