Using Graphics Processors for High Performance IR Query Processing

Shuai Ding
Polytechnic Inst. of NYU
Brooklyn, NY 11201
sding@cis.poly.edu

Jinru He
Polytechnic Inst. of NYU
Brooklyn, NY 11201
jhe@cis.poly.edu

Hao Yan
Polytechnic Inst. of NYU
Brooklyn, NY 11201
hyan@cis.poly.edu

Torsten Suel*
Yahoo! Research
Sunnyvale, CA 94089
suel@poly.edu

ABSTRACT

Web search engines are facing formidable performance challenges due to data sizes and query loads. The major engines have to process tens of thousands of queries per second over tens of billions of documents. To deal with this heavy workload, such engines employ massively parallel systems consisting of thousands of machines. The significant cost of operating these systems has motivated a lot of recent research into more efficient query processing mechanisms.

We investigate a new way to build such high performance IR systems using graphical processing units (GPUs). GPUs were originally designed to accelerate computer graphics applications through massive on-chip parallelism. Recently a number of researchers have studied how to use GPUs for other problem domains such as databases and scientific computing [9, 8, 12]. Our contribution here is to design a basic system architecture for GPU-based high-performance IR, to develop suitable algorithms for subtasks such as inverted list compression, list intersection, and top-k scoring, and to show how to achieve highly efficient query processing on GPU-based systems. Our experimental results for a prototype GPU-based system on 25.2 million web pages shows promising gains in query throughput.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval.

General Terms

Algorithms, Performance

Keywords

Search Engines, Query processing, Index Compression, GPU

1. INTRODUCTION

Due to the rapid growth of the web and the number of web users, web search engines are faced with enormous performance challenges. Current large-scale search engines are based on data sets of many terabytes, and have to be able to answer tens of thousands of queries per second over tens of billions of pages. At the same time, search engines also have to accommodate demands for increased result quality and for new features such as spam detection and personalization.

To provide high throughput and fast response times, current commercial engines use large clusters consisting of thousands of servers, where each server is responsible for searching a subset of the data (say, a few million pages). This architecture successfully distributes the heavy workload over many servers. Thus, to maximize overall throughput, we need to maximize throughput on a single machine. This problem is not trivial at all since even a single machine needs to process many queries per second. To deal with this workload, search engines use many performance optimizations including index compression, caching, and early termination.

In this paper, we investigate a new approach to building web search engines and other high-performance IR systems using Graphical Processing Units (GPUs). Originally designed to accelerate computer graphics applications through massive on-chip parallelism, GPUs have evolved into powerful platforms for more general classes of compute-intensive tasks. In particular, researchers have recently studied how to apply GPUs to problem domains such as databases and scientific computing [9, 8, 12, 19, 11]. Given their extremely high computing demands, we believe that search engines provide a very interesting potential application domain for GPUs. However, we are not aware of previous published work on using GPUs in this context. Building an efficient IR query processor for GPUs is a non-trivial task due to the challenging data-parallel programming model provided by the GPU, and also due to the significant amount of performance engineering that has gone into CPU-based query processors.

We make several contributions here, described in more detail later. We outline and discuss a general architecture for GPU-based IR query processing that allows integration of the existing performance optimization techniques. We then provide data-parallel algorithms and implementations for major steps involved in IR query processing, in particular index decompression, inverted list traversal and intersection, and top-k scoring, and show how these techniques compare to a state-of-the-art CPU-based implementation. Finally, we study how to schedule query loads on hybrid systems that utilize both CPUs and GPUs for best performance.

2. BACKGROUND AND RELATED WORK

For a basic overview of IR query processing, see [24]. For recent work on performance optimizations such as index compression, caching, and early termination, see [2, 23, 6].

We assume that we are given a collection of N documents (web pages covered by the search engine), where each document is uniquely identified by a document ID (docID) between 0 and $N-1$. The collection is indexed by an *inverted index* structure, used by all major web search engines, which allows efficient retrieval of documents containing a particular set of words (or *terms*). An inverted index consists of many

*Current Affiliation: CSE Dept., Polytechnic Inst. of NYU

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

inverted lists, where each inverted list I_w contains the docIDs of all documents in the collection that contain the word w. Each inverted list I_w is typically sorted by document ID, and usually also contains for each docID the number of occurrences of w in that document and maybe the locations of these occurrences in the page. Inverted indexes are usually stored in highly compressed form on disk or in main memory, such that each list is laid out in a contiguous manner.

Given such an inverted index, the basic structure of query processing is as follows: The inverted lists of the query terms are first fetched from disk or main memory and decompressed, and then an intersection or other Boolean filter between the lists is applied to determine those docIDs that contain all or most of the query terms. For these docIDs, the additional information associated with the docID in the index (such as the number of occurrences) is used to compute a score for the document, and the k top-scoring documents are returned. Thus, the main operations required are index decompression, list intersection, and top-k score computation.

Index Compression: Compression of inverted indexes reduces overall index size as well as the total amount of disk and main memory data transfers during query processing. There are many index compression methods [24]; the basic idea in most of them is to first compute the differences (gaps) between the sorted docIDs in each inverted list. We then apply a suitable integer compression scheme to the gaps, which are usually much smaller than the docIDs (especially for long inverted lists). During decompression, the gaps are decoded and then summed up again in a prefix sum type operation. In our GPU-based query processor, we focus on two compression methods that are known to achieve good compression ratios and that we believe are particularly suitable for implementation on GPUs: the well-known *Rice coding* method [24], and a recent approach in [25, 13] called *PForDelta*.

To compress a sequence of gaps with Rice coding, we first choose an integer b such that 2^b is close to the average of the gaps to be coded. Then each gap n is encoded in two parts: a quotient $q = \lfloor n/(2^b) \rfloor$ stored in unary code, and a remainder $r = n \bmod 2^b$ stored in binary using b bits. While Rice decoding is often considered to be slow, we consider here a new implementation recently proposed in [23] that is much faster than the standard one. The second compression method we consider is the PForDelta method proposed in [25], which was shown to decompress up to a billion integers per second on current CPUs. This method first determines a b such that most of the gaps in the list (say, 90%) are less than 2^b and thus fit into a fixed bit field of b bits each. The remaining integers, called *exceptions*, are coded separately. Both methods were recently evaluated for CPUs in [23], and we adopt some of their optimizations.

List Intersection and DAAT Query Processing: A significant part of the query processing time is spent on traversing the inverted lists. For large collections, these lists become very long. Given several million pages, a typical query involves several MBs of compressed index data that is fetched from main memory or disk. Thus, list traversal and intersection has to be highly optimized, and in particular we would like to be able to perform decompression, intersection, and score computation in a single pass over the inverted lists, without writing any intermediate data to main memory.

This can be achieved using an approach called Document-At-A-Time (DAAT) query processing, where we simultaneously traverse all relevant lists from beginning to end and compute the scores of the relevant documents [24, 5, 15]. We maintain one pointer into each inverted list involved in the query, and advance these pointers using forward seeks to identify postings with matching docIDs in the different lists. At any point in time, only the postings currently referenced by the pointers must be available in uncompressed form. Note that when we intersect a short list (or the result of intersecting several lists) with a much longer list, an efficient algorithm should be able to skip over most elements of the longer list without uncompressing them [18]. To do this, we split each list into chunks of, say, 128 docIDs, such that each chunk can be compressed and decompressed individually. DAAT can implement these types of optimizations in a very elegant and simple way, and as a result only a part of the inverted lists needs to be decompressed for typical queries. DAAT is at first glance a sequential process, and to get good performance on GPUs we need to find data-parallel approaches that can skip large parts of the lists.

Score Computation: Web search engines typically return to the user a list of 10 results that are considered most relevant to the given query. This can be done by applying a scoring function to each document that is in the intersection of the relevant inverted lists. There are many scoring functions in the IR literature that take into account features such as the number of occurrences of the terms in the document, the size of the document, the global frequencies of the terms in the collection, and maybe the locations of the occurrences in the documents. In our experiments here, we use a widely used ranking function called BM25, part of the Okapi family of ranking function; see [22] for the precise definition. We could also use a cosine-based function here, of course; the exact ranking function we use is not important here as long as it can be efficiently computed from the data stored in the index, in particular docIDs and frequencies, plus document sizes and collection term frequencies that are kept in additional global tables. Scoring is performed immediately after finding a document in the intersection.

During traversal of the lists in a CPU-based DAAT implementation, the current top-k results are maintained in a small memory-based data structure, usually a heap. When a new document in the intersection is found and scored, it is compared to the current results, and then either inserted or discarded. This sequential process of maintaining a heap structure is not suitable for GPUs, and we need to modify it for our system. In contrast, implementation of the actual scoring function is trivial and highly efficient on GPUs.

Graphical Processing Units (GPUs): The current generations of GPUs arose due to the increasing demand for processing power by graphics-oriented applications such as computer games. Because of this, GPUs are highly optimized towards the types of operations needed in graphics, but researchers have recently studied how to exploit their computing power for other types of applications, in particular databases and scientific computing [9, 8, 12, 19, 11]. Modern GPUs offer large numbers of computing cores that can perform many operations in parallel, plus a very high memory bandwidth that allows processing of large amounts of data. However, to be efficient, computations need to the carefully structured to conform to the programming model offered by the GPU, which is a data-parallel model reminiscent of the massively parallel SIMD models studied in the 1980s.

Recently, GPU vendors have started to offer better support for general-purpose computation on GPUs, thus removing

some of the hassle of programming them. However, the requirements of the data-parallel programming model remain; in particular, it is important to structure computation in a very regular (oblivious) manner, such that each concurrently executed thread performs essentially the same sequence of steps. This is challenging for tasks such as decompression and intersection that are more adaptive in nature. One major vendor of GPUs, NVIDIA, recently introduced the Compute Unified Device Architecture (CUDA), a new hardware and software architecture that simplifies GPU programming [1]. Our prototype is based on CUDA, and was developed on an NVIDIA GeForce 8800 GTS graphics card. However, other cards supporting CUDA could also be used, and our approach can be ported to other programming environments.

Probably the most closely related previous work on GPUs is the very recent work in [12, 11]. The work in [11] addresses the problem of implementing map-reduce operations on GPUs; this is related in that map-reduce is a widely used framework for data mining and preprocessing in the context of search engines. However, this framework does not apply to the actual query processing in such systems, and in general the problems considered in [11] are quite different from our work. The recent work in [12] is more closely related on a technical level in that query processing in search engines can be understood as performing joins on inverted lists. Also, one of the join algorithms in [12], a sort-merge join, uses an intersection algorithm very similar to the case of our intersection algorithm with a single level of recursion. However, beyond this high-level relationship, the work in [12] is quite different as it does not address issues such as inverted index decompression, integration of intersection and decompression for skipping of blocks, and score computation, that are crucial for efficient IR query processing.

Parallel Algorithms: Our approach adapts several techniques from the parallel algorithms literature. We use previous work on parallel prefix sums [3], recently studied for GPUs in [10, 17], and on merging sorted lists in parallel [7], which we adapt to the problem of intersecting sorted lists.

3. CONTRIBUTIONS OF THIS PAPER

We study how to implement high-performance IR query processing mechanisms on Graphical Processing Units (GPUs). To the best of our knowledge, no previous work has applied GPUs to this domain. Our main contributions are:

(1) We present a new GPU-based system architecture for IR query processing, which allows queries to be executed on either GPU or CPU and that contains an additional level of index caching within the GPU memory.

(2) We describe and evaluate inverted index compression techniques for GPUs. We describe how to implement two state-of-the-art methods, a version of PForDelta [25, 13] and an optimized Rice coder, and compare them to CPU-based implementations. Our implementation of PForDelta achieves decompression rates of up to 2 billion docIDs per second on longer inverted lists.

(3) We study algorithms for intersecting inverted lists on GPUs, based on techniques from the literature on parallel merging. In particular, we show how to integrate compression and intersection such that only a small part of the data is decoded in typical queries, creating a data-parallel counterpart to DAAT query processing.

(4) We evaluate a basic version of our GPU-based query processor on the 25.2 million pages of the TREC GOV2 data set and associated queries, and compare it to an optimized CPU-based query processor developed in our group. We show that the GPU-based approach achieves faster processing over all queries, and much faster processing on expensive queries involving very long lists and on disjunctive queries.

(5) We study the query throughput of different system configurations that use either CPU, or GPU, or both, for query processing under several scheduling policies.

The remainder of this paper is organized as follows. In the next section, we discuss some assumptions and limitations of our work. Section 5 outlines the proposed GPU-based query processing architecture. In Section 6 we study index compression schemes for GPUs, and Section 7 looks at list intersection algorithms. Section 8 evaluates the performance of the full query processing mechanism on the 25.2 million pages from the TREC GOV2 collection. In Section 9 we evaluate scheduling mechanisms for systems that use both GPUs and CPUs. Finally, Section 10 provides concluding remarks. Our code is available at `http://cis.poly.edu/westlab/`.

4. ASSUMPTIONS AND LIMITATIONS

In addition to query processing, large web search engines need to perform many other operations including web crawling, index building, and data mining steps for tasks such as link analysis and spam and duplicate detection. We focus here on query processing, and in particular on one phase of this step as explained further below. We believe that this part is suitable for implementation on GPUs as it is fairly simple in structure but nonetheless consumes a disproportionate amount of the overall system resources. In contrast, we do not think that implementation of a complete search engine on a GPU is currently realistic.

Modern search engines use far more complex ranking functions than the simple BM25 variant used by us. Such engines often rely on hundreds of features, including link-based features derived, e.g, using Pagerank [4], that are then combined into an overall scoring function using machine learning techniques. To implement such a scoring function, search engines typically divide query processing into two phases: An initial phase uses a fairly simple ranking function, such as BM25 together with some global document score such as Pagerank, to select a set of candidate documents, say a few hundred or thousand. In a second phase, the complete machine-learned scoring function is applied to only these candidates to select the overall top results. Thus, our approach can be seen as implementing the first phase, which aims to select promising candidates that the complete scoring function should be applied to. In contrast, the second phase has a very different structure, and implementing it on a GPU would be an interesting and challenging problem for future work.

In our experiments, we assume that the entire index is in main memory, or at least that caching performs well enough to effectively mask disk access times. Of course, if disk is the main bottleneck, then any approach based on optimizing CPU or GPU performance is futile. In general, large-scale search engine architectures need to balance CPU, main memory, and disk cost and performance – if processor (CPU or GPU) throughput is improved, a savvy system designer will exploit this, e.g., by using fewer processors or adding data, disks, or main memory in order to rebalance the architecture at a more cost-efficient point.

Finally, we do not consider index tiering and early termination techniques, which allow answering of top-k queries without traversing the full index structures. Such techniques could in principle be added to our approach. In particular, tiering [21, 6] could be trivially combined with our approach as it does not impact query processing within a node.

5. GPU-BASED SEARCH ARCHITECTURE

We now describe and discuss the proposed query processing architecture. We start out with a typical (slightly simplified) CPU-based architecture, shown in Figure 1, where a query enters the search engine through a query integrator node. This query integrator first checks a local cache of query results, the *result cache*. If the same query was recently issued by another user of the engine, then the result of that query may already be contained in the result cache and can be directly returned to the user. Otherwise, the query integrator forwards the query to a number of machines, each responsible for a subset of the collection. Each machine has an inverted index structure for its own subset, and computes the top-k results on this subset. The results are returned to the master, who then determines the overall top-k results. The index inside each machine is either kept completely in main memory, or it resides on disk but a substantial amount of main memory is used to cache parts of the index. In practice, cache hit rates of 90% or more are obtained as long as about 20% or more of the index data can be kept in main memory. In our setup, we assume that disk is not the main bottleneck.

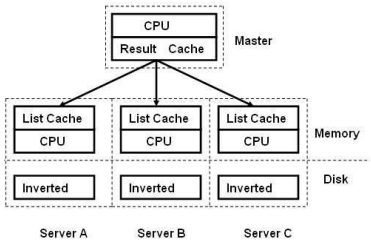

Figure 1: A basic CPU-based search engine cluster.

Our GPU-based architecture can use the same distributed architecture, but each node contains CPUs as well as GPUs for query processing. We show the structure of a single node in our system in Figure 2. The compressed inverted index is either completely in main memory, or stored on disk but partially cached in main memory for better performance. The GPU itself can access main memory only indirectly, but has its own global memory (640 MB in our case) plus several specialized caches, including shared memory shared by many threads. Data transfers between main memory and GPU memory are reasonably fast (a few GB/s), and can be performed while the CPU and GPU are working on other tasks. Memory bandwidth between the GPU and its own global memory is in the tens of GB/s and thus higher than typical system memory bandwidths; however, for best performance memory accesses need to be scheduled to avoid bank conflicts.

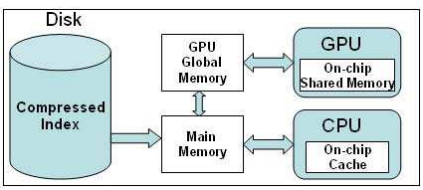

Figure 2: Architecture of a GPU-based system.

Each query is first preprocessed by the CPU, and the corresponding inverted lists are retrieved from disk if not already in main memory. In the case where both CPU and GPU participate in query processing, the CPU then decides whether to schedule the query on CPU or GPU. In addition to the index cache in main memory, the GPU also maintains its own cache of index data in GPU global memory to avoid unnecessary data transfers from main memory. This cache is smaller than the main memory cache, and thus the hit rate will be lower, say around 50% to 80% instead of over 90%. The assignment of a query to CPU or GPU is done based on the current load of each processor, the characteristics of the query (certain types of queries may run better on CPU and others on GPU), and the availability of index data in GPU memory. If data needs to be transferred to GPU global memory to run the query, then this transfer is performed while processing other queries. To get good utilization of both processors, we assume a certain level of concurrency that allows reordering of queries as long as queries are answered within a certain acceptable delay, say 500 ms or less.

A query is executed on the GPU by uncompressing and intersecting the inverted lists from shortest to longest, similar to *Term-At-A-Time* (TAAT) processing on CPUs. However, to get the performance benefits of skipping over parts of the longer lists enjoyed by DAAT, we split list intersection into several phases, an initial phase that determines which parts of the longer list need to be decompressed, the decompression of those parts, and then the intersection of the decompressed parts with the shortest list (or the intersection of the lists already processed). We will show that the amount of docID data decompressed under this scheme is very close to that under DAAT, while allowing an efficient data-parallel implementation on the GPU. After each intersection, we also fetch the frequencies of any docIDs in the intersection and update their BM25 scores. Finally, the top-k results are returned to the CPU. More details are provided in the next few sections.

6. GPU-BASED LIST DECOMPRESSION

In this section, we present two index decompression algorithms for GPUs based on Rice coding and PForDelta, two methods known to achieve good compression on inverted lists. Both can be efficiently implemented using parallel prefix sums, a basic primitive in parallel computation [3], and thus are good candidates for implementation on GPUs.

6.1 Rice Coding for GPUs

Recall the description of Rice coding from Section 2, where each integer is encoded into a fixed-width binary and a variable-width unary part. As in [23], we keep the unary and binary parts separately in two different bit fields. The important new observation for our purpose is that we can decompress the code by running two separate parallel prefix sums, one on the binary codes, and the other on the unary codes, where each prefix is tuned slightly differently as described below. Then each uncompressed docID can be retrieved by summing up its binary prefix sum and 2^b times the corresponding unary prefix sum. Note that as a result we directly obtain the original docIDs, not just the gaps between docIDs. In summary, decompression reduces to two customized prefix sums, plus a shift and addition for each element.

Parallel Prefix Sum: Our implementation relies on an highly efficient implementation of parallel prefix sum. There are two types of prefix sum operations, inclusive prefix sum,

where the sum includes the element itself, and exclusive prefix sum, where it does not. In this paper, we use the inclusive prefix sum, that is, given an array $< a_0, a_1, ..., a_{n-1} >$, the resulting prefix sum is an array $< a_0, (a_0 + a_1), ..., (a_0 + a_1 + ... + a_{n-1}) >$. The prefix sums problem has been studied extensively in the parallel processing literature [3]. An efficient implementation on GPUs was presented in [10], and some additional optimizations are described in [17]. The basic idea in [10], based on [3], is to perform the computation using a tree structure that is traversed first bottom-up and then top-down, resulting in a fast and work-optimal parallel implementation. This idea is further improved in [17], which primarily optimizes memory accesses by loading more data per step from GPU memory.

We adopt the best implementation in [17] and add some optimizations of our own since our prefix problems are somewhat more specialized than the general case. First, we design a bit-oriented rather than integer-oriented prefix sum as we are dealing with numbers of fixed bit width much smaller than 32. Second, we *localize* the prefix sum to blocks of a certain size, say 128 elements, by storing one base address with respect to which the prefix sums are performed in each block. This is motivated by the fact that inverted lists are typically compressed in blocks such that each block can be decompressed independently; in this case each prefix sum only accumulates blocks of, say, 128 consecutive elements. We call such a prefix sum associated with blocks a localized prefix sum. This partition of inverted lists into blocks not only results in faster prefix sum operations, but also allows us to skip many blocks completely, as in CPU-based DAAT query processing, but in a data-parallel manner.

Index Organization: As explained in Section 2, Rice coding encodes an integer (the gap between two consecutive docIDs) by choosing a number of bits b such that 2^b is close to the average of all the gaps, and then representing each integer as $q \cdot 2^b + r$ for some $r < 2^b$. Then the integer is encoded in a unary part, consisting of q 1s followed by a 0, and a binary part of b bits representing r. In our implementation, all the binary and unary codes are stored separately in two bit vectors, a unary list I_u and a binary list I_b. To take advantage of the localized prefix sum operation, and to enable skipping in the underlying inverted list, we additionally store for each block of 128 bits in I_u, and for each block of 128 b-bit number in I_b, the sum of all elements preceding this block. (These arrays of sums could themselves be compressed recursively, and then decompressed with very minor overhead before accessing a list, but in our implementation we simply store them as 32-bit integers.) Thus, all prefix sums within a block are performed with these two values as base sums.

Processing Unary Codes: To process the unary codes in I_u, we use a bit-wise prefix sum that interprets I_u not as a sequence of unary codes but just as a bit array. After a prefix sum on the n bits of I_u we obtain an integer array of size n where the i-th integer contains the number of 1s up to and including bit i in the bit array. Thus, in the position corresponding to the j-th 0 value in the bit vector, we obtain the sum of all unary codes for the first j gaps, which multiplied by 2^b gives us the total unary contributions of all numbers up to the j-th gap. Next, we compact the result array so that only the values in positions with 0 values in the bit vector are retained; now the value for the j-th gap is in the j-th position of the result array. Thus, the prefix sum produces an array of n integers from our array of n bits, while compaction reduces this to an array with one entry for each docID in the list.

Decompressing Binary Codes and Combining: The binary codes can be decoded in a simpler way, without compaction step, since each binary code has exactly b bits. Thus, we now use a prefix sum operation on b-bit numbers, where b is usually much smaller than the 32 bits per element used by standard prefix sums. This saves on memory access costs during prefix computation, at the cost of some complexity in our program. Finally, we can now directly compute the i-th docID by computing the sum of the i-th binary prefix sum and 2^b times the i-th unary prefix sum (after compaction).

6.2 Decompression using PForDelta

Recall that PForDelta first selects a value b such that most gaps are less than 2^b, and then uses an array of b-bit values to store all gaps less then 2^b while all other gaps are stored in a special format as exceptions. However, while exceptions we organized in a linked list in [25, 13, 23], we store them in two arrays. More precisely, for each exception we store its lower b bits with all the non-exceptions, and then store the higher *overflow* bits and the offsets of the exceptions in two separate arrays. We then recursively apply PForDelta to compress these two arrays. During decompression, we recursively decompress the two arrays specifying the overflow bits and offsets of the exceptions. Then we decompress the other values using a b-bit oriented prefix sum, and add the higher bits of the exceptions to the positions specified by the offsets. In contrast to [25, 13, 23], we select a single value of b for the entire inverted list, rather than potentially a different value for each block. This avoids memory alignment and other implementation issues that arise if we have a different bit-width for the prefix sums in each block. We store the lowest b bits of all gaps in their corresponding b-bit slot, and for any exceptions we store the additional higher bits and the offset of the exception in corresponding entries of two additional arrays. As it turns out, using the same b for an entire inverted list would result in worse compression if we stored exceptions as in [23]. Instead, we recursively compress the array with the overflow bits, and the array with the exception offsets (more precisely, the gaps between the offsets), using again PForDelta. This provides good compression while giving us a uniform b-bit slot size for the entire inverted list. As before, decompression and prefix computation can be performed in a block-wise manner by storing one uncompressed docID for each block. (We use a block size of 512 elements for our GPU-based PForDelta.) More details on how to skip over blocks during query processing are provided next.

6.3 Experimental Results

We describe our setup, which we also use in later sections. The data set we used in our experiments is the TREC GOV2 data set of 25.2 million web pages, and we selected 1000 random queries from the supplied query logs. On average, there were about 3.74 million postings in the inverted lists associated with each query. All experiments are run on a machine with a 2.66GHz Intel Core2 Duo CPU (only a single core was used), 8GB of memory, and an NVIDIA GeForce 8800 GTS graphics card with 640 MB of global memory and 96 stream processors. We focus on decompression speed, since compression is a one-time cost while decompression happens constantly during query processing. All methods compress at least tens of millions of postings per second.

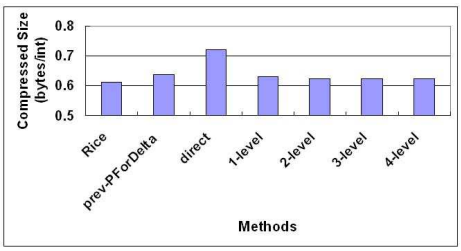

Figure 3: Compression in bytes per docID for different methods and recursion levels. The second bar from the left corresponds to the PForDelta method with a different selection of b for each block used in [23].

Algorithm	CPU	GPU
Rice	310.63	305.27
PForDelta	1165.13	1237.57

Table 1: Decompression speeds on CPU and GPU in millions of integers per second.

We first compare the average compressed size of a docID under the different algorithms, in particular Rice Coding, our version of PForDelta without recursive compression and with one to four levels of recursion, and the version of PForDelta from [23]. In Figure 3, we see that Rice decoding gives the best compression ratio, as expected. Our GPU-based PForDelta without recursion is much worse than the version in [23], which can select a different b for each block. However, a single level of recursive compression already achieves a slightly smaller compressed size than the PForDelta in [23], while additional levels give only slight extra benefits. In the following, we always use a single level of recursion.

Next we compare decompression speeds on GPU and CPU, shown in Table 1, where the speed is represented in millions of integers uncompressed per second. From the table we can see that the GPU achieves better performance than CPU for PForDelta but is very slightly slower for Rice coding, and that PForDelta is much faster than Rice coding. We note however one difference between the CPU and GPU methods: Our GPU implementations directly compute the actual docIDs during decompression, while the CPU versions only obtain the gaps. This is done because we found that in CPU-based query processors, the summing up of the gaps to obtain docIDs is most efficiently done during the forward seeks for finding matching docIDs in DAAT query processing rather than during decompression. Thus, the results for GPU in Table 1 contain some work that is done during other phases of the query processing in the CPU, and the actual relative performance of GPUs is somewhat better than shown.

Next we look at the decompression speed of PForDelta as we vary the lengths of the inverted lists, shown in Figure 4. We show on the x-axis the length of the list in 1000s of docIDs; in fact, we bin the list lengths into ranges so that, e.g., a value of 32 on the x-axis represents list lengths between 16000 and 32000 (this is necessary since we use real rather than synthetic inverted list data). As discussed above, the GPU-based implementations compute actual docIDs while the ones for CPU only compute gaps. To evaluate the impact of this decision, we also tested a version of PForDelta on GPUs that does not perform the final localized prefix sum on b-bit numbers and thus only returns gaps. From Figure 4, we see that the GPU implementation without localized prefix sum performs much better than the other methods, and decodes up to 2.5 billion integers per second for longer lists.

We also observe that the GPU-based methods are much

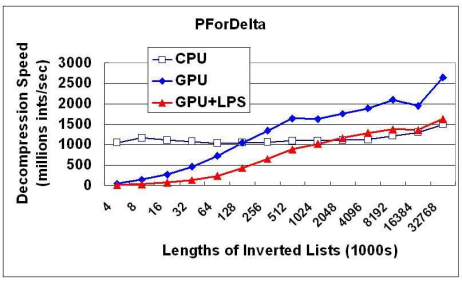

Figure 4: Decompression speed of PForDelta on CPU and GPU, for inverted lists of different lengths, in millions of integers per second. GPU+LPS is the algorithm with localized prefix from Table 6.3, while GPU is a version without prefix that only computes gaps between docIDs.

worse than the CPU for short lists, and outperform the CPU for longer lists. There are two reasons. First, there are startup costs involved in running tasks on the GPU that are significant for short lists. Second, a certain data size is needed to exploit the full parallelism of the 96 stream processors in the GPU during all phases of the computation (though additional fine tuning may improve this a bit). If only some of the stream processors are used, then the more flexible programming model and higher clock frequency of the CPU (2.66Ghz) win out over the 500Mhz frequency of the GPU. On our data set, the average length of the lists occurring in queries is fairly large; as a result we get the slight advantage for GPUs in Table 1. We expect GPUs to do even better on larger data sets.

7. GPU-BASED LIST INTERSECTION

In this section, we describe how to perform intersections between inverted lists during query processing. We first define a basic operation, *Parallel Merge Find*, and then use this to introduce our merging algorithms. These algorithms are further developed in the next section, when we integrate intersection with decompression and score computation.

7.1 Intersect, Merge, and Parallel Merge Find

Our intersection algorithms are based on parallel algorithms for merging sorted lists. Note that in principle, intersection does not require merging, and there are other solutions based, e.g., on hashing. We select a merging-based approach because of the availability of suitable highly efficient parallel algorithms for merging [7], but also for another more subtle reason that will become apparent in the next section: We can use the same merge-based approach not just to perform the actual intersection, but also to select those blocks that need to be uncompressed. This turns out to be crucial for performance. An intersection algorithm that does not use the linear ordering of docIDs would not work for this task.

To describe our merging algorithms, we define an operation called *Parallel Merge Find*. Given two sorted and uncompressed inverted lists, list A of m numbers and list B of n numbers, Parallel Merge Find in parallel finds for each element A_i in A, where $i = 1, 2, ..., m$, the pair B_j, B_{j+1} in B such that $B_j < A_i \leq B_{j+1}$. Parallel Merge Find is of course closely related to the problem of merging two lists of integers, and our resulting intersection algorithm is motivated by the classical approach in [7]. An obvious way to implement Parallel Merge Find on a GPU involves using one thread for each element A_i to do an independent binary search in list B.

7.2 Intersection Algorithms for GPUs

As discussed in Section 2, although DAAT works well in a CPU-based architecture, the sequential nature of the list traversal via pointer movements makes it unsuitable for efficient parallel processing. Instead, we use a merge-based parallel intersection algorithm based on applying the above Parallel Merge Find operation as follows: Given two uncompressed lists A and B, shown in Figure 5, we first select a small subset of splitter elements in A, say elements 8, 14, and 24, and use Parallel Merge Find to find the corresponding values (or closest pairs of values) in B. Note that this implicitly partitions both A and B into segments as show in Figure 5. To intersect the complete lists A and B, we now only need to intersect the elements in one section of A with the elements in the corresponding section of B, by calling Parallel Merge Find on each segment. This process can be repeated recursively, such that the segments are further divided into smaller sub-segments. Note that the basic idea of this recursive merging algorithm is well established in the parallel algorithms literature [7].

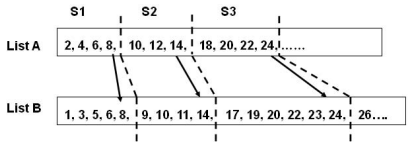

Figure 5: An example of parallel intersection, where 8, 14, and 24 occur in both lists and thus direct matches are found rather than neighboring elements.

For queries involving k lists, we order the lists from shortest to longest, and then first intersect the shortest with the second-shortest list. In general, we process all lists in order of increasing length, where in each step we intersect the output of the previous intersection with the next longer list. We note that this is essentially a Term-At-A-Time (TAAT) approach, as opposed to the more popular DAAT approach used in current CPU-based systems. However, we will show in Section 8 how to avoid uncompressing most parts of the longer lists during intersection, thus achieving one of the main benefits of the DAAT approach on GPUs.

7.3 Experimental Results

We now perform a preliminary evaluation of our intersection algorithms. For now we assume that we have two lists that are in completely uncompressed form. The performance of our algorithms is summarized in Figure 6, where we consider the direct algorithm that performs one binary search into B for each element in A, as well as algorithms performing 1 to 4 recursive levels of splitting the lists into segments as described above. Figure 6 assumes two lists of equal length, and varies the list length from 5000 to 16 million elements. We see that for longer lists the recursive approaches perform much better than the direct approach. In particular, using two or three levels of recursion we get close to best performance over the entire range of values, with costs below $4ns$ per element for larger lists. We also ran experiments (not shown due to space constraints) where we fixed the length of the shorter list to 200000 and then vary the length of the longer list from 200000 to 12.8 million. As is to be expected, we see a moderate rise in cost from about 4 to $15ns$ per element in the shorter list as the length of the longer list increases. Overall, three levels of recursion appear to perform best, and in the following we focus on this case.

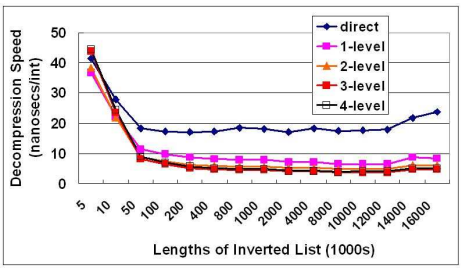

Figure 6: Intersection speeds with different levels of recursions for two lists of the same length, where list length is varied from 5000 to 16 million. We show performance in nanoseconds per integer in one of the lists.

8. RANKED QUERY PROCESSING

As discussed in Section 2, a state-of-the-art query processor involves not only compression and list intersection, but also other techniques that support skipping over parts of the index and scoring and accumulation of top-k results. In this section, we complete our GPU-based query processing architecture by integrating all the necessary techniques.

8.1 Advanced Intersection with Skips

In our basic intersection algorithm discussed in the previous section, we assumed that both lists are completely decompressed before we intersect them. We now show how to remove this assumption. In our solution, we assume that at the beginning, the shortest list is uncompressed, and then in each step we intersect the result from previous intersections (or in the first step, the shortest list) with the next longer list in compressed form. To do so, we add one additional step before the intersection, where we determine which of the blocks of the longer list need to be decompressed.

Suppose we want to intersect two lists, a shorter list A that is already decompressed, and a longer list B that is still in compressed form. We assume a block-wise compression scheme, where for each block of, say, 128 docIDs we store the first value (or sometimes the last value of the previous block) in uncompressed form in a separate much smaller list B^*. Now suppose that instead of intersecting A and B, we first perform a Parallel Merge Find (or its recursive variants) between A and B^*, and that for two consecutive entries B^*_i and B^*_{i+1} there is no element in A that has a value in between these two values. This implies that the block in B that is delimited by B^*_i and B^*_{i+1} cannot possibly intersect with any element in A, and does not have to be decompressed. If A is much shorter than B, which is common when intersecting the result of previous intersections with the longer lists in the query, then most blocks of B can be skipped.

Thus, our advanced intersection algorithm involves three steps. First, we determine the blocks in the longer list that need to be decompressed by using a Parallel Merge Find operation. Next, we decompress only those blocks from the longer list that are needed, and finally we use the algorithm from the previous section to complete the intersection. We note that overall, compressing inverted lists in blocks and keeping one docID per block in uncompressed form in a separate array serves two purposes in our system. First, we can skip over many blocks without decompressing them, and second, we can use a localized prefix sum instead of a slower global prefix sum during decompression. Finally, we point out that the number of blocks that can be skipped under this method is in fact the same as the number of blocks skipped under a

variant of DAAT query processing in which forward seeks are used in all lists except for the shortest one, where we process one element at a time. We omit the formal argument due to space constraints. (We note however that in our recursive PForDelta we always decompress the complete arrays for the exceptions, which are much smaller than the full lists.)

8.2 Ranked Query Processing

In our query processing system, ranked query processing involves two more steps after performing an intersection between the inverted lists: One step is to decompress the frequency values corresponding to the candidate docIDs and use these values to calculate BM25 scores [22]. The other step is to select the k documents with the highest scores. Both steps are executed in parallel on the GPU as follows.

Score Computation: In systems with block-wise compression schemes, typically an entire block of values is decompressed if we need any one of the values. However, for frequencies this is actually not necessary for two reasons. First, frequencies are stored as absolute values, not gaps between values, and thus we do not need to sum up any preceding values in the block to retrieve a value. Second, in PForDelta, each value that is not an exception occupies a fixed b-bit slot. Thus, to fetch a frequency value in a particular slot of a block (say, the i-th entry in the block), we first need to check if the entry is an exception; if not, we directly retrieve it from its slot, and otherwise we locate and add the overflow bits from the separate array. Thus, we can decompress and retrieve individual frequency values.

In DAAT processing on CPUs, we usually only fetch the frequencies and compute the scores for those documents that are in the intersection of all the lists. In our approach, this is not easy to do as we process terms one at a time. Instead, after each intersection, and also for the entire shortest list, we fetch the frequency values of the current candidate documents for the term currently being processed, compute the per-term score, and add it to a current scores for the document. Thus, we accumulate scores for documents one term at a time. Fortunately, the overhead for this is not too high, as the actual computation of scores is very fast in GPUs.

Choosing Top-k Documents: CPU-based systems typically use a heap structure in order to maintain the current top-k results during intersection and scoring. In our setup, this is not suitable due the sequential nature of the heap structure and the fact that we accumulate scores one term at a time. We could of course sort all the final scores, but this is relatively slow even with very fast GPU implementations such as [8] (since computing top-k results is an easier problem than sorting). We found that a 2- or 3-level approach based on a high-degree heap works well for values of k up to several thousand. To initialize this heap, we divide the candidates into groups of some size, say 512 per group, and compute the maximum score in each group. We then take the maximum scores of the groups, and divide them into groups of size 512, then take the maximum of each such group, and so on. We then repeatedly extract the maximum and update those branches of the tree where the maximum came from.

8.3 Supporting Disjunctive Queries

We have so far focused on intersection-based queries as these are widely used in web search engines. We now discuss disjunctive (OR) queries. We experimented with several approaches, but a brute-force TAAT approach, where we we maintain a score accumulator for every document in the collection, performed best on GPUs. For each query, we initialize this data, and then go over the lists for the query terms one after the other, compute the contribution of each posting according to the ranking function, and add the score to the corresponding accumulator. Finally, we use the same approach as in the conjunctive case to select the top-k elements. We found that for the queries in our query log, where on average there are several millions of postings associated with the query terms, such a direct-mapped approach is preferable to a hash structure that maps to a smaller set of accumulators. For other types of queries, other approaches may be preferable. (We also tried a brute-force approach in our CPU-based system, but did not see any benefits.)

Disjunctive queries are known to be much more expensive than conjunctive queries on CPUs, since we need to compute scores for all encountered postings, not just those in the intersection of the lists. Moreover, all postings have to be decompressed, while conjunctive queries can skip many blocks of postings entirely. There are two simple ideas for improving performance on disjunctive queries; see, e.g., [14]. One is to store precomputed quantized scores instead of frequency values in the index; this significantly reduces the cost of score computation but may increase index size. (Quantized scores usually require about one byte per posting, versus 4 to 5 bits per frequency value under schemes such as PForDelta.)

The other idea is to only compute a full OR for queries that benefit from it. It has been shown [14] that for queries where there is a sufficient number of results in the intersection, conjunctive queries perform as well as disjunctive ones under BM25. Thus, a simple optimization first computes an AND, and then issues another OR query if there are not enough results. (We note that this is also closely related to the WAND style of query processing proposed in [5].) We did not implement quantized scores, but will show results for the second idea of first issuing an AND query, followed by an OR if not enough results are returned.

8.4 Experimental Results

The following experimental results are based on the same setup as before, using the 25.2 million web pages of the TREC GOV2 data set and 1000 queries selected at random from the associated query set. We compare our complete GPU-based query processing system to an optimized CPU-based system under development in our group. Note that the CPU-based system we are comparing to achieves running times that we believe to be competitive with the fastest state-of-the-art systems. In particular, the CPU-based system uses block-wise PForDelta for index compression, DAAT query processing, and optimized score computation using precomputed values for any expressions in BM25 that are based on term and document statistics. While we use BM25 as the scoring function in the experiments, we would expect essentially the same performance for many other simple term-based functions, or for functions that add a precomputed global document score such as Pagerank (provided in a separate array) into the score.

We compare GPU and CPU on three different queries types: conjunctive queries (AND), disjunctive queries (OR), and conjunctive followed by a disjunctive query if there are less than 10 results (AND+OR). All CPU runs were performed on a single core of a 2.66Ghz Intel Core2 Duo E6750 CPU.

As shown in Table 2, on AND queries our GPU-based system manages to only slightly outperform the CPU-based sys-

Algorithm	AND	OR	AND+OR
CPU	8.71	212.72	23.85
GPU	7.66	29.31	9.98

Table 2: Query processing speeds on CPU and GPU, for top-10 AND queries, OR queries, and AND+OR queries. For each method, we show the average cost of a query on the TREC GOV2 set in milliseconds.

	Top-k scoring	Intersection	Decompression
AND	8%	45%	47%

	Initialization	Decompression	Scoring	Top-k
OR	9%	30%	48%	13%

Table 3: Relative costs in percent for the different steps of GPU query processing, for AND and OR queries. For AND queries, top-k scoring contains both score computation and selection of top results, while for OR queries these are listed separately.

tem, though we expect some limited additional gains with further optimization. Note that in these runs, we are assuming that the inverted lists for the query are available in main memory for the CPU, and in global memory for the GPU; we address this assumption in the next section. We also show in Table 3 the relative costs for the three major parts of our GPU-based query processor for AND queries: Decompression, intersection, and scoring and selection of top-k documents. We see that most of the time for AND queries is spent on decompression and intersections. Less than 10% is spent on scoring and identifying top-k results. We found this also to be true for AND queries on the CPU (not shown), since only documents in the intersection are scored.

The situation is different for OR queries, where the GPU substantially outperforms the CPU. For AND+OR queries, the difference is not as large since for most queries only the AND part of the query is evaluated. For OR queries on the GPU, almost 50% is spent on score computation, as shown in Table 3, but this includes the cost of performing memory accesses to update the accumulators (which incurs some memory bank conflicts). Overall, not surprisingly, the GPU excels at large, fairly non-adaptive computations such as a brute-force OR, as opposed to more adaptive computations such as the AND with intersections and skipping.

Note that our CPU implementation of OR did not use precomputed quantized scores. Using such scores would improve performance, but would not bring it close to GPU performance as measurements showed that only about 40% of CPU time for OR was spent on computing scores. We plan to implement and evaluate such optimizations in future work.

Finally, we discuss the impact of query properties on CPU and GPU performance; details are omitted due to space constraints. While query processing costs grow with the number of terms in the query, the relative performance of GPU and CPU remain about the same. We also looked at queries of different *footprint sizes*, defined either as the sum of the inverted list sizes or the size of the shortest list in a query. For OR queries, not surprisingly the CPU is faster for small queries and the GPU for large queries. For OR, GPU significantly outperforms CPU for all but the smallest queries.

9. SCHEDULING ON CPU AND GPU

Under our basic system architecture, both CPU and GPU can be used to process incoming queries, and thus we could improve query throughput by using the GPU as a co-processor that takes care of part of the query load. We now investigate how to best achieve this by assigning queries in an incoming query stream to either CPU or GPU based on the characteristics of the query, the current load on each processor, and the performance requirements for the query.

Some comments about the setup. We assume that queries arrive in the system according to a Poisson process with a particular average arrival rate. Our goal is to schedule these jobs such that (1) no job takes longer than some maximum time (say, 500 ms after arrival for search engines), (2) the average delay of a job is as small as possible, and (3) the system can sustain a high query arrival rate. Each arriving job has two possible costs, a CPU cost that would be incurred if the job is scheduled on CPU, and a GPU cost. Thus, we are dealing with a job scheduling problem on two machines [16] that is NP Complete for the offline case.

Our problem is complicated by several factors. First, we do not actually know the CPU and GPU costs of an incoming query, but have to rely on estimates of these costs based on characteristics such as the lengths of the inverted lists involved. Second, by analyzing our results from previous sections, we found that CPU and GPU costs can diverge quite a lot: Basically, there are some queries that are much more efficient on the CPU, in particular many queries involving short lists or combinations of short and long lists, while queries with many long lists are usually more efficient on GPUs. An interesting consequence of this is that a combined system involving GPU and CPU could potentially be more than twice as fast as a system using either CPU or GPU, if each query is scheduled on the most suitable processor.

Assigning Queries to GPU or CPU: Given a stream of queries arriving one after the other, we need to decide which processor a query should be assigned to. We assume here that each processor has a queue for those queries it has to process. For an incoming query, we would like to assign it to the processor that will process this query more efficiently. However, if one processor is very busy, this might not be the best solution. In our approach we divide queries into three groups, with a queue for each group: (1) Queries that are estimated to be much more suitable for the GPU, (2) queries that are estimated to be much more suitable for the CPU, and (3) all other queries, which may be more suitable for one processor but which can also be at least reasonably efficiently computed on the other. The first two groups of queries are directly processed by the suitable processor, while the third group is essentially in a waiting stage and will be later moved to one of the other groups. Membership in the groups is determined by some threshold values (say, queries that are 20% faster on GPU than on CPU are in the first group) that could be chosen adaptively during the process.

Query Stealing and Scheduling: One disadvantage of the above query assignment policy is that when the query characteristics change temporarily or when processing time is not accurately predicted, the load on CPU and GPU may become unbalanced such that one processor has too much work and the other too little. To deal with this we employ work stealing [20]: When a processor is idle, it first executes the oldest job from the third group, and if there is no such job it will steal a job from the other processor. Moreover, if any job approaches the deadline (i.e., has passed a substantial part of the total time to its deadline), we prioritize this job and schedule it immediately on its preferred processor. Finally, for the first two groups each processor will process

its queries according to a mix of query cost (small jobs should get priority to minimize average delay), deadline, and affinity (run jobs that should definitely run on this processor).

Experimental Results: Our above scheduling algorithm is evaluated on a simulated environment, where we first use machine learning to estimate the performance of incoming queries on CPU and GPU based on the following features: The length of shortest list and the second shortest list, the sum of lengths of all lists, and the number of terms included in the query. We use the M5Rule method from Weka (an open-source machine learning tool) and train on 3000 other queries from the same overall query trace as the 1000 queries we used in our previous experiments. The mean relative prediction error for the query processing time is 22% for GPU cost and 20% for CPU cost.

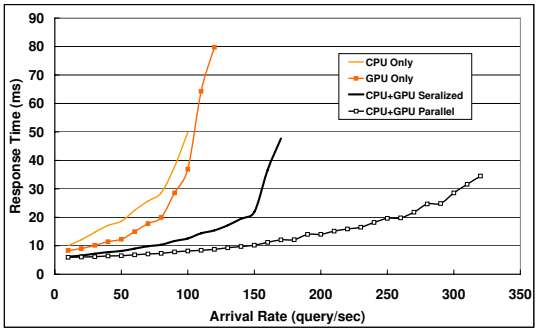

Figure 7: Query processing performance on AND queries for four methods: Using only GPU or CPU, choosing the better one for each query, and using both in parallel.

We compare the query processing performance of four configurations in Figure 7. The configurations are: (1) Using GPU only, (2) CPU only, (3) the better of GPU and CPU but processing only one query at a time, and (4) using CPU and GPU in parallel using the above scheduling method. We set the maximum acceptable delay for each query (deadline) to 500 ms after arrival, and consider a run failed if the arrival rate is so high that a query misses its deadline. Given this constraint we plot the average delay versus the arrival rate. From Figure 7, we see that using both processors in parallel achieves much better performance than all other methods. Using GPU and CPU in parallel we can sustain an arrival rate beyond 300 q/s, versus less than 100 for CPU only.

10. CONCLUSIONS

In this paper we proposed a framework for high-performance IR query processing using GPUs, and discussed and implemented algorithms for performing the various subtasks involved. Our experimental results showed the potential for performance gains from employing GPUs in search engine nodes, particularly in conjunction with CPUs. We are continuing to optimize our implementations and expect additional gains, in particular through tuning of the intersection.

Overall, we believe that we have made a promising first step in this paper, but more work needs to be done to figure out if GPUs can really be a cost-effective platform for search engine query processing. This depends not just on speed, but also on the cost and energy consumption of such devices, and currently a quad-core CPU still provides better value than a GPU. However, we feel that the real motivation for research an general-purpose GPU computing is to identify strengths and weaknesses in current GPUs and related architectures such as multi-core CPUs and cell processors that should be addressed by future processor generations, and to study techniques that will be useful on future CPUs with increasing parallelism.

11. REFERENCES

[1] Nvidia CUDA programming guide, June 2007. http://www.nvidia.com/object/cuda_develop.html.

[2] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. of the 30th Annual SIGIR Conf. on Research and Development in Information Retrieval*, July 2007.

[3] G. Blelloch. Prefix sums and their applications. *In J. H. Reif, editor, Synthesis of Parallel Algorithms*, pages 35–60, 1993.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *7th World Wide Web Conference*, 1998.

[5] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th Conf. on Information and Knowledge Management*, pages 426–434, Nov 2003.

[6] J. Cho and A. Ntoulas. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proc. of the 30th Annual SIGIR Conf. on Research and Development in Information Retrieval*, July 2007.

[7] R. Cole. Parallel merge sort. *SIAM J. on Computing*, 17, 1988.

[8] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proc. of the Int. Conf. on Management of Data*, 2006.

[9] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques*, 2005.

[10] M. Harris. Parallel prefix sum (scan) with CUDA, April 2007. http://developer.download.nvidia.com/compute/cuda/sdk/website /projects/scan/doc/scan.pdf.

[11] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proc. of Parallel Architectures and Compilation Techniques*, 2008.

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

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

[14] S. Heman, M. Zukowski, A. de Vries, and P. Boncz. MonetDBX100 at the 2006 TREC Terabyte Track. In *Proc. of the 15th Text REtrieval Conference (TREC)*, 2006.

[15] M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems*, 17(4):406–439, Oct. 1999.

[16] E. Lawler, J. Lenstra, A. Kan, and D. Shmoys. *Sequencing and scheduling: algorithms and complexity.* Elsevier, 1993.

[17] D. Lichterman. Course project for ECE498, Univ. of Illinois at Urbana-Champaign. http://courses.ece.uiuc.edu/ece498/al1 /Archive/Spring2007/HallOfFame.html.

[18] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. on Inf. Systems*, 14(4):349–379, 1996.

[19] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics, State of the Art Reports*, Aug 2005.

[20] R.Blumofe and C.Leiserson. Scheduling multithreaded computations by work stealing. *Proc. of the IEEE Symp. on Foundations of Computer Science*, 1994.

[21] K. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *1st Latin Amer. Web Congress*, 2003.

[22] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at trec-3. In *Proc. of the 3rd Text Retrieval Conference (TREC)*, Nov 1994.

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

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

[25] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. of the Int. Conf. on Data Engineering*, 2006.