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.