# Using Graphics Processors for High-Performance IR Query Processing

Shuai Ding, Jinru He, Hao Yan, and Torsten Suel
CIS Department, Polytechnic University
Brooklyn, NY, 11201, USA
sding@cis.poly.edu, jhe@cis.poly.edu, hyan@cis.poly.edu, suel@poly.edu

## ABSTRACT

Web search engines are facing formidable performance challenges as they need to process thousands of queries per second over billions of documents. To deal with this heavy workload, current engines use massively parallel architectures of thousands of machines that require large hardware investments.

We investigate new ways to build such high-performance IR systems based on 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 including databases and scientific computing [2, 3, 5], but we are not aware of previous attempts to use GPUs for large-scale web search. Our contribution here is to design a basic system architecture for GPU-based high-performance IR, and to describe how to perform highly efficient query processing within such an architecture. Preliminary experimental results based on a prototype implementation suggest that significant gains in query processing performance might be obtainable with such an approach.

**Categories and Subject Descriptors:** H.3 [INFORMATION STORAGE AND RETRIEVAL]
**General Terms:** Performance.
**Keywords:** Web search, query processing, GPU.

## 1. TECHNICAL PRELIMINARIES

For a good overview of IR query processing, see [7]. We assume that each document (e.g., each page covered by the engine) is identified by a unique *document ID* (docID). Our approach is based on an *inverted index* structure, used in essentially all current web search engines, which allows efficient retrieval of documents containing a particular set of words (or *terms*). An inverted index consists of many *inverted lists*, where each inverted list $I_w$ contains the docIDs of all documents in the collection that contain the word $w$, sorted by document ID or some other measure, plus possibly the number of occurrences in the document and their positions. Inverted indexes are usually stored in highly compressed form on disk, or sometimes in main memory if space is available.

Given 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 number of occurrences and their positions) is used to compute a score for the document, and the $k$ top-scoring documents are returned. These operations are usually pipelined so that the score of a document is computed immediately after the Boolean filter (usually intersection) is applied to it, which is itself done right after decompressing the relevant index entries, with no need to temporarily write the decompressed data to main memory. Thus, the main operations required are index decompression, intersection, and score computation. Query processing is responsible for a large fraction of the total hardware cost of a large state-of-the-art engine.

**Index Compression:** Compression of inverted indexes greatly reduces disk space use as well as disk and main memory accesses, resulting in faster query evaluation. There are many index compression methods [7]; the basic idea in most of them is to first compute the differences (gaps) between the sorted docIDs in an inverted list, and then apply a suitable integer compression scheme to the gaps. During decompression, the gaps are decoded and then summed up again in a prefix sum operation. In our prototype, we focus on two methods that we believe are particularly suitable for GPUs: *Rice coding*, and a recent method in [8] called *PForDelta*.

To compress a sequence of gaps with Rice coding, we first choose a $b$ 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 coding is considered somewhat slow in decompression speed, we consider here a new implementation proposed in [6] that is much faster than the standard one. The second compression method we consider is the PForDelta method proposed in [8], 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 values 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 [6].

**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 the computing power of these processors for other types of applications [2, 3, 5]. Modern GPUs offer multiple 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 basic 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 basic steps. This is especially challenging for operations such as decompression and intersection that tend to be more adaptive in nature. One major vendor of GPUs,

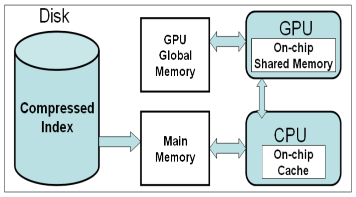

**Figure 1: Architecture of a GPU-Based System.**

NVIDIA, recently presented 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 and tested on an NVIDIA GeForce 8800 GTS graphics card. However, other cards supporting CUDA could also be used, and our general approach can be ported to other GPU programming environments.

## 2. A GPU-BASED IR QUERY PROCESSOR

We now first outline the basic structure of our GPU-based IR query processor, based on Figure 1. 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 via the CPU, but also has its own global memory (640 MB in our case) plus several other specialized caches, including shared memory that can be shared by many threads. Data transfer between CPU and GPU and thus between main memory and GPU is reasonably fast (at least 3 GB in our tests), while memory bandwidth between the GPU and its own global memory is in the tens of GB and thus much higher than typical system memory bandwidths. (But accesses need to be scheduled carefully to achieve this performance.)

A query is processed as follows: First, the CPU receives and preprocesses the query. The corresponding inverted lists are retrieved from disk if not already cached in main memory. The data is then transferred to the GPU Global Memory, which also maintains its own cache of index data in order to decrease the time for main memory-to-GPU data transfers. The GPU then decompresses and intersects the lists and returns the top results to the CPU. For best performance, the CPU may also process some of the queries itself such that both processors share the overall load. Thus, the CPU controls the overall process, while the GPU serves as an auxiliary compute device.

**Algorithms for GPU Query Processing:** We now give more details on our implementation; due to space constraints, we can only give a sketch of our techniques.

We first describe how to decompress an inverted list, for the case of Rice coding. As in [6], the unary and binary parts of the Rice code are kept separately. We can then decompress the code by running two separate prefix sums, one on the binary data and one on the unary data. The binary prefix sum operates on $b$-bit elements, while the unary one sums up the total number of '1' values in the unary data. Then each document ID can be retrieved by summing up its binary prefix sum and $2^b$ times the corresponding unary prefix sum. Multi-level tree-based algorithms for prefix sums on GPUs were discussed in [4] based on earlier SIMD algorithms; we adapted and fine-tuned these for unary and $b$-bit binary data.

The other challenge is the intersection of the different inverted lists. To do so, we again adopt older techniques from the parallel computing literature, for the related problem of merging lists of integers. In particular, we use multi-level algorithms that first select a small subset of splitter elements that are inserted into the other list, thus partitioning this other list into segments each of which only has to be compared to a small subset of elements in the first list; this process can then be applied recursively and in the other direction.

CPU-based algorithms typically pipeline the various operations in query processing. This is difficult in GPUs, but also less necessary due to the high global memory bandwidth. CPU-based algorithms also typically skip over some parts of an inverted list without decompressing it, if the list is much longer than the one it has to be intersected with. This is achieved by partitioning each list into blocks of some size, say 128 docIDs, and storing the first docID in each block in uncompressed form such that decompression can be resumed at any block boundary. This can also be implemented in a GPU, though slightly differently, by performing a merge-type operation between the uncompressed elements of a longer list and the elements of an already decompressed shorter list (similar to the intersection above); this determines the list of blocks in the longer list that need to be decompressed. Keeping one element in each block uncompressed also limits the prefix computation to within each block, resulting in a faster single-level computation than the multi-level approach in [4]. Finally, computation of rank scores is fairly straightforward.

## 3. EXPERIMENTS AND DISCUSSION

We now present preliminary experimental results that show the potential of the new architecture. Note that we have not yet implemented all of the features in the proposed query processor. All our runs use Rice coding, and we do not yet support blocked access to the inverted lists. For the experiments, we used the *TREC GOV2* data set of 25.6 million web pages, and selected 1000 random queries from the supplied query logs. On average, there were about 3.74 million postings in the inverted lists associated with a query. Our CPU implementation is based on the optimized code in [6].

| | Decompression | Intersection | Total |
|---|---|---|---|
| CPU Times | 25.89 | 5.73 | 31.62 |
| GPU Times | 11.39 | 1.95 | 13.35 |
| Speedup | 2.27 | 2.93 | 2.37 |

**Table 1: Times in ms per query for CPU and GPU.**

From Table 1, we see that we get a speedup of more than 2 even for our very preliminary implementation. There are various caveats, of course. Again, we have not implemented skipping and block-wise access. Also, results may be different for PForDelta compression, which runs very fast on modern CPUs. Finally, all results assume that the data is already in memory, or at least that disk is not the main bottleneck of the system. (Of course, if CPU is not the bottleneck, then a GPU cannot help either.) We are working on overcoming the current limitations and on completing a full prototype.

## 4. REFERENCES

[1] Nvidia CUDA Computer Unified Device Architecture – Programming Guide, June 2007. http://www.nvidia.com/object/cuda_develop.html.

[2] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. of the 26th SIGMOD*, 2006.

[3] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proc. of the 24th SIGMOD*, 2004.

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

[5] J. Owens, D. Luebke, and etc. A survey of general-purpose computation on graphics hardware. In *Eurographics*, 2005.

[6] 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.

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

[8] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. of the 22nd ICDE*, 2006.