Griffin: Uniting CPU and GPU in Information Retrieval Systems for Intra-Query Parallelism

Yang Liu*
WDC Research

Jianguo Wang
UC San Diego

Steven Swanson
UC San Diego

Abstract

Interactive information retrieval services, such as enterprise search and document search, must provide relevant results with consistent, low response times in the face of rapidly growing data sets and query loads. These growing demands have led researchers to consider a wide range of optimizations to reduce response latency, including query processing parallelization and acceleration with co-processors such as GPUs. However, previous work runs queries either on GPU or CPU, ignoring the fact that the best processor for a given query depends on the query's characteristics, which may change as the processing proceeds.

We present Griffin, an IR systems that dynamically combines GPU- and CPU-based algorithms to process individual queries according to their characteristics. Griffin uses state-of-the-art CPU-based query processing techniques and incorporates a novel approach to GPU-based query evaluation. Our GPU-based approach, as far as we know, achieves the best available GPU search performance by leveraging a new compression scheme and exploiting an advanced merge-based intersection algorithm. We evaluate Griffin with real world queries and datasets, and show that it improves query performance by 10x compared to a highly optimized CPU-only implementation, and 1.5x compared to our GPU-approach running alone. We also find that Griffin helps reduce the 95th-, 99th-, and 99.9th-percentile query response time by 10.4x, 16.1x, and 26.8x, respectively.

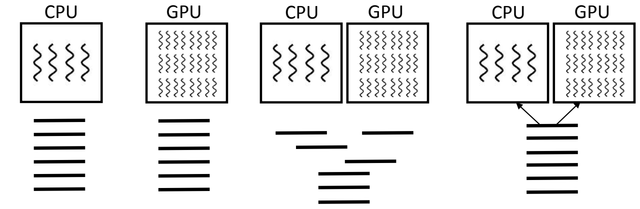
CCS Concepts • **Information systems → Search engine architectures and scalability**; • **Computing methodologies → Parallel algorithms**; • **Applied computing → Document searching**;

Keywords Information Retrieval, Search Engines, Query Processing, GPU, Parallel Algorithms

ACM Reference format:
Yang Liu*, Jianguo Wang, and Steven Swanson . 2018. Griffin: Uniting CPU and GPU in Information Retrieval Systems for Intra-Query Parallelism. In *Proceedings of PPoPP '18: Principles and Practice of Parallel Programming, Vienna, Austria, February 24–28, 2018 (PPoPP '18),* 11 pages.
https://doi.org/10.1145/3178487.3178512

1 Introduction

Information retrieval (IR) services serve as the key gateway to rapidly growing data sets, and must provide relevant results with consistently low latency [18]. To provide scalability, current IR systems resort to massive, coarse-grain parallelism by distributing queries across large compute clusters. To meet their latency goals, they rely on clever, highly-optimized algorithms that exploit intra-query parallelism on individual nodes.



(a) CPU-only (b) GPU-only (c) hybrid (d) Griffin

Figure 1. Intra-Query Parallelism Schemes.

Previous work[16, 18, 28, 29] explores intra-query parallelism by increasing CPU threads to process each query (Figure 1(a)). Existing studies [8, 12] also leverage the parallelism from GPUs (Figure 1(b)) and can obtain impressive speedup over CPUs [8]. Because queries have different characteristics, some queries may run better on GPU, while others run better on CPU. So a heterogeneous system can achieve better overall performance by running individual queries on proper processors [12] (Figure 1(c)).

However, the characteristics of a query can also change as the query executes. While the early stages of a query's execution may run well on the GPU, the later stages are often a better fit for the CPU, since as the query goes, the amount of processing needed will decrease. Thus, running an entire query solely on CPU or GPU statically (Figure 1(a), (b), and (c)) may not achieve the best performance. This suggests that a dynamic fine-grained approach will lead to better performance, by scheduling operations in different stages to suitable processors when processing a query (Figure 1(d)).

*Author conducted this work while at UC San Diego.

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178512

In this paper, we present *Griffin*, an IR system that combines two innovations to provide improved performance. First, Griffin uses both the CPU and GPU to process queries and migrates query execution from the GPU to the CPU as the characteristics of queries change. Griffin decides when and where to execute query operation without *a priori* knowledge of the query's characteristics. To make a proper decision, Griffin considers overheads due to data transfer between CPU and GPU, GPU memory management, as well as the system load. Griffin addresses these challenges with a dynamic intra-query scheduling algorithm that breaks a query into sub-operations and schedules them to the GPU or to the CPU based on their runtime characteristics. Griffin uses this scheduling algorithm to divide work between a state-of-the-art CPU-based search implementation and a new GPU-based search kernel called Griffin-GPU.

Griffin-GPU is the second key innovation in Griffin. Griffin-GPU combines two components. The first is the parallel Elias-Fano decompression [30] algorithm that provides fast decompression and a high compression ratio. The second is a load-balancing merge-based [15] parallel list intersection.

Our experiments on the real world query dataset [1] show that, Griffin speeds up the query processing by 10x and 1.5x compared to a highly optimized CPU-based query processing system and Griffin-GPU running alone, respectively. Griffin also reduces tail latency: It reduces the 95th-, 99th-, and 99.9th-percentile latencies by 10.4x, 16.1x and 26.8x, respectively, compared to the CPU-only implementation.

The remainder of this paper is organized as follows. Section 2 introduces the background of query processing, and discusses the characteristics of CPU and GPU. Section 3 describes the design and implementation of Griffin. We then evaluate the Griffin prototype in section 4. Section 5 discusses the related work. Finally, we conclude in Section 6.

2 Background

The increase in the amount data available via search has led to highly optimized search algorithms and systems. They aim to maximize performance (in terms of latency and bandwidth) and minimize memory and/or storage requirements. As a result, IR systems store indices in specialized compressed formats that minimize data storage while still allowing for fast search.

IR systems also tailor the compressed data structures and algorithms to match different hardware characteristics. As a result, the best techniques for a CPU-based IR system will differ from those for a GPU-based one. Below, we describe current approaches of query processing in search on CPU and GPU.

2.1 Query Processing

Query processing is at the heart of IR systems and is the focus of this paper. The most important query processing data structure is the *inverted index*, which consists of many *inverted lists*. Each inverted list corresponds to a search term, and holds the document IDs (docIDs) of the documents containing that term, usually in sorted order. IR systems usually store inverted index in a compressed form and decompress it as needed to minimize the memory and storage cost.

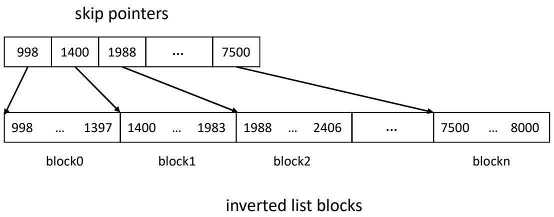


Figure 2. An Example of An Inverted List with Skip Pointers. The skip pointers store the offset and the first value of each inverted list block, and can support binary search to fast locate the required blocks.

To process a query, the system first loads the inverted lists of the query terms into memory, and decompresses them. It then computes the intersection of the lists, yielding the set of the common docIDs that contains all the search terms[1].

To compute the full intersection, the system performs a series of pair-wise intersections. It usually starts with the two shortest inverted lists (i.e., the two rarest search terms) to avoid unnecessary computation. This yields an intermediate list of docIDs, which the system then intersects with the next longer inverted list. The process repeats until all the lists have been incorporated or the list of matching docIDs is empty.

The basic algorithm of intersecting two sorted inverted lists is similar in spirit to merge sort: The system scans the sorted lists and records the common docIDs. Modern IR systems use sophisticated data structures like skip lists and skip pointers (Figure 2), to skip large portions of the lists during the scan.

Next, the system calculates a relevance score for each document using document metadata (e.g., term frequency or document popularity), sorts the documents according to this score, and returns the top results.

Below we describe the three operations in more detail.

2.1.1 Index Compression and Decompression

Since the inverted index can be very large, compression is necessary to save the cost of storage and data transfer. Aggressively optimized IR systems may even keep the entire inverted index in memory spreaded across machines. Thus, a higher compression ratio is as important as the decompression speed. Compression speed is less critical since compression only occurs once when generating a new index (usually offline).

There are many proposed compression schemes for inverted lists [39]. Many compression techniques in modern IR

[1] The discussion in this paper focuses on conjunctive queries where only documents that contain all terms may be returned.

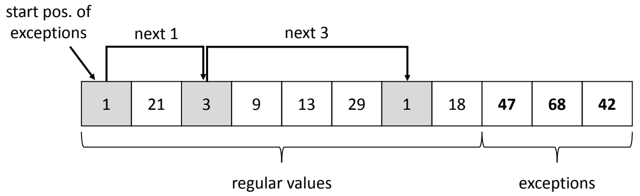


Figure 3. An Example of PforDelta Encoding. Given a sequence of docIDs $\ell(t) = (100, 121, 163, 172, 185, 214, 282, 300, 347)$, the corresponding sequence of d-gaps is $\ell(d) = (21, 42, 9, 13, 29, 68, 18, 47)$. With $b = 5$ bits for regular values, the exceptions are (42, 68, 47). The pointer (the gray positions) to the next exception element also takes 5 bits. The value of exceptions are stored at the end of the sequence.

systems start by computing the deltas (a.k.a *d-gaps*) between two consecutive docIDs. PforDelta [40] is a popular d-gaps-based compression scheme that provides good tradeoff between decompression speed and space overhead [32, 37, 38].

Given a sequence of d-gaps $\ell(d)$ out of the original ascending docIDs, PforDelta compresses these d-gaps in various number of bits. In particular, it encodes a majority of elements (e.g., 90%), called *regular values*, in b bits where b is determined by the smallest number of bits to represent the largest regular value. For the rest elements that cannot be represented by b bits (called *exceptions*), it uses b bits to store the position of the next exception (in a linked list manner) while leaving the actual values of exceptions uncompressed at the end of the whole compressed sequence. Figure 3 shows an example of PforDelta.

To avoid decompressing the entire list, the algorithms usually store d-gaps in blocks of fixed number (e.g., 128) of elements [31]. Each block contains information about the range of docIDs in the block, so the algorithm can quickly determine if a block contains any docIDs of interest.

2.1.2 List Intersection

List intersection is the key operation of query processing. It scans and intersects over several inverted lists of the corresponding terms to find the common docIDs.

SvS is a popular intersection algorithm [11, 34]. It orders the lists from the shortest to the longest, and computes partial intersections starting with the two shortest lists. For example, if we search “PPoPP Austria 2018”, the system may subtract and divide the terms into three inverted lists for terms “PPoPP”, “Austria”, and “2018”, respectively:

$$\ell(PPoPP) = (11, 15, 17, 38, 60),$$
$$\ell(Austria) = (3, 5, 8, 11, 13, 15, 17, 38, 46, 60, 65),$$
$$\ell(2018) = (2, 4, 6, 11, 13, 14, 15, 19, 25, 33, 38, 60, 70).$$

And the result of the intersection is the common docIDs in the three inverted lists:

$$\ell(PPoPP) \cap \ell(Austria) \cap \ell(2018) = (11, 15, 38, 60).$$

Starting from the shorter lists improves the overall performance, as the run-time of each merge step depends strongly on the length of the shorter list of the two.

2.1.3 Rank Scoring

The goal of IR Systems is to return the most relevant results to end users. This typically involves two steps. (1) Similarity computing: compute the similarity of the query q and each candidate document d; (2) Ranking: determine the top k results that have the highest scores. In this paper, we follow a popular ranking model BM25 [26, 27, 35] for similarity computation.

Typically, each entry in the inverted list contains a document frequency (in addition to document ID and positional information). When a qualified result ID is returned, its score is computed accordingly.

2.2 Query Processing on CPU

Most IR systems execute decompression, list intersection, and ranking operations on CPU. CPU is good at dealing with complex logic, and its advanced prefetch and branch handling can provide high performance and efficiency. As a result, CPU is able to run fast sequential merge, especially when the data accesses exhibit ample spatial locality. When the two lists involved in the intersection have similar lengths, the algorithm must access most of the items in both lists, leading to that locality. Therefore, CPU performance on the merge is high. Alternately, if the length difference between the two lists is large, CPU can perform binary search with the help of skip pointers to skip large portion of unnecessary computation including decompression and comparison. In this case, the CPU clock speed and aggressive branch handling will still perform well.

On the other hand, CPU cannot exploit the large amount of fine-grain parallelism that exists in query algorithms [17], due to the limited number of cores. The CPU accounts for over 70% of the response time [16] in search workloads and an even larger fraction when the query has many terms, since there are more decompression, intersection, and ranking computations to perform. As result, the fine-grain parallelism that CPUs cannot exploit is a significant missed opportunity.

2.3 Query Processing on GPU

GPUs offer a way to exploit the parallelism that CPUs cannot. Compared to CPUs, GPUs are able to provide massive parallelism with hundreds or thousands of simpler cores. GPU hardware multithreading and fast context switching can hide both arithmetic and memory latency, as well as avoid dependency stalls by overlapping thread execution. What is more, its SIMD execution model can amortize the overhead of instruction fetch and decode, and harness data parallelism [6]. In addition, GPU have much higher inner bandwidth (e.g., as high as 208GB/s in NVIDIA K20) than CPU, but with very limited memory (e.g., 5GB in NVIDIA K20).

On the other hand, GPUs incur substantial startup overheads related to data transfer and memory allocation. However, these costs occur just once, so running larger, more complex query operations can amortize them.

Decompression on GPU A good parallel decompression algorithm should be efficient on GPU without sacrificing high compression ratio or fast decompression speed. The CPU decompression method PforDelta is a poor match for GPU implementation, because it maintains a linked list to store the exception pointers that it must process sequentially. This leads to slow global memory accesses and thread divergence. Consequently, directly porting it to GPU results in poor decompression performance, while a higher decompression speed can only be achieved at the cost of lower compression ratio [8].

List Intersection on GPU GPU list intersection algorithms often rely on parallel binary search. When the length difference between the two lists involved in the intersection is large, parallel binary search reduces the search space quickly. In addition, since binary search can skip many blocks in the longer list, it also reduces the number of blocks that the GPU must decompress.

However, GPU binary search is not efficient. The frequent branch divergence results in idle threads and reduced performance. Even worse, when the length difference between the two intersecting lists is large, binary search is more likely to reach its worst case complexity $\log(N)$. This will lead to more frequent divergence, as the vast majority of the items in the longer list will be missing from the shorter list. In addition, as each thread accesses a different area of the memory, there is little opportunity to coalesce accesses, leading to lower memory bandwidth.

When the lengths of the two lists are close, the benefit of reducing search space in the binary search decreases. As we will see in Section 3, empirically, the benefits start to fade when the list size ratio falls below 128x. And it is quite common in reality: in the real-world dataset we use, more than 64% of *actual* intersections contain the lists with the length ratio lower than 128.

Parallel merge-based search might be an efficient alternative to binary search in these cases. A merge-based algorithm allows for cache line reuse among neighboring threads and reduce global memory access, since such algorithm can load data into the faster thread-shared memory. We describe our efficient merge-based algorithm in the next section.

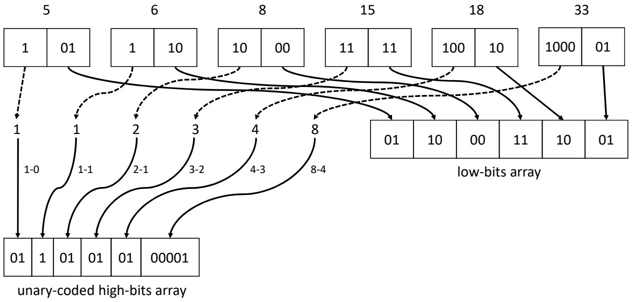


Figure 4. An EF Encoding Example. For the original integer sequence (5,6,8,15,18,33), upper bound $U = 36$, thus $b = \lfloor \log \frac{36}{6} \rfloor = 2$. The low-bits array on the right stores the low b bits of the original sequence, while the unary-coded d-gaps array on the bottom left encodes the high-bits array. Each encoded element in the d-gaps array ends with "1", and the number of "0s" inside an element represents the value of d-gap. The accumulated number of "0s" gives the actual high-bits value.

3 Design and Implementation of Griffin

This section describes the design and implementation of the Griffin prototype. Griffin consists of three main components: Griffin-GPU, the CPU query processing component, and the scheduler. Griffin-GPU implements the advanced parallel algorithms on the GPU. The CPU query processing component implements state-of-the-art CPU query algorithms [11, 26, 40]. The scheduler decides where to run the current query operation.

3.1 Griffin-GPU

Griffin-GPU executes query operations on GPU and relies on two key algorithms: a novel parallel decompression scheme called *Para-EF encoding* to decompress inverted lists, and a parallel *merge-based intersection* algorithm to efficiently intersect inverted lists on the GPU.

3.1.1 Parallel List Decompression

A suitable encoding scheme for Griffin-GPU has to satisfy three requirements: (1) high decompression speed; (2) high compression ratio; (3) high parallelism. The PforDelta scheme that is popular on CPUs is a not good fit for GPU implementation due to its sequential accesses to exceptions, which leads to slow global memory accesses and thread divergence. Directly porting it to GPU results in poor decompression performance, while a higher decompression speed can only be achieved at the cost of lower compression ratio [8].

Instead, we adopt and parallelize an encoding scheme called *EF (Elias-Fano) encoding* [13, 30], which is proved to have both higher decompression speed and compression ratio than PforDelta [30]. EF encoding is also a good candidate for GPU since there is few dependency between the operations of element decompression.

To compress a sequence of integers, EF encoding divides each integer into high bits and low bits, and encodes them into the *low-bits array* and the *high-bits array*. For the list with n integer elements and U as the maximum possible value, the low-bits array stores the (fixed) $b = \lfloor \log \frac{U}{n} \rfloor$ bits of each element contiguously. The high-bits array then stores the remaining upper bits (with variable lengths) of each element as a sequence of *unary-coded d-gaps* of these elements. To decompress these integers, we just need to recover the high bits from the unary-coded d-gaps array, find its corresponding low-bits, and concatenate them. Figure 4 illustrates the basic encoding and decoding of EF scheme in detail.

Griffin-GPU is the first GPU IR system to explore and utilize EF encoding, and we provide the first parallel EF decompression implementation *Para-EF decompression*, as described in Algorithm 1.

The algorithm first computes population count (popcount) for each element in the high-bits array *hb_array* (line 2) to determine the number of original (decompressed) elements each 32-bit word in the *hb_array* encodes. It then calculates prefix sum from the popcount and stores it in the temporary *ps_array*(line 3).

Algorithm 1: Parallel EF Decompression.

```
Input  : EF-compressed high-bits array $hb\_array$,
         low-bits array $lb\_array$, and the number of
         low bits $b$
Output : Decompressed array $decmp\_array$
1  for each thread $i$ do
2      $ps\_array[i] \leftarrow popcount(hb\_array[i])$;
3      $ps\_array[i] \leftarrow prefix\_sum(ps\_array[i])$;
4      $count = ps\_array[i] - ps\_array[i-1]$;
5      $offset = 0$;
6      while $offset < count$ do
7          $index\_array[offset + ps\_array[i-1]] \leftarrow i$;
8          $offset \leftarrow offset + 1$;
9      Recover $high\_bits_i$ from $hb\_array[index\_array[i]]$;
10     $decmp\_array[i] \leftarrow (high\_bits_i << b) \mid lb\_array[i]$;
```

Our algorithm divides the actual decompression into two phases: scheduling and decompressing. It first uses the prefix sum (synchronization point) result to schedule and assign decompression tasks to threads (line 4-8). In this way, each thread will be in charge of decompressing an individual value from the corresponding word in *hb_array*, and deliver it to the final decompressed array *decmp_array*. For example, if $ps\_array[0] = 13$, which means *word_0* contains 13 values, then *thread_0* to *thread_12* will decompress *element_0* to *element_12* from *word_0* of *hb_array*. If $ps\_array[1] = 20$, then *thread_13* to *thread_19* will decompress *element_13* to *element_19* from *word_1* of *hb_array*. And so on so forth. After the task distribution, each thread continues to recover the high_bits element and concatenate it to its corresponding low_bits to get the final decompressed element (line 9-10).

We implemented the fixed-length partitioned EF algorithm in CUDA [2], which provides a popcount instruction __popc [3]. We also use the parallel prefix sum to reduce the dependency in the original serial EF decompression. We store a look-up table in the shared memory of the GPU to further improve the performance of recovering the high-bits array. We also store the temporary arrays in shared memory.

3.1.2 Parallel List Intersection

Griffin-GPU's list intersection algorithm dynamically divides list intersection operations into two classes, depending on the relative sizes of the two lists. The cross-over point between the two techniques is a configurable parameter of the algorithm, and the default value is determined empirically in the next section.

When the difference between list lengths is large, Griffin-GPU uses parallel binary search with the skip pointers. Griffin-GPU first does binary search over the skip pointers instead of the long list to identify blocks that may contain that elements in the short list. It then only transfers, decompresses, and processes those blocks.

When the difference between list lengths is small, Griffin-GPU adapts a parallel merge-based intersection algorithm based on GPU MergePath [15], which derives from [24]. The goal of the algorithm is to reuse cache lines between neighboring threads, and reduce global memory access by accessing data in the shared memory.

This algorithm divides list intersection into two stages: *Partitioning* and *Merging* (Figure 5). In Partitioning stage, we divide the list into partitions that each potentially contains common elements from both lists. GPU MergePath sizes the partitions so a pair of partitions will fit in the GPU's shared memory. Then in Merging stage, we merge the two sub-lists inside each non-overlapping partition to get the final intersected results. The merging threads can run concurrently without synchronization.

One of the biggest challenges in efficient parallel merge algorithms is to find even partitions of the lists to achieve load balancing among GPU cores and eliminate the need for synchronization, which is not well addressed in previous studies [8, 12, 36]. Figure 5 shows an example of even partitions and their boundaries.

Instead of blindly doing binary search or deciding the partitions statically [8, 12], we first consider the process of merging list A and B from a different perspective. We can visualize this process as a *merge path* from the top-left corner to the bottom-right corner of an $|A| \times |B|$ grid (Figure 6), allowing only right and downwards directions. Because both list A and B are sorted, we always have:

$$\begin{aligned} &For\ all\ j' > j, \\ &if\ A[i] < B[j],\ then\ A[i] < B[j']. \end{aligned}$$

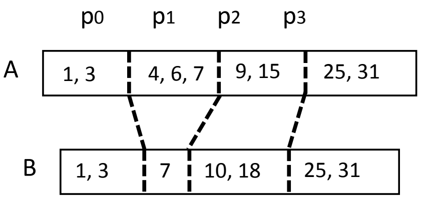


Figure 5. An Even Partition Example. By dividing elements from list A and B into 4 partitions (P_0,P_1,P_2,P_3), each partition contains 4 elements from both lists evenly.

Thus, merging the two lists is transformed to the process of finding the merge path through this matrix. We can construct such a path executing the following two operations:

- Advance rightward by one step, if $A[i] \leq B[i]$;
- Advance downward by one step otherwise.

The merge path is essentially the history of decisions made during the merge, and exactly one such path exists. We can use the existence of this path to locate the p partitioning points for A and B. We can find these points by drawing p cross diagonals from the top-right to the bottom-left of the grid. Because the cross diagonals and the path are in the opposite directions, they must meet at some intersections, and they are the partitioning points.

These points and diagonals have interesting properties:

- Given a partitioning point P_{ij}, we must have $A[i] = B[j]$ or $A[i]$ is the lower bound into B for the current partition.
- For a point on a diagonal P_{ij}, $i + j = |diagonal|$, where $|diagonal|$ is the distance from the top-left origin to the position where the diagonal intersects with the axis of list A.

We can use the above two properties to do binary search only along each diagonal (in parallel) to find the point where it crosses the merge path, and then use these points to partition A and B. Figure 6 illustrates the operation of MergePath.

The p partitioning points divide the all the elements from A and B into p partitions evenly, thus the process of finding these points ensures that the merge stage is perfectly load balanced. After discovering these p partitioning points, we can eventually do the serial merge inside each partition. More detail about the algorithm and its CUDA implementation can be found [15] and [4].

3.1.3 Ranking Selection

The final stage of query processing is identifying the top-ranked query results to return to the user. We evaluated three ranking functions in Griffin-GPU: GPU bucketSelect [7], GPU radixSort, and CPU partial_sort (provided by the C++ STL).

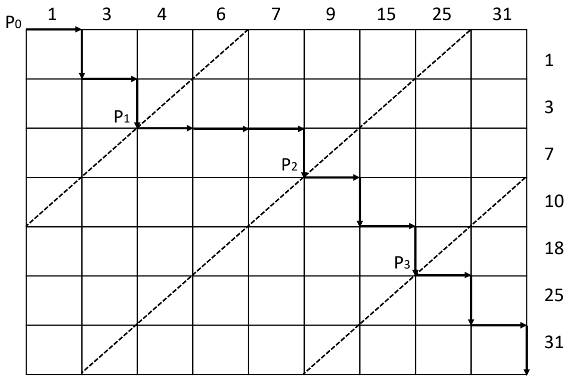


Figure 6. A Merge Path Example. To intersect list A = (1,3,4,6,7,9,15,25,31) and list B = (1,3,7,10,18,25,31) with 4 partitions ($p = 4$, so each partition has $(|A| + |B|)/p = 4$ elements), we draw 4 diagonals in the grid, and they intersect with the merge path shown with arrows at 4 points (P_0, P_1, P_2, P_3). *partition_*0 contains (1,3) from A and (1,3) from B, *partition_*1 contains (4,6,7) from A and (7) from B, *partition_*2 contains (9,15) from A and (10,18) from B, and *partition_*3 contains (25,31) from A and (25,31) from B. We can merge them concurrently to get the intersection: (1,3,7,25,31).

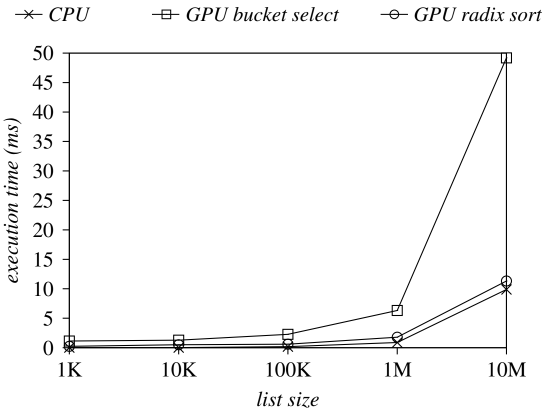


Figure 7. Ranking Performance Comparison.

GPU bucketSelect is a fast parallel K-selection algorithm. We use it to locate the Kth-max value first, and then we select all K-max values. The GPU radixSort is a brute-force solution that sorts all values in the list, and we pick the first K values. The CPU partial_sort returns only the K-max values.

We sample the list of full results for 100 queries and ran these three ranking algorithms on them. We find that the CPU implementation provides the best performance (Figure 7). We suspect the poor performance of the GPU algorithms is due to the small number of results (queries rarely result in more than several thousands matches). These small input sizes cannot saturate the GPU or amortize the overhead in GPU initialization and memory allocation.

3.2 Hybrid Query Processing in Griffin

The performance of list intersection in Griffin-GPU relative to the CPU implementation depends strongly on the relative sizes of the lists at hand: Griffin-GPU performs best when both lists are relatively long and the difference in their

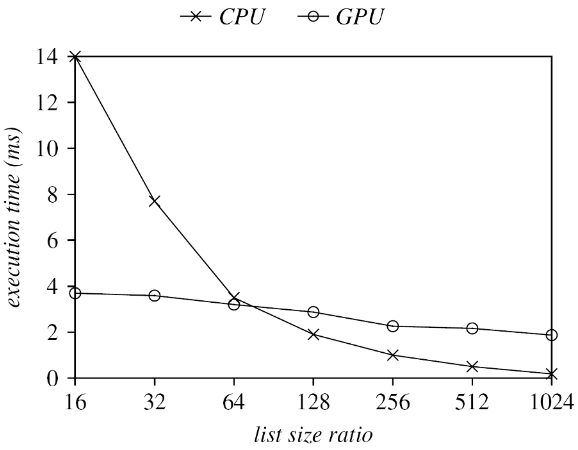


Figure 8. The Observation of the GPU/CPU Cross Over Point for Better Performance.

lengths is relatively small, while the CPU performance better when the difference is large.

As query processing proceeds, the length difference between the two lists involved will increase for two reasons. First, one of the two lists constitutes the intermediate results for the query, and this list shrinks monotonically during execution. Second, the query algorithm starts with the two shortest lists, so the algorithm intersects the list of intermediate results with longer lists.

To understand how list length ratio affects the performance of Griffin-GPU and CPU implementation, we need to compare their performance with different list length ratios. Instead of iterating all the possible ratios (which would be too many), we devide the list into 7 ratio groups: [1,16), [16,32), [32,64), [64,128), [128,256), [256,512) and [512,1024).

To have meaningful results, the candidate lists from these groups have to satisfy three requirements: 1) long enough to have accurate runtime measurements. 2) their runtime should be comparable (it is meaningless to compare the runtime of a short list pair with that of a long list pair, even they have the same list ratio). 3) have enough samples (cannot be too short or too long).

As a result, from each ratio group we randomly select 100 intersection list pairs, with the longer lists limited in the length range of [1M,2M]. We measure the run time of Griffin-GPU and CPU implementation, as shown in Figure 8.

Griffin-GPU outperforms the CPU version when the list lengths are with a factor of 128. In these cases, almost all of the data blocks in the intersected lists need to be decompressed, leaving little chance for binary search to avoid decompression. Thus, Griffin-GPU can utilize the parallel EF decompression as well the parallel merge-based intersection algorithm to efficiently merge the two lists. In addition, the GPU-related overhead of kernel invocation, memory management, and data transfer from CPU to GPU can be well amortized.

For larger ratios, the performance of the CPU implementation surpasses the GPU. As the ratio increases, the latency of Griffin-GPU continues to drop, but the latency of the CPU implementation decreases faster. This is because when the length difference between the two lists is large, performing binary search allows the algorithm to avoid decompressing large portions of the list. At this time, the modern CPUs with speculative execution and branch prediction can address the branch divergence effectively while avoiding the additional overhead of moving data to GPU.

Besides empirical measurements, the value of 128 as a cross over point between GPU and CPU is also supported by our formal analysis, given the premise that we compress 128 elements inside each data block with EF encoding, as we show below.

Let R and S be two lists with $|R| \le |S|$. Let $\lambda = \frac{|S|}{|R|}$, then we show that if $\lambda > 128$, it is more likely to skip unnecessary data blocks:

$$\lambda > 128 \Longleftrightarrow \frac{|S|}{|R|} > 128 \Longleftrightarrow |R| < \frac{|S|}{128}$$

In other words, when $\lambda > 128$, the number of elements in the short list R is smaller than the number of blocks in the long list S. Thus, there exists at least one block in S that is irrelevant and can be skipped, as demonstrated in a simplified example in Figure 9.

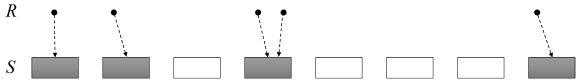


Figure 9. An Example to Explain the Ratio. List R contains 5 elements while list S contains 8 blocks, and two elements in R are mapped to the same block in S. As a result, there are 4 blocks (in white) in S that can be skipped.

Similarly, we also show if $\lambda \le 128$, then all the blocks in S are relevant. This is because $\lambda \le 128 \Longleftrightarrow |R| \ge \frac{|S|}{128}$, indicating the elements in R are likely to be mapped to all the elements in S. Note the value of 128 is closely related to the fact that we compress the list in 128-element blocks. So we could generalize our analysis and choice of the value to different block sizes.

To exploit the crossover point between the performance of the GPU and CPU implementations, Griffin uses both GPU and CPU to cooperatively process queries. When a query arrives in the system, the scheduler first decides if the current query is suitable for running on Griffin-GPU or on CPU, depending on the length ratio for the two shortest inverted lists. If the ratio is less than 128, Griffin begins execution on the GPU.

After each intersection, the scheduler will check if the length ratio of the two lists in the next round is less than 128. If it is, processing continues on the GPU. Otherwise, Griffin transfers the intermediate results to CPU and executes the rest of the query there. We implement a light-weight scheduler to explore intra-query parallelism, but it could be extended to support other features like load balancing.

4 Evaluation

In this section, we experimentally evaluate Griffin to answer the following questions:

- What is the performance of Griffin-GPU?
- How effective Griffin is as a hybrid GPU/CPU IR system to process queries cooperatively?
- How much can Griffin improve the tail latencies?

4.1 Methodology

In our evaluation, we run real-world queries over inverted lists generated from real web data, and we assume the whole dataset has been loaded in the host main memory. We conduct the experiments on a server with a 4-core Intel Xeon E52609V2 CPU at 2.5 GHz and with 64 GB DDR3-1600 DRAM. The server installs an NVIDIA Tesla K20 GPU with 5 GB GDDR5 memory, which connects to the server through 16 lane PCIe 2.0 with 8 GB/s bandwidth. We run Ubuntu Linux kernel 3.16.3 with CUDA Toolkit 7.0.

It would be interesting to compare Griffin with some existing GPU IR system implementations such as [8, 12]. But under the present circumstances, it is hard, if not impossible to do direct comparison, for two reasons: First, the implementations of those systems are not publicly available; Second, the results reported in previous studies are from and optimized for different hardware platforms and runtime environments, on different benchmarks with much smaller data sizes. Instead, we compare Griffin against both a highly-optimized CPU-only implementation and GPU-only implementation with state-of-the-art algorithms and report the speedups.

4.2 Benchmark

The benchmark used in our evaluation includes two parts: the queries and the web data [32]. The queries we run are from the query logs collected from the TREC [1] 2005 and 2006 (efficiency track). The web data clueweb12 [5] is a collection of 41 million Web documents (around 300 GB) crawled in 2012. It is a standard benchmark and widely used in the information retrieval community. We parse the documents and build the inverted lists, each entry of which contains a document ID and the corresponding document frequency [22]. In our experiments we randomly select 10,000 queries over about 100GB of these web documents with our limited disk space.

To have a better understanding of the benchmark, we first analyze the inverted lists and the queries we use. Figure 10 gives the size distribution of the inverted lists involved in our experiments. Most lists are of the size between 1 K and 1 M elements. Figure 11 shows the distribution of the number of terms for the queries. About 27% queries contain 2 terms, 33% contain 3, and 24% contain 4. This distribution shows that multiple rounds of list intersections are common in the benchmark, indicating that the query charateristics change ofen.

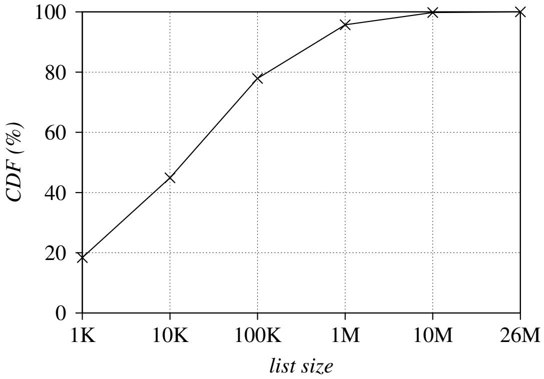


Figure 10. Inverted List Size Distribution.

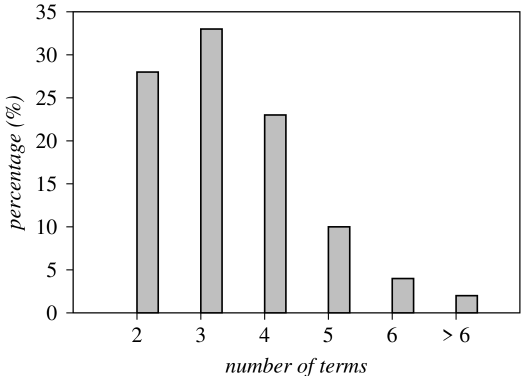


Figure 11. Number of Terms Distribution.

4.3 Performance of Griffin-GPU

We run several micro benchmarks to test the performance of Griffin-GPU in decompression and list intersection.

4.3.1 Decompression

Griffin-GPU implements Para-EF decompression algorithm described in Section 3, and we compress the inverted lists into 128-element blocks. We compare the average compression ratio of the EF scheme to that of state-of-the-art PforDelta over all inverted lists in our tests. EF scheme can achieve an average compression ratio of 4.6, which is 1.4x better than PforDelta (Table 1).

Table 1. Compression Ratio Comparison.

Scheme	PforDelta	EF
Compression Ratio	3.3	4.6

To demonstrate the decompression speed of Griffin-GPU, we randomly select about 7 K lists with lengths ranging from 1 K to 10 M and group them by sizes. We run both the Para-EF decompression of Griffin-GPU and CPU PforDelta decompression on these lists. Figure 12 depicts the average decompression time for each of the groups, and shows the speedup of Griffin-GPU over the CPU PforDelta decompression. When the lists are very short (e.g., of ~ 1*K* or ~ 10*K* elements), the speedup is relatively low (< 2). With the increase in the list size, however, the speedup increases from ~11x to ~29.6x. There are two reasons for this result. First, a longer list with more elements is more likely to saturate the GPU with higher degree of parallelism than a short list. Second, decompressing a longer list requires more computation, and this amortizes the overhead of data transfer and GPU memory allocation.

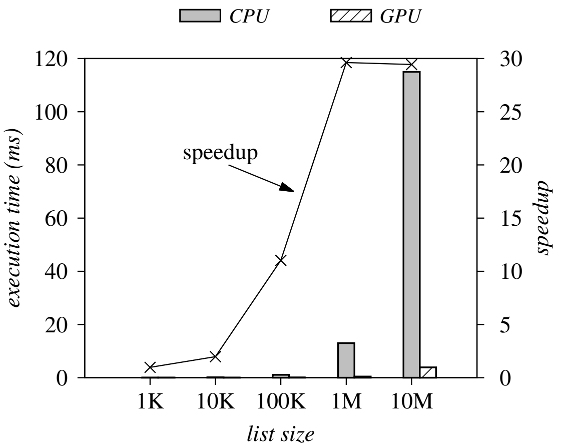


Figure 12. Decompression Speed Comparison.

4.3.2 List Intersection

To compare the performance of Griffin-GPU's parallel merge-based intersection to CPU merge, CPU binary, and parallel binary search, we run the experiments of list intersection on selected pairs of lists from our dataset. The list pairs have comparable list lengths (the two lists either have similar lengths or the length of the longer list is less than 16x longer than the shorter list), and their lengths ranges from 1 *K* to 10 *M*.

Figure 13 shows the performance of list intersection with different methods. With the relatively longer lists, both CPU merge and Griffin-GPU merge outperform their binary search counterparts. This is because when two lists have comparable lengths, merge-based methods can make better use of the cache and local memory of the processor. We also notice Griffin-GPU's merge can achieve up to 87.35x speedup over the CPU merge.

CPU binary search is slowest in these cases, while GPU binary search can achieve a speedup up to 102x over its CPU counterpart due to the parallelism. However, Griffin-GPU merge can still have up to 2.29x speedup over the very fast GPU binary search.

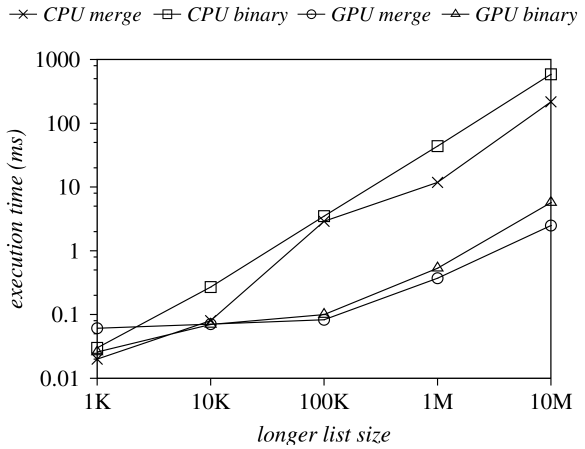


Figure 13. List Intersection Comparison.

4.4 Overall Performance

To verify Griffin's effectiveness as a hybrid GPU/CPU IR system that processes queries cooperatively, we test the end-to-end query processing latency with Griffin against a highly-optimized pure CPU search implementation and Griffin's own pure GPU implementation (Griffin-GPU). We first divide the queries into different groups based on the number of terms present in the queries, and then run the three different configurations on each group to get the average latency. From Figure 14 we can see that, Griffin can consistently outperform the pure CPU search and the Griffin-GPU, with an average speedup of ~ 10x and ~ 1.5x, separately.

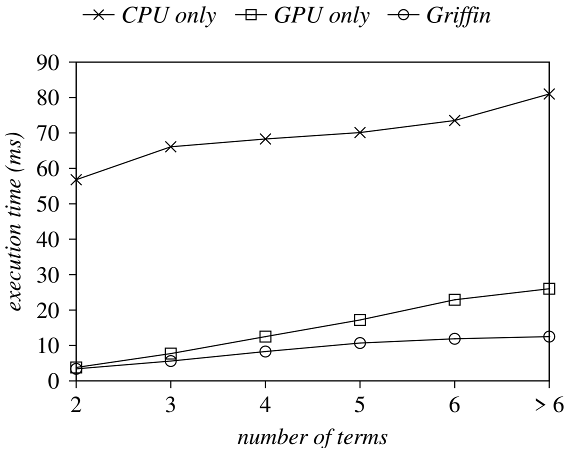


Figure 14. End-to-End Query Latency Comparison.

4.5 Case Study: Tail Latency Reduction

Reducing tail latency is very important for interactive services such as search, since very long tail latency will significantly affect the quality of service and the user experience negatively. To see if Griffin can effectively reduce tail latency of the queries, we compare the tail latency of the CPU search versus Griffin in Figure 15. As we can see, Griffin can achieve a speedup of 6.6x, 8.3x, 10.4x, 16.1x, and 26.8x, for 80th-, 90th-, 95th-, 99th-, and 99.9th-percentile response time over the CPU search.

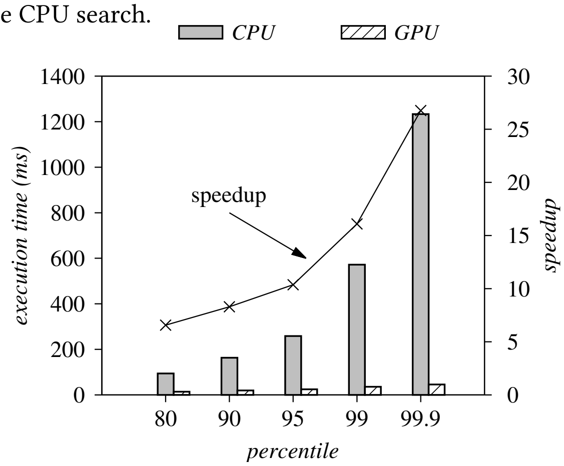


Figure 15. Tail Latency Reduction with Griffin.

5 Related Work

Parallelizing query processing with GPU There have been efforts in parallelizing query processing with GPU. Ding *et al.* [12] are among the first to explore building IR systems with GPU, and propose to schedule individual queries to either GPU or CPU ("hybrid" as they call). However, their decompression algorithm is hard to achieve both high compression ratio and fast decompression speed together, while their *Parallel Merge Find* intersection does not consider load balancing partitioning for GPU (key factor for high GPU performance) at all.

Both Griffin and [12] can schedule query processing to both CPU and GPU. Griffin improves on [12] in three ways: 1) For each single query, [12] can only send the *whole* query to either CPU or GPU; while Griffin makes finer-granularity intra-query decisions about scheduling. 2) Griffin's design adapts the scheduling to account for changing query characteristics while optimizing for performance and load balance. [12] has to deal with the changing characteristics with some machine learning models, but it is not clear if using simple models and training on a small set are accurate enough to predict query characteristics in reality (their results are from simulation). 3) Griffin's scheduler can be easily extended to support more complex scheduling [21].

Wu *et al.* also propose a CPU-GPU cooperative method for list intersection [36] on individual queries. But again, they fail in considering intra-query parallelism, neglecting the changing characteristics of the queries during runtime.

Ao *et al.* [8] propose a new linear-regression-based compression scheme and list intersection method. But they assume the linear properties of the datasets, and may not perform well on some datasets [8]. In addition, their design caches all inverted lists in the very limited GPU memory (in their case 1.5GB on NVIDIA GTX 480). While saving much time of data transfer between CPU and GPU as well as the overhead of GPU memory allocation, such design is not practical or scalable even only caching the most frequently accessed data, given the rapid growing volume of data today.

Parallelization in interactive services to reduce response latency. Adaptive job scheduling in multiprogrammed environment has been studied [9, 10, 14, 23]. The scheduler assumes no *a priori*, and can adjust the degree of parallelism as the job executes based on the job characteristics. Similarly, Griffin can schedule part of a query to either CPU or GPU based on the changing query characteristics.

Degree of Parallelism Executive (DoPE) proposed by Raman *et al.* [25] provides an API to choose different parallelism options. The runtime will decide the degree of parallelism dynamically. To reduce the average response time of queries, Joen *et al.* [18] propose adaptive parallelism, which chooses different degrees of parallelism for requests in advance, based on the system load and request requirements. Griffin could adopt these techniques for flexible scheduling.

To achieve load balance when applying multithreading in query processing, [18, 28, 29] explore intra-query parallel algorithms. While in Griffin, the merge-based intersection algorithm in Griffin-GPU can automatically achieve load balancing partitioning of indices.

To reduce tail latency, Jeon *et al.* [19] adopt machine learning to predict request service demand, and parallelize the predicted-to-be-long requests accordingly. Haque *et al.* [16] outperforms [19] by using *Incremental Parallelism* to dynamically increase parallelism to reduce tail latency. It precomputes parallelism policy offline, and adds parallelism using runtime information like system load. This method is very effective in reducing tail latency, and is complementary to Griffin. Because Griffin dynamically schedules parts of queries to CPU and GPU cooperatively, decompression and intersection of longer lists tend to run on GPU with massive parallelism and high throughput. As a result, Griffin can reduce the latency of many long queries if run on CPU. Griffin can combine the incremental parallelism techniques to further lower tail latency.

6 Conclusion

In the face of rapidly growing data sets and query loads, large-scale interactive information retrieval services must provide relevant results with consistent, low response times, which is very challenging. This paper presents Griffin, an IR system that explores the intra-query parallelism by adaptively scheduling parts of a query to GPU or CPU, to reduce the average response latency. Griffin introduces two GPU algorithms in Griffin-GPU: (1) a parallel EF decompression that provides both high compression ratio and fast decompression speed; and (2) a merge-based list intersection that achieves load balancing partitioning and efficient merging. We evaluate Griffin in a big dataset with real world queries and inverted lists generated from web data. The experimental results demonstrates that Griffin's adaptive scheduling can achieve an average speedup of 10x compared to a highly-optimized state-of-the-art CPU implementation, and 1.5x compared to Griffin-GPU. We also show that Griffin can effectively achieve a speedup of 6.6x, 8.3x, 10.4x, 16.1x, and 26.8x, for 80%, 90%, 95%, 99%, and 99.9% percentile response time over the CPU search.

In this paper, our "proof of concept" prototype assumes that the compressed lists are loaded in DRAM. This is a valid assumption, since in practice IR systems usually cache hot data in memory [33]. If data size is beyond the available DRAM, we could extend Griffin's scheduler to apply more advanced scheduling and data transfer management [20?].

It would be interesting to apply Griffin to more complex scenarios under heavy system loads with multiple users. We leave this as future work.

Acknowledgments

We would like to thank the anonymous reviewers for their many thoughtful suggestions that improve this paper. In particular, we would like to thank our shepherd, Idit Keidar, for her comments and patience.

References

[1] http://trec.nist.gov/.

[2] https://developer.nvidia.com/about-cuda.

[3] http://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html#group__CUDA__MATH__INTRINSIC__INT_1g43c9c7d2b9ebf202ff1ef5769989be46.

[4] https://nvlabs.github.io/moderngpu/intro.html#libraries.

[5] http://www.lemurproject.org/clueweb12.php.

[6] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *ASPLOS*, pages 19–34, 2014.

[7] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach. Fast k-selection algorithms for graphics processing units. *JEA*, 17:1–29, 2012.

[8] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[9] N. Bansal, K. Dhamdhere, and A. Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40(4):305–318, 2004.

[10] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[11] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[12] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *WWW*, pages 421–430, 2009.

[13] P. Elias. Efficient storage and retrieval by content and address of static files. *JACM*, 21(2):246–260, 1974.

[14] D. Feitelson. A survey of scheduling in multiprogrammed parallel systems, 1994. Research report. IBM T.J. Watson Research Center, 1994.

[15] O. Green, R. McColl, and D. A. Bader. Gpu merge path: A gpu merging algorithm. In *ICS*, pages 331–340, 2012.

[16] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ASPLOS*, pages 161–175, 2015.

[17] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *ISCA*, pages 314–325, 2010.

[18] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *EuroSys*, pages 155–168, 2013.

[19] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in web search. In *SIGIR*, pages 253–262, 2014.

[20] Y. Liu, H. Tseng, M. Gahagan, J. Li, Y. Jin, and S. Swanson. Hippogriff: Efficiently moving data in heterogeneous computing systems. In *ICCD*, pages 376–379, 2016.

[21] Y. Liu, H. Tseng, and S. Swanson. Spmario: Scale up mapreduce with i/o-oriented scheduling for the GPU. In *ICCD*, pages 384–387, 2016.

[22] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[23] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *TOCS*, 11(2):146–178, 1993.

[24] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path - parallel merging made simple. In *IPDPSW*, pages 1611–1618, 2012.

[25] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: The degree of parallelism executive. In *PLDI*, pages 26–37, 2011.

[26] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR*, pages 232–241, 1994.

[27] A. Singhal. Modern information retrieval: a brief overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001.

[28] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, pages 219–225, 2005.

[29] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, pages 963–972, 2011.

[30] S. Vigna. Quasi-succinct indices. In *WSDM*, pages 83–92, 2013.

[31] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. Milc: Inverted list compression in memory. *PVLDB*, 10(8):853–864, 2017.

[32] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, pages 993–1008, 2017.

[33] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. The impact of solid state drive on search engine cache management. In *SIGIR*, pages 693–702, 2013.

[34] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson. Ssd in-storage computing for list intersection. In *DaMoN*, pages 1–7, 2016.

[35] J. Wang, D. Park, Y. Papakonstantinou, and S. Swanson. Ssd in-storage computing for search engines. *TC*, 2016.

[36] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *IPDPSW*, pages 1–8, 2010.

[37] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[38] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[39] J. Zobel and A. Moffat. Inverted files for text search engines. *CSUR*, 38(2), 2006.

[40] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.