Efficient GPU-based Query Processing with Pruned List Caching in Search Engines

Dongdong Wang, Wenqing Yu, Rebecca J. Stones, Junjie Ren, Gang Wang, Xiaoguang Liu, Mingming Ren
Nankai-Baidu Joint Lab, College of Computer and Control Engineering, Nankai University, Tianjin, China
Email: wangdongdong@nbjl.nankai.edu.cn

***Abstract*—There are two inherent obstacles to effectively using Graphics Processing Units (GPUs) for query processing in search engines: (a) the highly restricted GPU memory space, and (b) the CPU-GPU transfer latency. Previously, Ao *et al.* presented a GPU method for lists intersection, an essential component in AND-based query processing. However, this work assumes the whole inverted index can be stored in GPU memory and does not address document ranking.**

In this paper, we describe and analyze a GPU query processing method which incorporates both lists intersection and top-k ranking. We introduce a parameterized pruned posting list GPU caching method where the parameter determines how much GPU memory is used for caching. This method allows list caching for large inverted indexes using the limited GPU memory, thereby making a qualitative improvement over previous work. We also give a mathematical model which can identify an approximately optimal choice of the parameter.

Experimental results indicate that this GPU approach under the pruned list caching policy achieves better query throughput than its CPU counterpart, even when the inverted index size is much larger than the GPU memory space.

1. Introduction

In many domains, we see applications of graphics cards, or graphics processing units (GPUs), extending from graphics processing into many other general-purpose applications, enabling the software to run substantially faster. It would be desirable to similarly use GPUs to accelerate query processing in search engines, but this is seen as a considerable challenge:

> ...we do not think that implementation of a complete search engine on a GPU is currently realistic.
> – Ding et al. (2009) [10]

In the context of query processing, there are two inherent obstacles due to the GPU architecture: the limited GPU memory and transfer costs.

Ao *et al*. [4] proposed a way to effectively utilizing the GPU for one major step in query processing: lists intersection. However, this work does not addresses document ranking, another major step in query processing, and also assumed the inverted index can reside in the limited GPU memory, which will often be unrealistic. In this paper, we extend the work by Ao *et al*. and propose a GPU-based method for query processing that includes both lists intersection and top-k document ranking without this unrealistic assumption. Thus, this paper provides the next step towards large-scale GPU-based query processing in search engines.

2. Background

2.1. Query processing

The overall aim in query processing is to quickly return the most relevant documents in response to a user query. To this end, a series of procedures need to be undertaken before retrieving the satisfactory documents. Here we give a brief overview of the relevant tasks involved in query processing.

2.1.1. Posting lists. Documents are assigned unique identifiers, called *docID*s and numbered $0, 1, \ldots, N-1$, where N is the number of indexed documents. Any term t has a corresponding *posting list*

$$\ell(t) := \langle s_t; (d_0, f_0), (d_1, f_1), \ldots, (d_{s_t-1}, f_{s_t-1})\rangle \qquad (1)$$

which lists the documents $d_0, d_1, \ldots, d_{s_t-1}$ containing the term t; here $s_t = |\ell(t)|$ is the *length* of $\ell(t)$, i.e., the number of documents in $\ell(t)$, and $f_i = f(t, d_i)$ denotes the number of occurrences of term t in document d_i. We also ensure $d_0 < d_1 < \cdots < d_{s-1}$.

2.1.2. Lists intersection. The first step in query processing is to identify which documents are relevant to all the terms in the user's query. If $\ell(t)|_{\text{doc}}$ denotes the set of documents in the posting list for t, then the documents relevant to the u-term user query $\{t_0, t_1, \ldots, t_{u-1}\}$ are

$$\ell(t_0)|_{\text{doc}} \cap \ell(t_1)|_{\text{doc}} \cap \cdots \cap \ell(t_{u-1})|_{\text{doc}}. \qquad (2)$$

For implementation, it is beneficial if we have the posting lists $\ell(t_0), \ell(t_1), \ldots, \ell(t_{u-1})$ sorted in ascending order of length (so $\ell(t_0)$ is the shortest posting list).

Ao *et al*. presented an efficient method for lists intersection on the GPU. They proposed a *hash segmentation* method for lists intersection, where the documents in each posting list (1) are divided into contiguous hash buckets. We search for a document x by first finding its hash value, then searching for x within the corresponding hash bucket.

2.1.3. Top-k ranking. Of the documents relevant to all query terms (i.e., those documents in (2)) we return the *top-k ranking*, i.e., the k most relevant documents. In order to rank documents d for their relevance to the user query q, some ranking function is used.

Ranking functions. PageRank [7] ranks documents according to their overall importance in the hyperlinked network of web pages. PageRank is a ranking function that does not vary with the user query q, so it's not typically used by itself as a ranking function.

Another well-known ranking function is BM25 [28], which varies with both the document d and the user query q (and two parameters $a \geq 0$ and $b \in [0, 1]$). Here we adopt the following equivalent form:

$$\mathrm{BM25}_{a,b}(d, q) := \sum_{t \in q} \overline{w}_t(q)\ \mathrm{IR}_{a,b}(d, t), \text{ where}$$

$$\mathrm{IR}_{a,b}(d, t) := \frac{(1 + a)f(d, t)}{a\,(1 + b(l_d - 1)) + f(d, t)},$$

where l_d is the length of d divided by the average document length, $f(d, t)$ is the number of occurrences of t in d, and the normalized *inverse document frequency weight* is

$$\overline{w}_t(q) := \frac{w_t(q)}{\sum_{\text{term } t' \in q} w_{t'}(q)}, \text{ where}$$

$$w_t(q) := \log \frac{N - s_t + 0.5}{s_t + 0.5},$$

where s_t is the number of documents containing term t (which is included in the posting list (1)).

Given a ranking function (such as PageRank or BM25), we iterate through the documents in the intersection (2), retaining only the top-k ranked documents, which is finally returned to the user in response to their query.

2.1.4. Early termination. Before inspecting all documents in the intersection (2) for their top-k ranking, we might be able to deduce theoretically that no uninspected document could possibly make the top-k list, in which case, we needn't inspect the remaining documents; this is called *early termination* [22]. This property is achieved by assigning the docIDs in a way that skews more promising documents towards smaller indices.

Some forms of early termination require only that uninspected documents are unlikely to make the top-k list, but the early termination method discussed in this paper guarantees the same top-k ranking as exhaustively evaluating the AND results, i.e., (2).

2.1.5. List caching. To reduce latency, frequently accessed posting lists $\ell(t)$ are cached in memory. There are a range of list caching policies in the literature; we will compare our proposed policy against two well-known policies: (a) *Qtf* [6], which caches the posting lists for the terms t with highest search frequency f_t, and (b) *QtfDf* [5], which caches the posting lists for the terms t with the highest $f_t/|\ell(t)|$.

2.2. GPUs

Modern GPUs have a massively parallel architecture consisting of thousands of cores. NVIDIA GPUs support *Compute Unified Device Architecture* (CUDA) [25], where threads are organized into *thread blocks*, which are further divided into *warps*; all threads within a warp are synchronously executed. A GPU computation is performed by invoking a *kernel*, executed by a *grid* of thread blocks.

GPU memory is organized into a hierarchy, and is typically far smaller than the (CPU side) system memory. The main GPU memory, called the *global memory*, is the largest but the slowest GPU memory, and is accessible to all the GPU threads. Data transferred from the CPU to the GPU goes into the global memory. *Shared memory* is a much faster, but much smaller memory space, and is only accessible to the threads in the corresponding thread block. There are other aspects to GPU memory, but these two memory types will be the relevant ones in this paper.

Transferring data between the system memory and GPU memory incurs latency [14], which increases with (a) the total amount of data transferred and (b) the number of transfers. Efficient GPU programming consequently needs to account for (a) GPU memory usage and CPU-GPU transfers (e.g. through caching [21]), and (b) distributing the workload among the thousands of GPU cores [18].

2.3. Related work

GPUs have been widely utilized in general-purpose application in recent years; here we review the relevant work. Fang *et al*. [11] proposed an in-memory GPU algorithm for three common database operations. He *et al*. [15] implemented common relational query processing algorithms taking advantage of the GPU hardware features (i.e., the high parallelism and memory bandwidth) and evaluated their system using different workloads including queries that involve complex data types and multiple query operators on data sets larger than the GPU memory, which is similar to the problem we investigate here. In addition, efficient GPU algorithms for specific database operations have also been studied, such as join [29], selection [13] and sorting [31, for example]. Other work utilizes various software optimization techniques to accelerate key-value processing [17], improve kernel execution efficiency [37], reduce PCIe data transfers [37], and support query co-processing with both GPUs and CPUs [16, for example].

When it comes to GPU-based query processing for search engine applications, work is rather limited. Ao *et al*. [4] made a significant step towards GPU-based query processing. They observed that their method performs lists intersection up to around 23 times faster using a NVIDIA GTX480 GPU. However, their method has some drawbacks:

- Their method is restricted to only one component of query processing, namely, lists intersection.
- They assumed that the inverted index can be entirely loaded into the GPU memory, which is unrealistic at larger scales.

Other prior work is mostly limited to lists intersection as well: Tsirogiannis *et al*. [34] presented lists intersection algorithms tailored to architecture with chip multiprocessors. Wu *et al*. [36] presented a GPU-based lists intersection framework in which queries are first grouped into batches

and then processed in parallel on the GPU. Zhang *et al*. [39] proposed a Bloom filter batched algorithm for intersection aimed at reducing the number of memory accesses.

The GPU parallel lists intersection algorithm *Parallel Merge Find* proposed by Ding *et al*. [10] is similar to what we explore here. While ranked query processing was also studied, the way they achieved this is different from the method presented here as, like in many other works, one of the (unrealistic) assumptions they make is that the index can be fully stored in the GPU memory. In addition, early termination, which plays an important role in accelerating the answering of top-k queries was not discussed in their work, while we incorporate it into the method proposed in this paper. Some of the pruning techniques by Ntlous *et al*. [24] are shared with this paper; the issue we investigate here is in essence more complicated than theirs: we not only ensure the correctness of the search result, but also incorporate optimization techniques into our method so that it can efficiently perform query processing over inverted index whose size is larger than the GPU memory capacity.

Another major aspect of this paper is developing a caching strategy which is functional when limited to the GPU memory. In this paper, we propose a pruning-based list caching algorithm, which shares ideas with the work of Tsegay *et al*. [33] who also propose a caching scheme that caches only those parts of the inverted lists that are actually processed. However, their dynamic pruning scheme is based on impact-ordered inverted lists [2] and processes the lists in a score-at-a-time [3] fashion. Instead, we propose a method which operates on an index that is reordered based on global scores, and also maintains the ascending order of docIDs (and thus it is friendlier to lists intersection and index compression algorithms). Additionally, Tsegay *et al*.'s method can not guarantee that the same top-k ranking is returned as exhaustively evaluating the AND results.

Index pruning is a technique used to reduce the inverted index size, where posting lists are pruned to remove unused entries, while still ensuring the accuracy of top-k results. As one example, Altingovde *et al*. [1] proposed pruning based on "query views", while we propose a method based on "visiting frequency". Some differences between these two works are as follows:

(a) We do not need to ensure the accuracy of top-k results on the GPU side, as CPU post-processing will take over in the event of an incompletely processed query. As such, we have more flexibility in pruning than [1].

(b) The GPU has a fixed memory size which cannot be exceeded, so we must prune the inverted index to XGB, say. The method in [1] results in a fix-sized fraction of each posting list is cached (and likewise for the treaps data structure [20]), whereas we cannot allow this. Here, we only ensure the fraction does not exceed λ.

(c) Visiting frequency is computed in the presence of early termination, using the same early termination method which is used for online query processing.

(d) The caching unit here is a block, and the block size can be adjusted.

3. The proposed method

3.1. Overview

Figure 1 depicts the workflow of the proposed GPU-based query processing method. The main CPU thread batches incoming queries (along with other necessary data) which are transferred to the GPU for query processing. After processing, the GPU returns a batch of top-k to the CPU and, if necessary, the secondary CPU thread performs post-processing. The major changes to traditional query processing are as follows:

3.1.1. GPU lists intersection. We assign each query to an independent GPU thread block and each docID in the shortest list $l(t_0)$ to a unique GPU thread. Then each GPU thread will search its corresponding docID in the other lists to determine if it belongs to the intersection. To accelerate this operation, we use the hash segmentation algorithm proposed in [4] to contract the search ranges. As the length of $l(t_0)$ may be larger than the thread block size B, there will be $\lceil |\ell(t_0)|/B \rceil$ rounds to cover all the docIDs in $\ell(t_0)$.

3.1.2. GPU pruned list caching. A portion of the GPU global memory is reserved for caching posting lists. (We do not use caching on the CPU side.) Instead of caching entire posting lists, we cache only pruned posting lists. This has the effect of reducing the GPU global memory required for storing the cache, thereby allowing more posting lists to be represented within the same space.

3.1.3. Early termination. We introduce a method of early termination that incorporates pruned list caching. If the top-k ranking computed on the GPU might be incomplete, a CPU post-processing stage is used to complete the task.

3.1.4. CPU post-processing. When the GPU is unable to completely perform top-k ranking, the remaining computation is performed by the CPU after the batch is returned. We implement the proposed method so that the CPU post-processing stage is run by a secondary CPU thread, thereby effectively "hiding" this component from the throughput (although it will still increase the query response time).

3.1.5. Batch processing. We collect queries until the buffer size is reached, after which the batch of queries is transferred to the GPU. Along with the user queries, we also transfer any posting lists not included in the GPU list cache. The results are also returned to the CPU side as a batch.

We consequently assume that incoming queries are being received rapidly enough so as to not introduce latency by waiting for other queries in the batch. (This assumption was similarly, albeit implicitly made in [4] and [36], where batching was also assumed to be possible.) Ding *et al*. [10], for instance, didn't incorporate batching, but without batching we unavoidably incur (a) many CPU-GPU transfers, and (b) many expensive kernel invocations.

3.1.6. CPU parallelism. We use two CPU threads: the main thread is for batch making and data transfers while the secondary thread is used solely for post-processing. These two parallel threads can work nearly independently.

3.2. Top-k ranking with early termination

The ranking function we utilize combines PageRank with BM25:

$$\mathrm{S}(d,q) = \alpha\,\mathrm{PageRank}(d) + (1-\alpha)\,\mathrm{BM25}_{a,b}(d,q) \quad (3)$$

for the document d and query q, as proposed by Zhang *et al.* [38]. (We also normalize the PageRank and BM25 scores to ensure they stay in the same score space.) If d does not belong to the lists intersection, we set $\mathrm{S}(d,q) = -\infty$. We use the method proposed in [7] to generate the PageRank, which is stored in GPU global memory for all documents d.

For a query $q = \{t_0, t_1, \ldots, t_{u-1}\}$, each GPU thread is assigned a document in the shortest posting list $\ell(t_0)$ to determine whether it belongs to the lists intersection, and its score according to the ranking function (3). As there may be multiple rounds of intersection, we use a k-element array K to maintain the current top-k candidates and their corresponding scores. At the end of each round, the docIDs belonging to the intersection set and their corresponding scores will be recorded in an array C, and K is updated the new top-k candidates.

To find the top-k scored documents in C, we divide C into groups of n elements where (n is some power of 2), then each group is sorted in descending order according to score and the maximum k scores within the group are gathered. These steps are repeated until the top scores gathered from the groups can fit into just one group, from which the final top-k results are obtained in the same way. We then merge the top-k results of C into K by performing a bitonic sort on them. We use Demouth's implementation of bitonic sort [9], which uses the shuffle instructions to achieve fast value exchange within a warp and avoid thread synchronization.

To facilitate early termination, we reorder the docIDs in the inverted index such that the most relevant documents are assigned smaller docIDs using the SSI method by Zhang [38]. The docIDs d are renumbered in descending order of *global score* $\mathrm{GS}(d)$, where

$$\mathrm{GS}(d) = \alpha\,\mathrm{PageRank}(d) + (1-\alpha)\max_{t\in d}\mathrm{IR}_{a,b}(d,t). \quad (4)$$

In this way, we ensure $\mathrm{GS}(d') \le \mathrm{GS}(d)$ whenever $d' \ge d$. Consequently, (3) and (4) imply

$$\mathrm{S}(d',q) \le \mathrm{GS}(d') \le \mathrm{GS}(d) \quad (5)$$

for all queries q and documents d and d' with $d \le d'$.

If, during the computation of the top-k ranked documents, the current bottom-ranked top-k document is ranked higher than $\mathrm{GS}(d)$ for some $d \in \ell(t_0)$, then (5) implies that all documents $d' \in \ell(t_0)$ with $d' \ge d$ will not make the top-k list, and we can terminate the search.

3.2.1. Pseudo-code. We present a pseudo-code description of the GPU side of the proposed method in Algorithm 1. Lines 3 to 17 include the lists intersection and score computation of documents assigned to a CUDA block. Lines 18 and 19 update the top-k candidates each time a new top-k list arises. Line 20 checks the early termination condition, which is performed after each CUDA block has been processed.

Algorithm 1 GPU lists intersection and top-k ranking with early termination

Input: Posting lists $\ell(t_0), \ell(t_1), \ldots, \ell(t_{u-1})$ for the u-term query $q = \{t_0, t_1, \ldots, t_{u-1}\}$ (possibly pruned due to caching); CUDA block size B
Output: top-k documents K for query q

1: $C.id \leftarrow (), C.score \leftarrow (), K.id \leftarrow (), K.score \leftarrow ()$
2: **for** round $b = 0, 1, \ldots, \lceil |\ell(t_0)|/B \rceil - 1$ **do**
3: **for all** threads in the CUDA block **do**
4: $T \leftarrow$ threadID
5: $d \leftarrow \ell(t_0)[T + Bb]$
6: $acc \leftarrow \alpha\,\mathrm{PageRank}(d) + (1-\alpha)\,\overline{w}_{t_0}(q)\,\mathrm{IR}_{a,b}(d,t_0)$
7: $C.id[T] \leftarrow d$
8: **for** $i = 1, 2, \ldots, u-1$ **do**
9: **if** $d \in \ell(t_i)$ **then**
10: $acc \leftarrow acc + (1-\alpha)\,\overline{w}_{t_i}(q)\,\mathrm{IR}_{a,b}(d,t_i)$
11: **else**
12: $C.id[T] \leftarrow$ NULL, $acc \leftarrow -\infty$
13: break
14: **end if**
15: **end for**
16: $C.score[T] \leftarrow acc$
17: **end for**
18: sort C by score in descending order
19: merge $C[0 : k-1]$ into K using bitonic sort
20: **if** $K[k-1].score \ge \mathrm{GS}\big(\ell(t_0)[B(b+1)]\big)$ **then**
21: break
22: **end if**
23: **end for**
24: **return** K

3.3. GPU pruned list caching

3.3.1. Visiting-frequency based block admission. To utilize list caching on the GPU side, we propose a static block-wise pruned list caching policy whereby blocks (of posting lists) are cached according to their likelihood of being checked when performing lists intersection and top-k ranking with early termination. We call this admission policy *visiting-frequency-based block admission* (or VFB).

In VFB, we split each posting list into contiguous blocks B, typically consisting of 256 docIDs (the end blocks could have smaller size). We say a query $\{t_0, t_1, \ldots, t_{u-1}\}$ *visits* a document $d \in \ell(t_i)$ when (a) some document $d' \in \ell(t_0) \cap \ell(t_1) \cap \cdots \cap \ell(t_i)$ satisfies $d' \ge d$ for some document $d \in B$, and (b) early termination hasn't kicked in before document d' is reached. We say a query *visits* a block

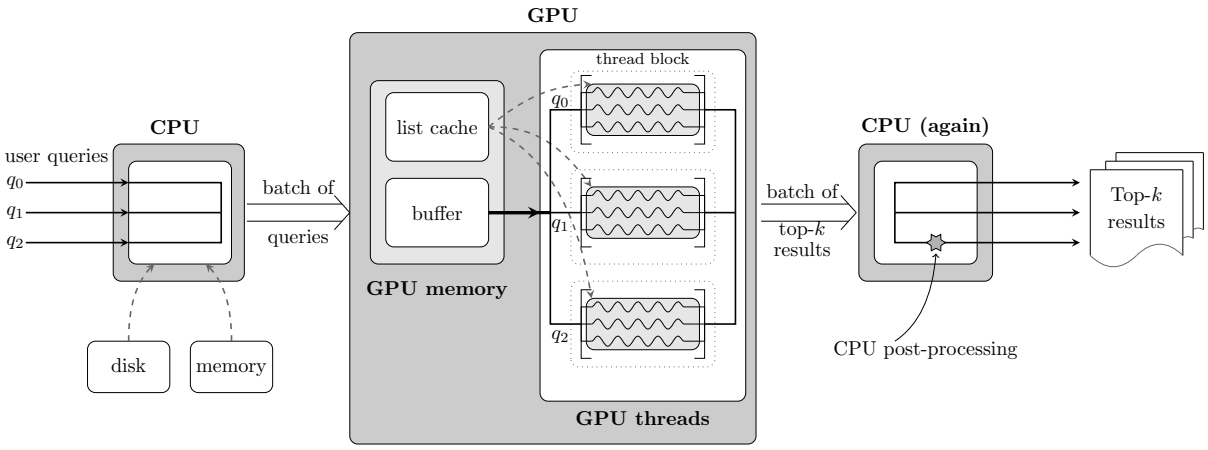


Figure 1: Workflow of the proposed method. The CPU receives queries and transfers them as a batch to the GPU together with the necessary posting lists not available in the GPU list cache. Each docID in the shortest list in each query is assigned to a GPU thread, which determines whether it belongs to the lists intersection and computes its score. The GPU combines the scores into a top-k list, which is returned to the CPU where, if necessary, CPU post-processing is applied.

if it visits some document whose docID is in the block. We define the *visiting frequency* $\mathrm{vf}_Q(t, B)$ of a block $B \subseteq \ell(t)$ in the query log Q as the number of queries in Q that visit B.

VFB admits the blocks within posting lists with the highest visiting frequencies that can fit within a limited memory space. When we have several blocks with the same visiting frequency, we prioritize the blocks $B \subseteq \ell(t)$ for which t occurs more frequently in the query log Q.

A toy example illustrating how VFB works is given in Figure 2. In this example, we have a query set $Q = \{q_0, q_1, q_2\}$ and an inverted index containing three posting lists $\ell(t_0)$, $\ell(t_1)$, and $\ell(t_2)$. The braces indicate which blocks are visited by which queries, and the shaded blocks are those which are admitted by VFB. In this example, we can admit all blocks with visiting frequencies ≥ 2. In the boundary case (visiting frequency = 1), we prioritize the blocks in $\ell(t_0)$ as t_0 occurs in more queries in Q than t_1 and t_2.

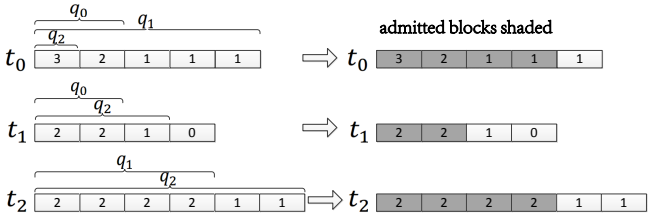


Figure 2: A toy example of how the visiting frequency of blocks is computed, and which blocks are admitted according to the VFB admission policy, given a 10-block memory space. Blocks are numbered with their visiting frequency.

3.3.2. Motivation. As with blocks, we define the *visiting frequency* $\mathrm{vf}_Q(t, d)$ of the document $d \in \ell(t)$ as the number of queries in Q that visit d. For the most frequent term t in the T06 query log (see Section 4.1), we compute the visiting frequency of each document $d \in \ell(t)$; the result is plotted in Figure 3. We can see a sharp drop in visiting frequency towards the end of $\ell(t)$, and this is due to early termination. This is the motivation for truncating posting lists to save cache memory. The idea is that the truncated part will only be infrequently accessed, and when it needs to be accessed, the consequent post-processing will not be overwhelming.

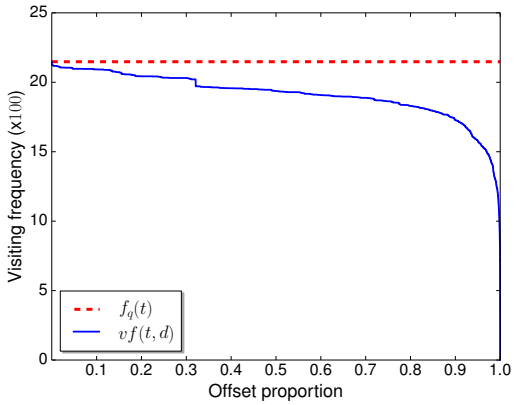


Figure 3: Visiting frequency for documents in $\ell(t)$. The offset proportion for the i-th document in $\ell(t)$ is $i/(|\ell(t)|-1)$.

3.3.3. Maximum admission ratio. We also consider restricting the proportion of posting lists cached to $\leq \lambda$, which we call the *maximum admission ratio*. A smaller λ-value can result in the representation of more lists in the GPU list cache, but will result in smaller proportions of posting lists being stored in the GPU list cache, and consequently increase the post-processing time. With an appropriately tuned λ-value, the CPU post-processing time can be overlapped by the main thread, which consequently does not decrease overall throughput.

3.3.4. Score computation is not affected. All the necessary values (like PageRank, term frequency, and document length) for score computation can be retrieved regardless of whether the list is full or partial, so the results of the ranking function are not affected by the caching of pruned lists. Consequently, when used in conjunction with CPU post-processing, this method guarantees the same top-k ranking as exhaustively evaluating the AND results.

3.3.5. GPU list cache size. A portion of the GPU global memory is reserved for list caching, denoted S_c; the remaining portion, denoted S_b, is mainly reserved for transferring the batched data from the CPU side to the GPU global memory. The choice of GPU list cache size (i.e., S_c vs. S_b) will have a significant effect on the overall performance: a smaller GPU list cache leaves more GPU memory available as a buffer, implying a larger batch size. However, a smaller GPU list cache also results in a lower cache hit rate, and thus more posting lists are required to be transferred within a batch. We experiment with varying GPU list cache size in Section 4.2, along with the maximum admission ratio λ.

3.4. CPU post-processing

When a user submits a u-term query $q = \{t_0, t_1, \ldots, t_{u-1}\}$ we ordinarily compute the lists intersection and top-k ranking from the posting lists $\ell(t_0), \ell(t_1), \ldots, \ell(t_{u-1})$. However, the GPU may have access only to pruned posting lists, which we denote

$$\ell^*(t_0), \ell^*(t_1), \ldots, \ell^*(t_{u-1}).$$

Let $m = \min_{0 \le i \le u-1} \max \ell^*(t_i)$. By design, each $\ell(t_i)$ and $\ell^*(t_i)$ agree for documents $\le m$. Therefore, if early termination kicks in for any document $d \le m$, then the GPU has found the correct top-k documents to return to the user. Otherwise, the GPU may not have found the correct top-k documents, and we perform a CPU post-processing stage.

A CPU post-processing stage is called when there might be a document with docID $> m$ omitted from the top-k documents. In this case, we use the CPU to continue performing the lists intersection and top-k ranking on the documents $\ell(t_0), \ell(t_1), \ldots, \ell(t_{u-1})$ from the first document $d \in \ell(t_0)$ satisfying $d > m$.

4. Experimental testing

4.1. Setup

For our experiments, we use the TREC GOV2 [35] dataset which consists of about 25 million documents crawled from web sites in the .gov domain during early 2004. For the evaluation, we divide the filtered Terabyte 2006 ($T06$) query set[1] consisting of approximately 75000 queries into two equal-sized parts: *T06-train*, the training query log, and *T06-test*, the test query log. We use T06-train to estimate the parameters and determine the static GPU list cache, while T06-test is used for performance evaluations. All experiments are run on a machine with a 3.50 GHz Intel Core i7-4770k CPU, with 32 GB of memory and a NVIDIA GeForce GTX Titan graphics card with 6 GB global memory. Where relevant, we measure the GPU list cache memory as a proportion of the size of the inverted index (i.e., what it would be if we cached everything).

We set the ranking function parameters in (3) to $a = 1.2$, $b = 0.75$, and $\alpha = 0.3$; while they affect the quality of the final top-k ranking, they do not significantly affect the throughput and response time of the proposed method.

4.2. GPU cache size and admission threshold

Two tunable parameters will directly affect the number of posting lists represented in the GPU list cache thereby influencing the system's performance: the amount of GPU memory used for caching (see Section 3.3.5) and the admission threshold λ (see Section 3.3.3).

4.2.1. Optimizing the GPU cache size. We divide the GPU memory primarily into two parts: cache space S_c and buffer space S_b. A larger S_c usually means higher hit rate, hence a better overall performance. However, this is not always true on our GPU-based query processing system where part of the GPU memory needs to be reserved as buffer space which directly determines the batch size. The smaller S_b is, the fewer queries will be processed by the GPU in a single kernel invocation. Accordingly, we seek the optimal partition between cache and buffer space given a fixed-size GPU memory space whereby the system throughput can be maximized within a query response time constraint θ. Note that the parameter θ here also indicates the query type: $\theta = \infty$ for *non-interactive queries* and $\theta < \infty$ for *interactive queries*.

We build batches by adding queries along with the uncached posting lists (i.e. the posting lists involved which are not in the GPU list cache) and other necessary auxiliary information, until its size reaches S_b, which is then transferred to the GPU for processing, after which the results are returned (invoking CPU post-processing, if necessary). The total time for processing all queries in a collection of N queries can thus be mathematically modeled by

$$N \cdot [\mathrm{c}(n) + \mathrm{t}(\mathrm{S}_b) \cdot \overline{\mathrm{dq}}(\mathrm{S}_c)] \tag{6}$$

where $\mathrm{c}(n)$ is the average calculation time per query (including the pre-processing time, i.e., the time spent determining which data needs to be transferred, batch making on the CPU, and query processing time on the GPU) and n is the number of queries per batch (the *batch volume*); $\mathrm{t}(\mathrm{S}_b)$ is the average CPU-GPU transfer time per MB; and $\overline{\mathrm{dq}}$ is the average volume of data transferred per query. Here we ignore the CPU post-processing time since it can be mostly hidden by the CPU main thread. The functions c, t, and $\overline{dq}$ will vary with batch volume n, buffer size S_b, and cache size S_c, respectively, while we assume other factors are insignificant

1. Queries with empty intersection results are excluded.

and can be omitted for simplicity. The system throughput is thus given by

$$\mathrm{TP} = 1/\big(\mathrm{c}(n) + \mathrm{t}(\mathrm{S}_b) \cdot \overline{\mathrm{dq}}(\mathrm{S}_c)\big). \quad (7)$$

In order to determine how TP in (7) varies with the cache size S_c, we compute the functions c, t, and $\overline{dq}$ individually, as shown in Figures 4(a) and 4(b), and Figure 4(c).

For each S_c-value we test, we set S_b as the maximum buffer size which satisfies the response time constraint θ. In Figure 5 (top), we present the fitted throughput distributions under two different θ constraints. As a check of the model, we also give experimental results in Figure 5 (bottom). These figures show, for various caching algorithms, the maximum difference in S_c and S_b between theoretical and empirical maxima is at most around 4% (this occurs in the case of θ=150ms for QtfDf where the optimal S_c to M proportion is 0.89 in the fitted model and 0.93 in the empirical result). This model can be used to give an initial estimate of the optimal partition between the cache space and buffer space. Subsequently, for each S_c-value near the estimate, we determine the maximum S_b by benchmark testing, which gives the actual optimum partition.

4.2.2. Influence of the admission threshold. In varying λ from 0.75 to 1, we see variation in the GPU cache hit rate of around 5 percentage points, so λ has some impact on the hit rate. However, this variation is swamped by the unavoidable low hit rate incurred by only being able to cache a small fraction of the inverted index (e.g. around a 30% hit rate when caching 10% of the inverted index). The hit rate, in itself, is not necessarily a good indicator of overall performance [12], [26], so we instead explore the performance as λ varies using query processing time.

Figure 6 gives the average query processing time as the GPU list cache size and λ varies, split into the GPU processing time and the CPU post-processing time. (For the purposes of profiling, we use a single CPU thread for this experiment, so the CPU post-processing time is not hidden; the buffer size is also fixed at 800MB.) We see that as the GPU cache size increases, the GPU processing time decreases. Moreover:

(A) Increasing the GPU list cache size results in (a) a better cache hit rate and (b) more queries fitting into a single batch[2], resulting in a smaller transfer volume (further details are in Section 4.3.1), which results in decreasing the GPU processing time since more queries can be processed on GPU in one kernel invoking which is beneficial for utilizing the computation power of GPU.

(B) As intended, decreasing λ rebalances the workload: reducing the GPU workload, but increasing the CPU post-processing.

4.3. VFB, Qtf, and QtfDf

2. Since fewer posting lists need to be included with the queries in a batch, the number of queries per batch increases despite the buffer size decreasing as the GPU list cache size increases (given a fix-sized GPU memory space).

4.3.1. CPU-GPU transfers. In Figure 7, we plot the CPU-GPU transfer volume for the proposed caching policy (VFB), and compare it with Qtf and QtfDf. VFB decreases the transfer volume while maintaining a cache hit rate comparable to that of Qtf and QtfDf. While QtfDf has a much higher hit rate than other policies, it incurs a large transfer volume overhead (as we expect from prior work [5], [32]).

4.3.2. Throughput. Figure 8 compares the throughput of the proposed GPU-based query processing method with VFB vs. Qtf and QtfDf. In this experiment, we restrict the total GPU memory available. We present the results achieved by the best partition scheme found under the guidance of the aforementioned model. We describe the queries in Figure 8 as non-interactive, as response time is not a priority.

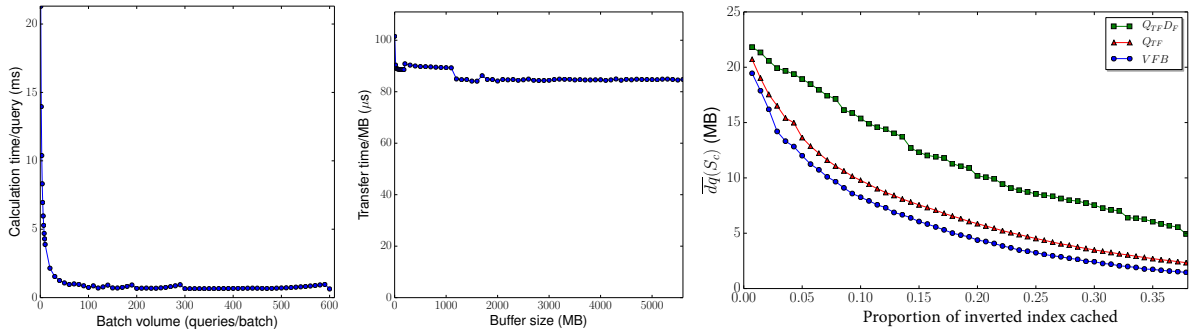
We see that the throughput of the proposed GPU-based query processing method increases almost linearly with available GPU memory size regardless of the specific caching policy. So we can expect even higher throughput can be achieved as GPUs with larger memory space become available. When it comes to the effectiveness of the three caching policies involved, VFB outperforms Qtf and QtfDf consistently, and by up to 30% and 140%, respectively. We attribute the difference in throughput between Qtf and QtfDf to frequently queried terms with long posting lists: these lists do not reside in the GPU list cache under QtfDf, so the proposed method adds them to the batch which is transferred from the CPU, increasing the transfer overhead.

Figure 9 performs a similar comparison of VFB vs. Qtf and QtfDf but with required response times (interactive queries). For each caching method, we exhaustively test every partition near the predicted "best" partition given by the model in Section 4.2.1 given the total memory space; from these, we select the highest throughput subject to the time constraint θ. With a fixed-sized GPU memory space and a specific caching method, different time constraints θ are better achieved with different partitions. (We don't test λ-values with $\lambda < 0.75$ as, in our experimental setting, choosing a smaller λ-value would result in suboptimal throughput, and would not affect Figure 9.)

We see that high throughput can also be achieved in the proposed GPU-based query processing method with VFB even for interactive queries, i.e., where response time is taken into consideration. The VFB policy consistently outperforms its counterparts (Qtf and QtdDf) with this additional restriction.

4.4. Comparison with mutlithreaded CPU

We also compare the throughput and speedup of the proposed scheme vs. a CPU multithread version for different k-values (k as in "top-k"). For the implementation of top-k ranking on CPU, we adopt the same hash segmentation method for lists intersection and early termination algorithm. Similarly, each query is assigned as a subtask to one CPU thread in a round-robin fashion to determine the most relevant k documents.



(a) c(n) in (7) (b) t(S_b) in (7) (c) $\overline{\text{dq}}$ in (7) (with $\lambda = 0.75$)

Figure 4: Computing the functions c and t, and $\overline{\text{dq}}$ for Qtf, QtfDf, and VFB.

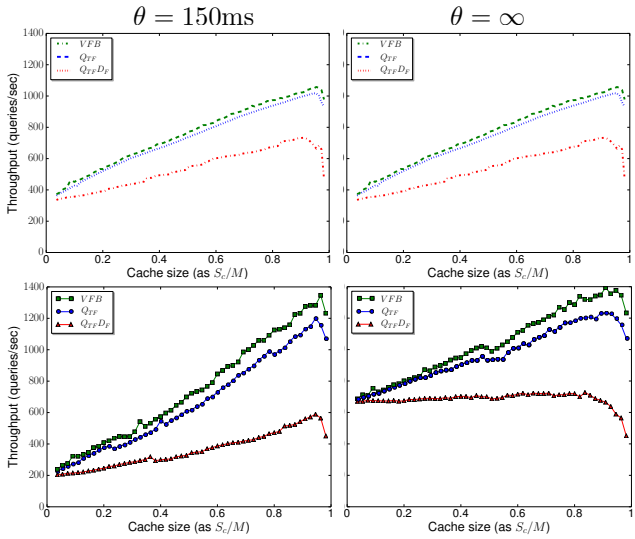


Figure 5: Throughput: model fitted results (top) and empirical results (bottom); $M = 5600$MB and $\lambda = 0.95$. Here, different colors and line types are used for fitted and empirical results separately.

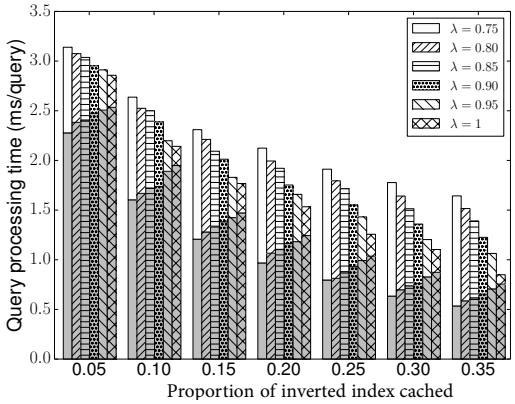


Figure 6: Processing time for T06-test as the admission threshold λ and the GPU list cache memory vary. The white and gray components respectively show the proportion of time from CPU post-processing and the GPU processing.

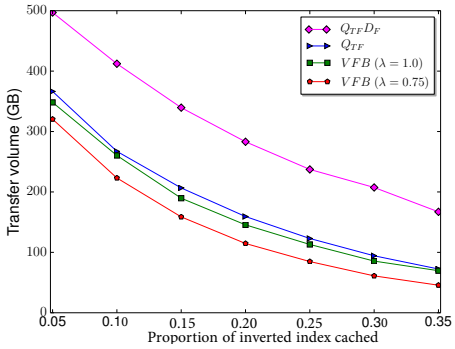


Figure 7: Transfer volume for T06-test for VFB, Qtf and QtfDf.

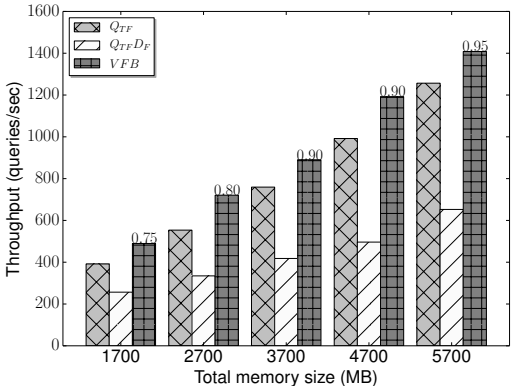
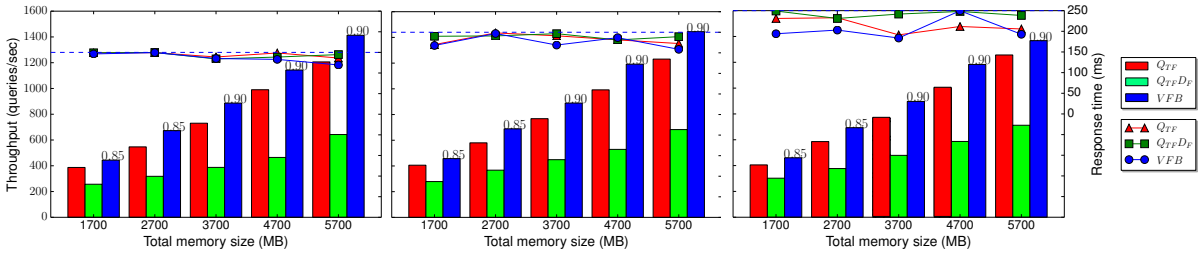


Figure 8: Throughput for non-interactive queries. The λ reported for VFB is labeled on each corresponding bar.

In Figure 10 (top), we present the throughput of top-k ranking with early termination. Compared with the CPU best case (i.e., CPU-8), the proposed method improves performance 3%, 19%, 24%, and 24% when $k = 1$, 5, 10, and 15, respectively. Importantly, these improvements are achieved where the total GPU memory size is about 40% of the index size; the improvements will be more significant given a GPU with a larger memory space. An unavoidable potential issue with this experimental approach



(a) response time $\leq$ 150ms (b) response time $\leq$ 200ms (c) response time $\leq$ 250ms

Figure 9: Throughput and average response time for (interactive) queries constraining the response time under the caching policies VFB, Qtf, and QtfDf. The λ reported for VFB is the optimal value among $\lambda \in \{0.75, 0.80, 0.85, 0.90, 0.95, 1\}$. For each figure, the left y-axis label serves for the bar plots while the right one corresponds to the line plots.

is that the authors' implementation might disproportionately favor either the CPU or GPU. In an effort to minimize this possibility, modifications to the code originally by Ao *et al.* [4] were made solely by the first author, and an approximately equal amount of optimization was applied to both. While GPU versions of more sophisticated methods may improve components of our implementation (e.g. lists intersection [8], [19]), they would benefit both the CPU and the GPU implementations (but would require carefully balanced implementation on the GPU; see Section 2.2).

Making simultaneous use of both the CPU and GPU is an interesting future research direction. As a preliminary result, we investigate the speedup using a straightforward method: multiple threads on CPU are introduced to concurrently process queries according to a specific forwarding policy. Here, we make the queries whose terms are all cached a priority for the GPU and a carefully tuned parameter is used to balance the workload distribution between CPU and GPU. The results are plotted in Figure 10 (bottom), where "combined" denotes this concurrent method. We see even this simple forwarding policy is effective at improving speedup, and expect a well-developed approach would perform better.

5. Concluding remarks

In this paper we propose a GPU-based query processing method, extending Ao *et al.*'s proposed GPU lists intersection method to encompass GPU top-k ranking. We design a pruned list caching method, which results in more lists being represented in the limited GPU memory and reduced CPU-GPU data transfers. The list caching method is parameterized, so we have flexibility in deciding the amount of GPU memory for the list cache. Moreover, we propose a mathematical model which gives an efficient way to choose a suitable GPU list cache size. In this way, we overcome the unrealistic assumption that the whole inverted index can be stored in GPU memory.

As in prior work, the proposed method utilizes query batching, but if user queries are not incoming fast enough to enable batching, there are two straightforward solutions: (a) simply do not use GPU query processing (instead performing query processing on the CPU as usual) until the assumption becomes true, or (b) invoke transfers before a batch is full in order to improve the response time, but at the expense of CPU-GPU transfers and throughput. In this setting, throughput is no longer a meaningful metric as it is limited by the rate of incoming queries.

Search engine servers also use a range of caches, such as a query cache, which would further limit access to queries by a back-end server, where GPU-based query processing would be beneficial. Thus, as a future research direction, we propose designing a dynamic caching algorithm which can respond to the inhomogeneity of the query stream. This technique could further benefit from CPU-GPU job scheduling, where some queries are processed purely by the CPU, to further reduce data transfer and decrease the query latency. Further, the proposed VFB caching method only caches posting lists, so it could potentially benefit from more sophisticated caching methods (e.g., additionally caching query results [23], [30] or partial intersection results and pre-computed scores [27]).

6. Acknowledgments

This work is partially supported by NSF of China (grant numbers: 61373018, 11301288, 11550110491), Program for New Century Excellent Talents in University (NCET130301). Stones was also supported by the Tianjin 1000 Scholars Plan.

References

[1] I. S. Altingovde, R. Ozcan, and O. Ulusoy. Static index pruning in web search engines: Combining term and document popularities with query views. *ACM Trans. Inf. Syst.*, 30, 2012.

[2] V. Anh. *Impact-based document retrieval*. PhD thesis, The University of Melbourne, 2004.

[3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. SIGIR*, pages 372–379, 2006.

[4] 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. *Proc. VLDB Endowment*, 4:470–481, 2011.

[5] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. SIGIR*, pages 183–190, 2007.

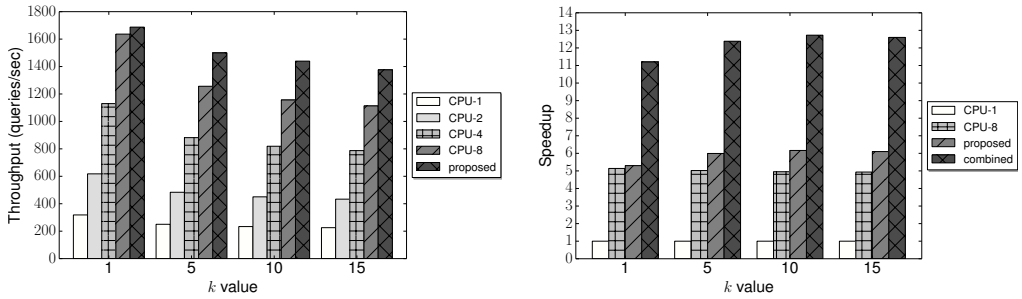


Figure 10: Throughput (left) and speedup (right) of the proposed GPU-based query processing scheme vs. a multithreaded CPU-based scheme. In the speedup comparison, we also include a preliminary CPU-GPU cooperative scheme. CPU-x indicates that x CPU threads are used.

[6] R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *Proc. SPIRE*, volume 2857, pages 56–65, 2003.

[7] S. Brin and v. p. y. p. Page, Lawrence journal=Comp. Networks ISDN Syst. The anatomy of a large-scale hypertextual web search engine.

[8] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29, 2010.

[9] J. Demouth. Shuffle: Tips and tricks. http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf, 2013.

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

[11] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. GPUQP: query co-processing using graphics processors. In *Proc. SIGMOD*, pages 1061–1063, 2007.

[12] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proc. WWW*, pages 431–440, 2009.

[13] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proc. SIGMOD*, pages 215–226, 2004.

[14] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proc. ISPASS*, pages 134–144, 2011.

[15] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Systems*, 34, 2009.

[16] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled CPU-GPU architectures. In *Proc. VLDB Endowment*, pages 329–340, 2014.

[17] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: scaling-up scale-out key-value stores. In *Proc. SoCC*, pages 43–57, 2015.

[18] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Exploiting core criticality for enhanced gpu performance. In *Proc. SIGMETRICS*, pages 351–363, 2016.

[19] A. Kane and F. W. Tompa. Skewed partial bitvectors for list intersection. In *Proc. SIGIR*, pages 263–272, 2014.

[20] R. Konow, G. Navarro, C. L. A. Clarke, and A. López-Ortiz. Faster and smaller inverted indices with treaps. In *Proc. SIGIR*, pages 193–202, 2013.

[21] L. Li, A. B. Hayes, S. L. Song, and E. Z. Zhang. Tag-split cache for efficient GPGPU cache utilization. In *Proc. ICS*, pages 43:1–43:12, 2016.

[22] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. *Proc. VLDB Endowment*, 29:129–140, 2003.

[23] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. *World Wide Web Journal*, 9:369–395, 2006.

[24] A. Ntoulas and J. Cho. Pruning policies for two-tieded inverted index with correctness guarantee. In *Proc. SIGIR*, pages 191–198, 2007.

[25] NVIDIA. NVIDIA CUDA C programming guide. 2015.

[26] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Cost-aware strategies for query result caching in web search engines.

[27] R. Ozcan, I. Sengor Altingovde, B. Barla Cambazoglu, F. P. Junqueira, and Ö. Ulusoy. A five-level static cache architecture for web search engines. *Information Processing & Management*, 48:828–840, 2012.

[28] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. *NIST special publication*, page 109, 1995.

[29] R. Rui, H. Li, and Y.-C. Tu. Join algorithms on GPUs: A revisit after seven years. In *Big Data*, pages 2541–2550, 2015.

[30] P. C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proc. SIGIR*, pages 51–58, 2001.

[31] H. Shamoto, K. Shirahata, A. Drozd, H. Sato, and S. Matsuoka. Gpu-accelerated large-scale distributed sorting coping with device memory capacity. *IEEE Trans. Big Data*, 2:57–69, 2016.

[32] J. Tong, G. Wang, and X. Liu. Latency-aware strategy for static list caching in flash-based web search engines. In *Proc. CIKM*, pages 1209–1212, 2013.

[33] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. In *Proc. CIKM*, pages 987–990, 2007.

[34] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proc. VLDB Endowment*, 2:838–849, 2009.

[35] E. M. Voorhees. Overview of TREC 2003. In *Proc. TREC*, pages 1–13, 2003.

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

[37] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Proc. MICRO*, pages 107–118, 2012.

[38] F. Zhang, S. Shi, H. Yan, and J.-R. Wen. Revisiting globally sorted indexes for efficient document retrieval. In *Proc. WSDM*, pages 371–380, 2010.

[39] F. Zhang, D. Wu, N. Ao, G. Wang, X. Liu, and J. Liu. Fast lists intersection with Bloom filter using graphics processing units. In *Proc. SAC*, pages 825–826, 2011.