GPU-Accelerated Block-Max Query Processing

Haibing Huang, Mingming Ren*, Yue Zhao,Rebecca J. Stones, Rui Zhang, Gang Wang, Xiaoguang Liu

Nankai-Baidu Joint Lab, College of Computer and Control Engineering, Nankai University, Tianjin, 300071, China
{hbhuang,renmingming, zhaoy,rebecca.stones82, zhangruiann, wgzwp, liuxg}@nbjl.nankai.edu.cn

Abstract. In this paper, we propose a method for parallel top-k query processing on GPU(s). We employ a novel partitioning strategy which splits the posting lists according to document ID numbers. Individual GPU threads simultaneously perform top-k query processing within their allocated subsets of posting lists, the results of the query are merged to give the final top-k results. We further design a CPU-GPU cooperative query processing method, where a majority of queries involving shorter posting lists are processed on the GPU side. We experiment with AND, OR, WAND, and Block-Max WAND (BMW) queries, with experimental results showing a promising improvement in query throughput, particularly in the case of BMW queries.

Keywords: Information Retrieval, GPU, Index Partition, Query Assignment

1 Introduction

Search engines face a large number of queries from users. To provide high query throughput and response time, current commercial search engines use large clusters consisting of thousands of nodes. Each node is responsible for processing a subset of the whole posting data. Distributing the workload over a large number of nodes facilitates the timely return of top-k results to users. In this paper, we design a GPU-accelerated query processing method, where the workload is distributed over GPU threads (and even the CPU).

In many domains, we see applications of graphics processing units (GPUs) extending from their original purpose (graphics processing) into a wide range of general-purpose applications, primarily for the single goal of making software run faster. GPU programming requires carefully balancing workloads, data transfers, and utilizing the GPU's memory hierarchy, and programs are typically custom built for an application.

Several GPU-based query processing techniques have been proposed previously (see Section 2.5 for a review). The research presented here takes three new directions: (a) we design a method which can subdivide the task of generating the top-k results for a single query among threads, (b) we address several query processing strategies, such as WAND [3] and block-max WAND (BMW) [6] (WAND

queries for block-max indexes), and (c) we extend the proposed method to utilize both the CPU and GPU for query processing. In addition, previous work has typically assumed that the inverted index can be fully loaded into the GPU memory, which is unrealistic for large indexes, which we do not assume here.

The remainder of this paper proceeds as follow: Section 2 gives the background and related research of the information retrieval and parallel query processing. Section 3 presents our method of GPU-based and CPU-GPU cooperation algorithm. Section 4 presents the experimental results of our method. Finally, Section 5 concludes and discusses future work.

2 Background and Related Work

2.1 Block-max index

Documents are assigned *docID* numbers $0, 1, \ldots, N-1$, where N is the number of indexed documents. A term t has a corresponding *posting list*, denoted

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

which lists the documents $d_0, d_1, \ldots, d_{s_t-1}$ containing the term t. The posting list has *length* s_t and the number of occurrences of term t in document d_i is denoted $f_i = f(t, d_i)$. We assume $d_0 < d_1 < \cdots < d_{s-1}$. Posting lists belong to a large index known as the *inverted index*.

The differences between consecutive docIDs in a posting list are referred to as d-gaps, and are numerically much smaller than the raw docID values. As such, d-gaps are typically used in place of the raw docIDs to reduce the inverted index size with effective compression method.

The inverted index is usually stored in a compressed form to significantly reduce its size. At the same time, its contents need to be readily accessible to allow fast query processing. Many inverted index compression techniques having been proposed [17] balancing these goals. In this paper we use *NewPFD* [15] to compress the posting lists and corresponding frequency lists with a block size of 64 (although the proposed method could use other compression techniques).

Ding and Suel [6] proposed a *block-max index* data structure, where posting lists are partitioned into *blocks* comprising, say, 64 docIDs (corresponding to the 64 docIDs in the *NewPFD* blocks). The docIDs and their frequencies are stored in a compressed format, and are stored along with the least and greatest docIDs and the maximum "impact score" (essentially, the maximum contribution to the top-k ranking). In this way, every *NewPFD* block can be decompressed separately, and the top-k results can be computed with early termination. This was shown to be an effective technique for improving the performance of WAND query processing. In this paper, we borrow aspects of this index method.

2.2 Query Processing

In query processing, for documents relevant to a query, we compute a *score*, and those with the highest score are considered the most relevant. To this end, we

traverse all of the relevant posting lists (those with terms in the query) from beginning to end. For index traversal, we use a *Document-At-A-Time* (DAAT) approach, where each list has a pointer that points to the "current" docID, which moves forward to identify the docIDs which are common among the relevant posting lists for conjunctive query. The document scores are computed while traversing the lists, and we can use min-heap data structure to store the top-k results.

The DAAT approach can work well for conjunctive (AND) and disjunctive (OR) query processing [3], and WAND [3] and BMW [6] can also be implemented using the DAAT approach. The WAND and BMW algorithm can avoid fully evaluating the score of all documents in the posting list of each term belonging to a given query, a smart pointer movement technique is used to skip many documents that would be evaluated by an exhaustive algorithm. In this paper, we will take these four kind query processing strategies into consideration.

2.3 Scoring

Ranking functions are used to give a numerical score for a document d and a query q. BM25 [9] is a well-known ranking function, which varies with both d and q (and two parameters $a \geq 0$ and $b \in [0, 1]$), given by

$$\mathrm{BM25}_{a,b}(d, q) := \sum_{t \in q} w_t(q)\ \mathrm{IR}_{a,b}(d, t) \tag{2}$$

where

$$\mathrm{IR}_{a,b}(d, t) = \frac{(1 + a) f(d, t)}{a\big(1 + b(l_d - 1)\big) + f(d, t)} \tag{3}$$

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

$$w_t(q) = \log \frac{N - s_t + 0.5}{s_t + 0.5} \tag{4}$$

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

During query processing, we use DAAT approach to iterate through the relevant posting lists, retaining the top-k highest scoring documents, the top-k results are returned to the user finally.

2.4 GPUs

Modern GPUs have a massively parallel architecture consisting of thousands of cores. NVIDIA brand GPUs support *Compute Unified Device Architecture* (CUDA) [8], where threads are organized into *thread blocks* and *thread blocks* are organized into *Grid*. A GPU computation is performed by invoking a *kernel* which is executed by a *grid* of thread blocks.

GPUs have their own memory, which is organized into a hierarchy, and the GPU memory is usually far smaller than the (CPU side) system memory. The relevant GPU memories for this paper are: (a) *Global memory*, 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. (b) *Shared memory*, which is much faster than the global memory, but is much smaller, and is only accessible to the threads in the corresponding thread block. (c) *Registers*, the fastest but the most scarce memory resource. Each multiprocessor has a set of registers partitioned among the warps (which partition thread blocks). Overall, registers are unique to a thread, shared memory is unique to a block, and global memory exist across all blocks.

Efficient GPU programming requires careful consideration of (a) GPU memory usage, (b) CPU-GPU transfers, (c) workload distribution. and (d) parallel algorithm.

2.5 Related work

GPUs have been widely utilized in general-purpose application. Zhang *et al.* [18] proposed an effective algorithm which can parallelize DNN training on multiple GPU cards in a single computing server. Fang *et al.* [7] proposed a in-memory GPU algorithms, which support three common database operations. Agrawal [1] utilized data parallel accelerators and a software architecture, Rhythm, to address throughput and efficiency demands of future server workloads.

To speed up the query processing, there are many previous papers that focus on how to efficient parallel query processing. Rojas *et al.* [10] proposed the parallelization the Block-Max WAND algorithm using two-level ranking on distributed search engine. Ding *et al.* [4] achieved good performance using speciallized mechanisms for executing batch queries. Tatikonda *et al.* [12] achieved more than five times reduction average query processing time by exploiting parallelism at the finest-level of granularityin on an eight-core system.

There are also several previous papers that focus on GPU-based query processing. Ding *et al.* [5] presented a general architecture for GPU-based query processing and proposed a parallel lists intersection algorithm with the GPU, but queries are dispatched to CPU or GPU one by one, incurring a impractical transfer overhead. Wu *et al.* [14] presented a GPU-based lists intersection framework in which queries are first grouped into batches, and then processed in parallel on the GPU using their proposed PARA algorithm. Zhang *et al.* [16] proposed a Bloom filter batched algorithm for intersection aiming at reducing the number of memory accesses for each GPU thread. Ao *et al.* [2] proposed linear regression and hash segmentation algorithms for GPU-based lists intersection (a component of query processing), which was up to around 23 times faster using a NVIDIA GTX480.

However, previous GPU-based query processing methods have some limitations:

- Methods have been restricted to conjunctive (AND) query processing.

– The total inverted index is typically assumed to be residing in the GPU global memory, which might be assuming an unrealistic GPU memory size on current hardware for large indexes.

In this paper, we do not assume the whole inverted index resides in the GPU memory. To cope with this, we batch transfer the user queries together with the relevant posting lists not residing in the GPU memory. Another major aspect of this paper is also incorporating OR, WAND, and BMW strategies for query processing.

3 The Proposed Method

3.1 Overview

Figure 1 illustrates the proposed GPU-based query processing framework.

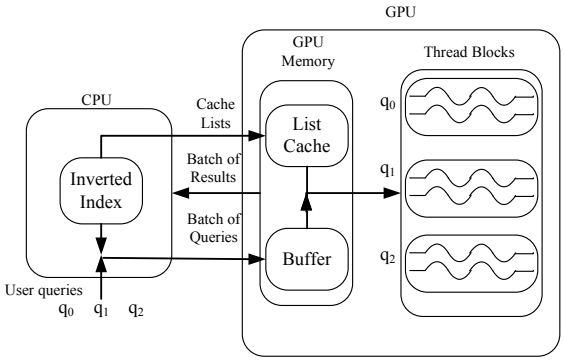

Fig. 1. The workflow of GPU-based query processing.

The entire inverted index is assumed to reside in the (CPU-side) system memory. In the case that the index's size actually exceeds the CPU memory capacity, some queries will require disk access to be processed. Without modifying the inverted index compression method, this will be unavoidable and an essentially constant overhead (i.e., will not substantially vary with the design of the GPU query processing method). and we assume user queries are continuously incoming rapidly enough to allow them to be batched and transferred jointly to the GPU. The queries will be added to the current batch along with any required posting list not residing in the GPU list cache. Once the batch size reaches a certain threshold, or the number of queries in the batch reaches the maximum value that can be processed, the batch is transferred to the GPU global memory.

The GPU global memory space contains two main parts: a *list cache*, which contains a portion of the whole inverted index, and a *buffer space*, which contains the batches. The posting lists residing in the list cache is determined by some

admission policy (described in Section 4.1). A hash table is maintained on the CPU side to record which posting lists in the GPU list cache.

Algorithm 1 shows our proposed GPU-based query processing algorithm. The algorithm assigns a thread block the task of generating the top-k results for a single user query. An individual thread within a thread block generates the local top-k results on its assigned subset of the docIDs. Specifically, the set of docIDs $\{0, 1, \ldots, N-1\}$ is partitioned into d-sized ($d = \lceil N/P \rceil$) intervals $\{0, 1, \ldots, d-1\}$, $\{d, d+1, \ldots, 2d-1\}$, and so on, with each thread being assigned to work on one part, and we have P threads in every thread block. In our experiments, we will test a range of P-values.

Algorithm 1 The proposed GPU-based query processing algorithm

Input: a batch of queries Q
Output: top-k results for each query in Q
1: Transfer Q to the GPU buffer space
2: **for** thread block $b_{\text{id}} \in \{0, 1, \ldots, (|Q| - 1)\}$ **do**
3: **for each** thread $t_{\text{id}} \in \{0, 1, \ldots, P - 1\}$ **do**
4: Compute local top-k results by using a query processing strategy.
5: **end for**
6: Synchronize threads for thread block b_{id}
7: Merge local top-k results for thread block b_{id}
8: **end for**
9: Transfer every thread block top-k results to CPU

Figure 2 shows a toy example of the inverted index partition strategy. The four threads T0, T1, T2, and T3 are responsible for processing the subsets of three compressed block-max posting lists respectively. Thread T0, for example, is responsible for the subsets of posting lists in first left dashed box, i.e., the docIDs interval $\{0, \ldots, 999\}$.

After the posting lists are partitioned for each thread, the threads perform query processing on their assigned sub-posting lists and compute the *local top-k results*. Before computing the merge operation in a thread block, a synchronization barrier is needed. Once all of the local top-k results are obtained, the threads in a thread block merge every thread's local top-k results to compute the thread block top-k results. we select insert sort method to complete the merge operation. Once the thread block top-k results for the whole batch of queries has been computed, they are transferred to the CPU as a batch, and the final results can be displayed to the users.

3.2 CPU-GPU cooperative version

Algorithm 1, by itself, would result in a large amount of CPU idle time. To avoid this, we propose a CPU-GPU cooperative algorithm, which is a modified version of the proposed GPU query processing method. Essentially, some queries

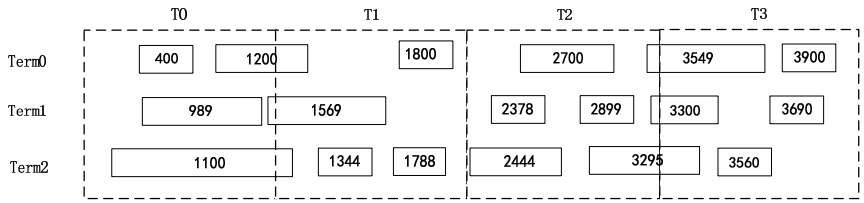

Fig. 2. A toy example of the document-based index partition strategy. Threads T0, T1, T2, and T3 are responsible for their assigned subsets of three compressed block-max posting lists. The number in the solid box shows the max docID in corresponding compressed a NewPFD block. The subsets are indicated by a dashed boundary and each subset's docIDs interval is 1000 in this example. For term Term0, thread T0 is responsible for the whole first NewPFD block and first part of the second NewPFD block. Thread T1 is responsible for second part of the second NewPFD block and the whole third NewPFD block, and so on.

are processed on the CPU side and the other queries would be transferred to the GPU for query processing.

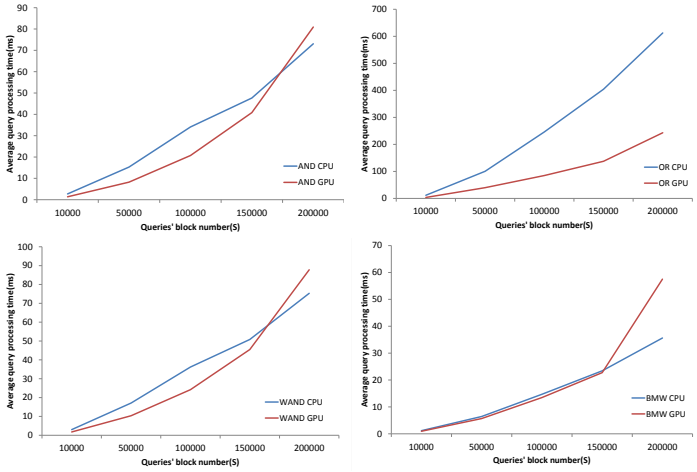

Fig. 3. The average query processing time on different queries' block number S (horizontal axis) on CPU and GPU for AND, OR, WAND, and BMW queries.

Before we introduce the CPU-GPU cooperative algorithm, we first do some experiments about the relation between the queries' posting length and query processing time about our GPU algorithm. Figure 3 shows the average query processing time for different queries' posting block number S with AND, OR, WAND, and BMW queries (both with $P = 64$ threads per thread block and top-

$K = 10$, other parameters have similar results). We can see that CPU algorithm is more effective than GPU algorithm when query block number S increases(except for OR queries).

The reason is that: as the document-based index partition is a simple partitioning method and the distribution of docIDs is clustered, long posting lists result in imbalanced lengths of sub-posting lists. A thread responsible for processing longer sub-posting lists will spend a greater amount of time than other threads, which will be idle because of the synchronization step.

Therefore, we propose a *length-based distribution* (LBD) method to determine which queries to distribute to the CPU and GPU. From the figure 4, we can find that short posting lists queries (queries's posting lists block number less 50000) take up the majority of the query set, approximately 81%. The GPU will be responsible for processing queries involving short posting lists, comprising a majority of the queries. and the CPU will be responsible for processing the smaller number of queries containing longer posting lists. Specifically, the GPU processes the queries whose relevant posting lists have fewer than S *NewPFD* blocks in total, and we will experiment with varying the threshold S.

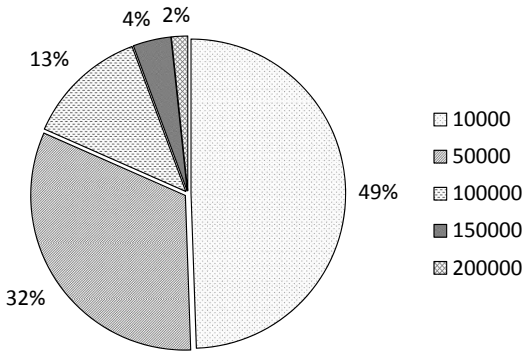

Fig. 4. The proportion of the queries' posting lists block number in TREC 2009 query set.

4 Experimental testing

4.1 Experimental setup

For our experiments, we use the TREC GOV2 [13] data set which consists of about 25 million documents crawled from web sites in the .gov domain during early 2004. The index data is composed of approximately 12 GB inverted index, another 12 GB frequency information index and a file about 97 MB storing document sizes. The docIDs are assigned according to the lexicographic order of

their URLs [11]. We use *NewPFD* to compress the index and also store a array about the greatest docIDs and the maximum "impact score" of each block. We choose the *RREC 2009* query set, which contains 32,255 queries, as our test query set. We carry out our experiments on a 2.60 GHz Intel(R) Xeon(R) E5-2630 CPU with 64 GB of memory and a NVIDIA GeForce GTX Titan graphics card with 6 GB global memory. The gcc version is 4.4.6, and the nvcc version is 6.5.12.

For the parameters in the ranking function BM25, we set $a = 1.2$ and $b = 0.75$. These parameters may affect the quality of the final top-k ranking, but will not significantly affect the throughput and response time of our method. We experiment with top-$k = 10$ unless we have special statement.

For the GPU list cache policy, we calculate that the number of the short lists (posting length 1,. . . ,64) takes up the majority of the whole compression lists; approximately 97.45% in GOV2 data set. As long lists need more transfer time and there are fewer long lists. Therefore we put the lists with more than 1 *NewPFD* block in the GPU list cache memory as our (static) cache policy.

4.2 Query processing time

Table 1 shows the average query processing time of the proposed GPU query parallel processing method (the non-cooperative version) as the number of threads GPU thread block and top-K vary (i.e., the P-value and top-K), for AND, OR, WAND and BMW query processing strategies. For comparison, we also include CPU a thread results in the bottom of Table 1.

Table 1. The comparison between average query processing time (ms) for the GPU as the number of threads per GPU thread block and top-K vary with the CPU a thread , for the AND, OR, WAND, and BMW queries. Values in bold show the best result in corresponding row.

P	AND			OR			WAND			BMW		
	top1	top10	top20	top1	top10	top20	top1	top10	top20	top1	top10	top20
32	8.07	8.08	8.14	45.57	45.88	45.97	8.29	10.22	11.18	3.54	4.90	6.49
64	**5.78**	**5.81**	**5.86**	30.26	30.32	30.37	**5.97**	**7.45**	**8.23**	**2.69**	**4.47**	**5.21**
128	6.06	6.08	6.13	**29.48**	**29.53**	**29.57**	6.19	7.86	8.73	2.96	**4.47**	**5.21**
256	7.53	7.56	7.61	34.38	34.42	34.49	7.55	8.97	9.65	4.14	6.51	7.60
320	9.33	9.35	9.39	39.86	39.91	39.95	9.24	11.71	13.03	4.91	7.51	8.71
512	12.24	12.28	12.36	49.02	49.08	49.21	12.05	15.23	16.93	6.17	9.14	10.53
CPU	12.70	12.82	12.93	65.77	66.08	66.15	15.11	15.25	15.34	8.82	8.96	9.03

We see that the GPU method results in improved query processing times for every query type, AND, OR, WAND, and BMW, with a average query processing time drop of up to 54.68%, 55.31%, 51.15%, and 50.11%, respectively (all when $P = 64$ and top-$K = 10$). We attribute this modest improvement to workload imbalance: the processing of long posting lists is time-consuming, resulting in long synchronization waiting times. The proposed CPU-GPU cooperative version aims to reduce this problem, by performing query processing on queries involving long posting lists on the CPU. and $P = 64$ threads per thread block is almost always the best P-value among those tested except for OR queries . In addition, average query processing time increases as top-K increases for every query type and different threads per thread block.

In the proposed GPU-based query processing algorithm, the query processing time splits into three major stages: (a) *Initialization*: The GPU threads identify which docIDs in posting lists belong to its assigned docID range, along with other initialization tasks; (b) *Scoring*: Traversing the posting lists to calculate the top-k results with the different strategies (AND, OR, WAND, and BMW); and (c) *Merging*: Going from the local top-k results to the final top-k results. Figure 5

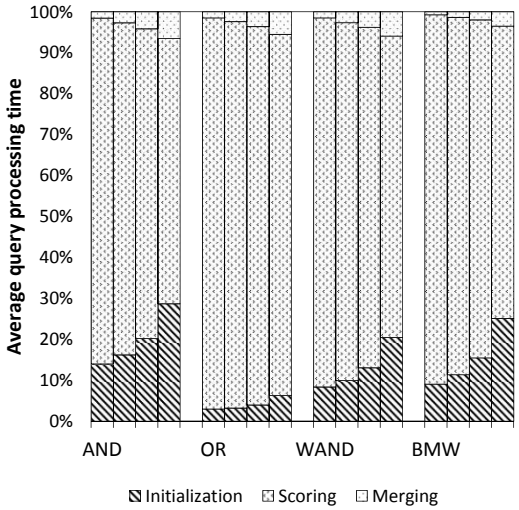

Fig. 5. The proportion of the average query processing time for the three stages in GPU-based query processing. We include measurements for $P \in \{32, 64, 128, 256\}$ threads per thread block, and for AND, OR, WAND, and BMW queries.

plots the proportion of the average query processing time of these three stages, computed using 1000 random user queries. We see (a) the top-k scoring takes up most of the GPU query processing time and (b) the proportion of time spent on scoring decreases as P increases, while the proportion of initialization and

merging time increases as P increases. This can explain the unimodal behavior seen in Table 1.

4.3 CPU-GPU cooperative version

We compare the performance of the proposed LBD method (introduced in Section 3.2) with the GPU and CPU-only methods, and with a *simple distribution* (SD) method, which randomly allocates half of the queries to CPU and the other half to GPU. In order to achieve the best query processing throughput with our LBD method, we carry out the experiments by varying the threshold S . Figure 6 shows the average query processing time with different threshold S for AND, OR, WAND, and BMW queries. We see that (a) threshold $S = 30000$ results in the best performance among those tested in every case, and (b) $P = 64$ threads per thread block is almost always the best P-value among those tested.

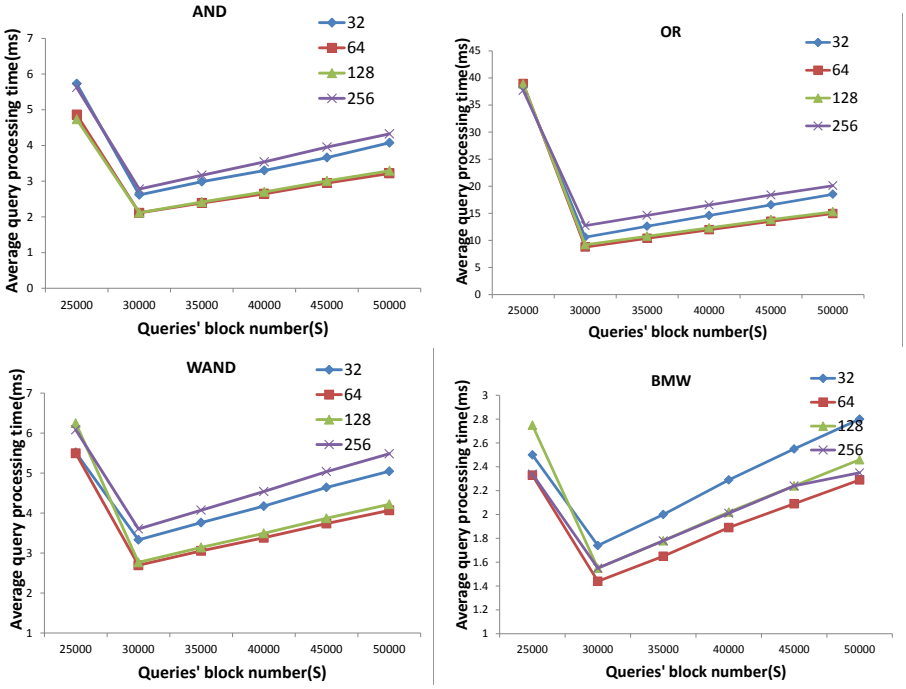

Fig. 6. The average query processing time for different threshold S (horizontal axis) in LBD. We include measurements for $P \in \{32, 64, 128, 256\}$ threads per thread block, and for AND, OR, WAND, and BMW queries.

Table 2 tabulates the average query processing time of SD and LBD, with a varying number of threads per thread block P (all when top-K = 10), and for AND, OR, WAND, and BMW queries. We list the best results among the

tested threshold value (S) as the LDB results. We see (a) that LBD outperforms SD in every case, and (b) LBD algorithm shows a greater improvement in query processing time, particularly in the case of BMW queries.

Table 2. Average query processing time (ms) for LBD and SD, and the difference Δ as a percentage of the SD time. Values in bold highlight the best observed average query processing time.

Method	32			64			128			256		
	SD	LBD	Δ	SD	LBD	Δ	SD	LBD	Δ	SD	LBD	Δ
AND	5.36	2.61	51.31%	4.2	**2.11**	49.76%	4.03	2.12	47.39%	5.22	2.78	46.74%
OR	23.2	10.6	54.31%	17.54	**8.76**	50.06%	16.87	9.19	45.52%	20.8	12.71	38.89%
WAND	6.32	3.33	47.31%	4.85	**2.69**	44.54%	4.84	2.77	42.77%	6.15	3.6	41.46%
BMW	3.79	1.74	54.09%	3.82	**1.44**	62.30%	3.85	1.55	59.74%	4.19	1.77	57.76%

4.4 Extensions

In this section we give some extensions of our GPU-based algorithm. we will test our algorithm on multi-GPUs cluster. In the experiment setting, we distribute query batches across four GPUs. Table 3 tabulates the average query processing time of the one GPU and four GPUs algorithm as the number of threads GPU thread block (all top-$K = 10$). Interestingly, $P = 128$ threads per thread block is the best P-value among those tested.

Table 3. Average query processing time(ms) of the GPU-based algorithm on four GPUs and one GPU as the number of threads per GPU thread block, and the difference Δ as a percentage on one GPU time, and for the AND, OR, WAND and BMW queries. Values in bold highlight the best observed average query processing time.

Method	64			128			256		
	1 GPU	4 GPU	Δ	1 GPU	4 GPU	Δ	1 GPU	4 GPU	Δ
AND	5.81	1.51	74.01%	6.08	**1.38**	77.30%	7.56	1.65	78.17%
OR	30.32	5.67	81.30%	29.53	**5.38**	81.78%	34.42	6.50	81.12%
WAND	7.45	1.75	76.51%	7.86	**1.63**	79.26%	8.97	1.98	77.93%
BMW	4.47	0.94	78.97%	4.47	**0.93**	79.19%	6.51	1.12	82.80%

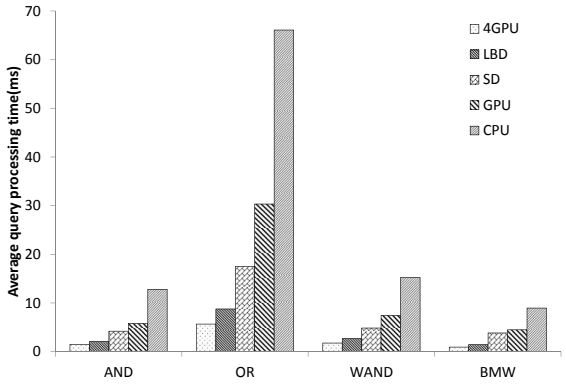

Fig. 7. The comparison between average query processing time (ms) for GPU-only, CPU-GPU cooperative (LBD and SD) and multi-GPUs(4 GPUs) ($P = 64$ and top-K = 10) with the CPU time. and for AND, OR, WAND, and BMW queries.

Figure 7 compares the average query processing time for the two CPU-GPU cooperative query processing methods (LBD and SD, both with $P = 64$ threads per thread block) along with the GPU non-cooperative , 4 GPUs(all when P=64) and CPU-only query processing methods. We see that utilizing a GPU and multi-GPUs for query processing can result in performance improvements for AND, OR, WAND, and BMW queries. Of the inspected methods, the LBD CPU-GPU cooperative method minimized query processing time, particularly in the case of BMW queries.

5 Conclusion and Future Work

In this paper, we propose two GPU-based query processing methods: one where the GPU performs parallel query processing for queries, and a CPU-GPU co-operative version where queries are simultaneously processed by both the CPU and GPU. We further develop a method for deciding which queries are processed by the CPU and GPU based on the lengths of the posting lists relevant to a query. In addition, we also evaluate the parallel query processing algorithm on mult-GPUs. Experiments indicate the CPU-GPU cooperative version results in around a 84% drop in average query processing time on one GPU and the multi-GPUs results can achieve 89% drop in average processing time.

We make the following suggestions on how to build upon this work:

- The GPU list cache policy could be optimized for the LBD method, e.g., by designing a dynamic caching algorithm which determines which posting lists are more likely to be needed by the GPU.
- We have not incorporated early termination in this work, which would reduce the time spent on top-k ranking. We have also not incorporated CPU-side

parallelism in this work, which could allow the CPU to process a heavier workload.

Acknowledgment

This work is partially supported by NSF of China (grant numbers: 61373018, 61602266 11550110491), Science and Technology Development Plan of Tianjin (17JCYBJC15300,16JCYBJC41900) and the Fundamental Research Funds for the Central Universities (Grant number: 65141020).

References

1. Agrawal, S.R., Pistol, V., Pang, V., Tran, J., Tarjan, D., Lebeck, A.R.: Rhythm: Harnessing data parallel hardware for server workloads. In: Proc. ASPLOS. pp. 19–84 (2014)
2. Ao, N., Zhang, F., Wu, D., Stones, D.S., Wang, G., Liu, X., Liu, J., Lin, S.: Efficient parallel lists intersection and index compression algorithms using graphics processing units. Proc. VLDB Endowment 4, 470–481 (2011)
3. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.Y.: Efficient query evaluation using a two-level retrieval process. In: Proc. CIKM. pp. 426–434 (2003)
4. Ding, S., Attenberg, J., Baeza-Yates, R., Suel, T.: Batch query processing for web search engines. In: Proc. WSDM. pp. 137–146 (2011)
5. Ding, S., He, J., Yan, H., Suel, T.: Using graphics processors for high performance IR query processing. In: Proc. WWW. pp. 421–430 (2009)
6. Ding, S., Suel, T.: Faster top-k document retrieval using block-max indexes. In: Proc. SIGIR. pp. 993–1002 (2011)
7. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N.K., Luo, Q., Sander, P.V.: GPUQP: query co-processing using graphics processors. In: Proc. SIGMOD. pp. 1061–1063 (2007)
8. NVIDIA: NVIDIA CUDA C programming guide (2015)
9. Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M.M., Gatford, M.: Okapi at TREC-3. NIST special publication p. 109 (1995)
10. Rojas, O., Costa, V.G., Marín, M.: Efficient parallel block-max WAND algorithm. In: Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings. pp. 394–405 (2013)
11. Silvestri, F.: Sorting out the document identifier assignment problem. In: Advances in Information Retrieval, 29th European Conference on IR Research, ECIR 2007, Rome, Italy, April 2-5, 2007, Proceedings. pp. 101–112 (2007)
12. Tatikonda, S., Cambazoglu, B.B., Junqueira, F.P.: Posting list intersection on multicore architectures. In: Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011. pp. 963–972 (2011)
13. Voorhees, E.M.: Overview of TREC 2003. In: Proc. TREC. pp. 1–13 (2003)
14. Wu, D., Zhang, F., Ao, N., Wang, G., Liu, X., Liu, J.: Efficient lists intersection by CPU-GPU cooperative computing. In: Proc. IPDPSW. pp. 1–8 (2010)
15. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. WWW. pp. 401–410 (2009)

16. Zhang, F., Wu, D., Ao, N., Wang, G., Liu, X., Liu, J.: Fast lists intersection with Bloom filter using graphics processing units. In: Proc. SAC. pp. 825–826 (2011)
17. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: Proc. WWW. pp. 387–396 (2008)
18. Zhang, S., Zhang, C., You, Z., Zheng, R., Xu, B.: Asynchronous stochastic gradient descent for DNN training. In: IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013. pp. 6660–6663 (2013)