Efficient Lists Intersection by CPU-GPU Cooperative Computing

Di Wu, Fan Zhang, Naiyong Ao, Gang Wang, Xiaoguang Liu, Jing Liu
Nankai-Baidu Joint Lab, College of Information Technical Science, Nankai University
Weijin Road 94, Tianjin, 300071, China
Email: {wakensky, zhangfan555}@gmail.com, {aonaiyong, wgzwp}@163.com, liuxg74@yahoo.com.cn, jingliu@mail.nankai.edu.cn

***Abstract*—Lists intersection is an important operation in modern web search engines. Many prior studies have focused on the single-core or multi-core CPU platform or many-core GPU. In this paper, we propose a CPU-GPU cooperative model that can integrate the computing power of CPU and GPU to perform lists intersection more efficiently. In the so-called synchronous mode, queries are grouped into batches and processed by GPU for high throughput. We design a query-parallel GPU algorithm based on an element-thread mapping strategy for load balancing. In the traditional asynchronous model, queries are processed one-by-one by CPU or GPU to gain perfect response time. We design an online scheduling algorithm to determine whether CPU or GPU processes the query faster. Regression analysis on a huge number of experimental results concludes a regression formula as the scheduling metric. We perform exhaustive experiments on our new approaches. Experimental results on the TREC Gov and Baidu datasets show that our approaches can improve the performance of the lists intersection significantly.**

I. Introduction

Trillions of user queries are being processed by Web search engines every day. Results quality and efficiency are the two most important criterion to evaluate a search engine. In this paper, we discuss the latter one. A lot of researches have studied on how to improve the efficiency of document retrieval in the search engines, such as caching, inverted index compression, early termination and massive parallelism. We focus on optimizing an important operation in the process of queries - intersection of k ordered lists.

To better understand this operation, we first look at the most fundamental data structure for indexing and query processing on text, the inverted index. An inverted index consists of many inverted lists containing a sorted list of document identifier (docid) and additional information such as term frequencies and positions of occurrences. An inverted list is associated with a term, that is, any document containing the term will appear in the term's corresponding inverted list. When a user query is submitted to the search engine, intersection operation is performed among the inverted lists of each term appearing in the query. It is a default operation when multi-term query processing, because users usually want to get the documents which contain all the query terms. Other operations such as union and difference are also supported in the modern search engine, but will not be addressed in this paper.

Lists intersection has attracted a significant attention [1], [2], [3], [4], [5], [6], [7]. Prior studies mainly focus on lists intersection on single-core or multi-core CPU platforms. Modern graphics processing units (GPUs) give a new way to solve the problem. Putting the operations on GPU has two advantages: firstly, the massive on-chip parallelism of GPU may greatly reduce the processing time of lists intersection; secondly, a great part of work on CPU is offloaded to GPU. In this paper, we will show how GPU could help search engine process queries efficiently.

Modern search engines are typically based on an asynchronous message passing model in which each newly arriving query is serviced by an independent thread. This strategy behaves well when load is light, because every query will be serviced by some thread immediately and the response time is usually short. Bonacic et al [8] found that heavy load can be very detrimental to overall performance, and proposes a so-called synchronous mode to solve this problem. Under light load, the system works in the traditional asynchronous mode while heavy load will trigger the synchronous mode. In synchronous mode, all active threads are blocked and a single thread takes control of query processing. Queries are grouped into batches, and then batches are processed one-by-one and each batch is processed by threads simultaneously. Motivated by this idea, we propose a CPU-GPU cooperative model described in detail in Section IV. Under light load, CPU is far from overloaded, response time is the only important thing we need to consider, so a heuristic algorithm is proposed to delivery a newly arriving query to the faster processor (CPU or GPU). Under heavy load, synchronous mode is trigged, queries are dispatched to GPU in batches. This approach maximizes the throughput (on the premise of acceptable response time) under heavy load and minimizes response time under light load. Moreover, we propose two algorithms for efficient lists intersection CPU processing. The second one, so-called query-parallel algorithm, must cooperate with CPU. According to acceptable response time and GPU's computing power, CPU determines the size of each batch dynamically. Our experimental results show that our methods can improve the performance of lists intersection greatly.

The remainder of this paper is organized as follows. The next section offers some technical background and discusses related work. Section III proposes the cooperative model, while Section IV discusses the GPU adaptive batching algorithm in more detail. Section V evaluates the efficiency of our

978-1-4244-6534-7/10/$26.00 ©2010 IEEE

algorithms and compares them with other approaches. Finally, Section VI provides concluding remarks and proposes future work.

II. Background And Related Work

In this section, we first discuss some related work about lists intersection. In subsection II-B, we discuss the parallel formulations for modern web search engines. In subsection II-C, we give some background knowledge on GPU programming. Finally, we summarize our contributions.

A. Lists Intersection

Researches on lists intersection problem go back to 1970s. Hwang and Lin [6], [7] studied intersection of two sorted lists. Demaine et al. [5] improved over this by proposing a faster method for computing the intersection of k sorted sets using an adaptive algorithm which repeatedly cycling through the sets in a round-robin fashion. We refer the reader to [1], [2], [3], [4], [5], [6], [7] for a more detailed summary on the serial algorithm of lists intersection.

There are already some literatures on parallel lists intersection algorithms on multi-core and many-core platforms. Tsirogiannis [9] studied lists intersection algorithms suitable for the characteristics of chip multiprocessors (CMP). They proposed a Dynamic Probes algorithm which takes advantage of multi-level cache hierarchies to reduce the overhead of random memory accesses. They also proposed a Quantile-based algorithm that reduces the cost of intersection by a load balancing algorithm. Ding et al. [10] parallelized the lists intersection in their GPU search engine. They used the Parallel Merge Find algorithm to compute the intersection of k lists. This algorithm splits the lists into smaller segments and merges these smaller segments respectively.

B. Parallel Formulations for Web Search Engines

There are two typical types of parallel strategies for query processing: inter-query formulation and intra-query formulation while the former explores the parallelism among queries, and the latter explores the parallelism within a query. The intra-query formulation distributes the partitioned inverted index to a cluster of computers, and then each machine calculates its local results simultaneously and finally the results are merged at a single machine. This approach is the most common parallel formulation used in modern search engines and has been well studied [11], [12].

Bonaicic et al. [9] found that sudden load peak can be very detrimental to overall performance of search engines. A so-called synchronous mode was proposed to solve the problem. The system dynamically switches its working mode in accordance with load observed. Light load triggers the asynchronous mode in which each query is serviced by a unique thread. Heavy load triggers the synchronous mode in which queries are grouped, then groups are processed in serial and each group is processed by multiple threads simultaneously. This is a typical inter-query formulation. It is very similar to our architecture will be discussed in III.

Tatikonda et al. [13] also studied the query throughput and query latency of inter or intra methods on multi-core processors.

C. GPU Programming

In this section, we give a brief overview of current GPU programming techniques used in our work. For more details about GPU programming we refer the readers to [14], [15]. At the hardware level, a GPU is a collection of multiprocessors each with several processing elements. For instance, a Nvidia GeForce GTX 280 has 30 multiprocessors each with 8 scalar processor (SP). This many-core architecture and very low thread scheduling overhead enable GPU to support up to thousands of threads efficiently. So programs should invoke enough threads to take full use of GPU's computing power. Multiprocessors are based on SIMD architecture, that is, scalar processors in a multiprocessor execute the same instruction with different data synchronously.

The programming model we use is Nvidia's Compute Unified Device Architecture (CUDA). For programmers, CUDA is a programming interface to the parallel architecture of Nvidia GPUs for general purpose computing. A CUDA program consists of a collection of threads running in parallel. Threads are organized into *thread blocks*. Each block is divided into *warps* each containing 32 threads. Warp is the basic scheduling and executing unit of a program in multiprocessor. So thread divergency in warp has to be avoided, otherwise great performance degradation will occur. Another important feature of CUDA for programming is memory hierarchy. We should try to confine most of data accesses to *shared memory* rather than *global memory*, because the former is orders of magnitude faster than the latter.

In recent years, GPU has rapidly become one of the most important parallel infrastructures. Tianhe-1, the fastest supercomputer in China (the fifth position in the newest top 500 list [16]) is a GPU-enabled cluster though it uses AMD's Radeon HD 4870x2. Another supercomputer with beyond petaflops peak performance is also deployed in Institute of Process Engineering, Chinese Academy of Sciences [17]. It is based on CUDA technology. In academia world, hundreds of papers about CUDA and other GPU technologies have been published. So we try to introduce GPU into list intersection problem.

D. Our Contributions

In this paper, we study how to improve the performance of lists intersection using GPU. The contributions of this paper include:

- Motivated by [8], we present a CPU-GPU cooperative model which can dynamically switch between the asynchronous mode and the synchronous mode. In both modes, GPU plays an important role in efficient lists intersection.
- Under light load, the system works in asynchronous mode. We minimize query response time in the aid of

GPU. Heuristic strategies are designed to decide whether the current query should be processed by GPU or CPU.
- Under heavy load, the system works in synchronous mode. Queries are grouped in batches and processed on GPU. We propose a query-parallel algorithm to balance load between thread blocks, therefore process a batch efficiently.
- We evaluate our approaches on TREC GOV and Baidu data sets, and compare them with the two traditional models. The results show that our algorithms can significantly improve performance of lists intersection on real world data sets.
- We also show that our algorithms have good scalability.

III. Cooperative Model

Our goal is to improve the performance of lists intersection in real web search engines. In practice, the load of a web search engine is changing every time. The system throughput and response time could be impacted seriously when system load fluctuates violently. In the asynchronous mode in which each newly arriving query is serviced by an independent thread, some queries will be blocked (therefore gain bad response time) by previous queries under heavy load. To deal with this problem, we propose a cooperative model that can dynamically switch between working modes according to load observed. There are two working modes in our model:

Asynchronous mode: If load is light, in other words, CPU is far from overloaded, we need not worry about throughput or succeed queries being blocked by previous queries. So the system works in asynchronous mode. Every newly arriving query will be processed immediately. Before processing the query, we determine the query should be processed in which processor - CPU or GPU. There are several points in determining the processor: GPU is good at processing large amount of data simultaneously, so if a query contains long lists, it is more likely that GPU can process it faster. While if a query contains only short lists, it can not make full use of GPU's computing power. Besides, we can not omit GPU kernel invoking time and data transferring time between CPU and GPU (uploading the query from CPU to GPU and downloading results from GPU to CPU). We must take all of these factors into account to make proper decision. In Section V, we give a compressive analysis on query scheduling.

Synchronous mode: Under heavy load, the system works in synchronous mode, queries are grouped in batches and processed by GPU. In the next section, we will introduce two algorithms that try to process batches of queries efficiently on GPU.

IV. GPU Batching Algorithms

We developed GPU batching algorithms for efficient lists intersection. The common key idea of the two algorithms is to pump enough queries to GPU at a time to make full use of SPs in GPU. We will first give the basic idea of our two algorithms: query-partition algorithm and query-parallel algorithm and discuss the latter in detail.

In CUDA platform, threads are grouped in thread blocks and synchronization between threads in different blocks is expensive. So an intuitive idea is assigning each query to a unique block. We call this approach the *query-partition algorithm*, $PART$ in short. Queries may be quite different in lists' length, therefore may have huge diversity of computation complexity. This is the root cause of load imbalance in PART, which appears as some multiprocessors idling while the other multiprocessors still busy on their (big) queries. In our previous work [18], a cpu-aided approach was presented to solve this problem. CPU takes queries' computation amount into account when batching. Queries with similar size are grouped into the same batch. However, this approach shows its effectiveness only for very large queries set, for example, a set containing 33337 queries [18]. This requirement is obviously unpractical. Such a queries set is too big to guarantee reasonable response time. Moreover, size based batching introduces nontrivial extra burden on CPU.

So we developed a *query-parallel algorithm* ($PARA$ in short) that may process a query by several blocks cooperatively according to its size instead of assigning each query to a single block. We will discuss PARA in three aspects: CPU preprocessing, GPU processing and data transferring.

CPU Preprocessing: When a batch of N queries are ready, CPU will first sort lists in each query by increasing length, and send the batch to GPU. The size N of a batch is determined by total computation load of queries in the batch. Roughly, the computational complexity of a query is estimated by the number of elements in the its shortest list. The batch size N is limited by a *computation threshold*. In Section V, we will discuss in detail how to estimated a query's computation load precisely. As a result, compared with PART, PARA can control the total computation load delivered to GPU and load assigned to each block more precisely.

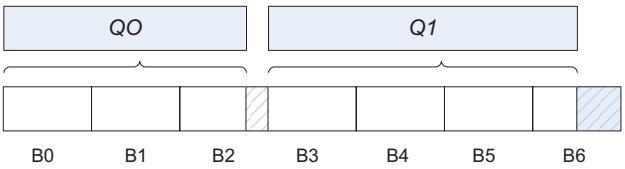

Fig. 1. Element-thread mapping.

GPU Processing: Unlike PART's query-block mapping, PARA adopts element-thread mapping. That is, for each query, PARA assigns each element in the shortest list to a unique thread. Therefore, a query may be undertaken by multiple blocks. We will use Q_N to represent the nth query, and B_N represent the nth thread block. For example, as Fig. 1 shows, the length of shortest list of Q_0 is 750, while the block size is 256. Q_0 will be assigned to blocks B_0, B_1 and B_2. The threads in B_0 and B_1 are fully taken advantage of. 238 of all 256 threads in B_2 are active, while the remaining threads in this block are left idle. The next query in the batch will be assigned to the following blocks. We can see that, compared

with PART, PARA is more likely to distribute computation load evenly.

After being assigned an element e, a thread performs binary search in the remaining longer lists to determine whether e is a common element (docid) of all lists. If it is, its corresponding position in a 0-1 array is set to 1, otherwise the position is set to 0. That is, this array identifies which docids belong to the result list (the intersection of all lists). Each query has its own 0-1 array that has the same length as the shortest list. Then a scan (prefix sum) is performed on the 0-1 array to calculate elements' positions in the result. Note that in PARA, each thread is responsible for an element, so a standard scan algorithm is used. In PART, each query is processed by a single block and a thread may be responsible for several elements, thus each block executes a sectionalized scan algorithm [18].

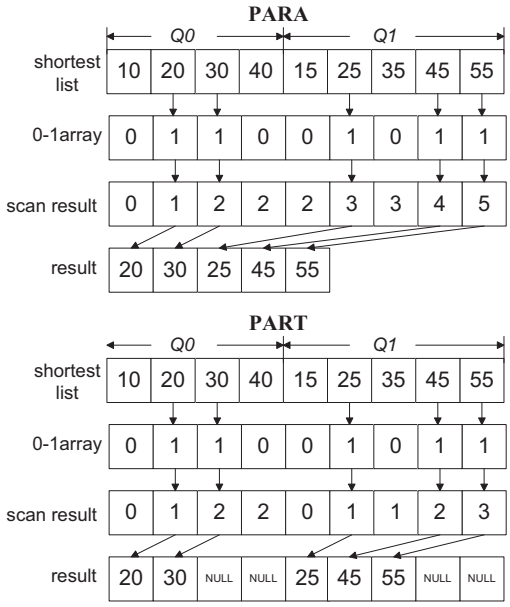

Fig. 2. GPU kernel scheme

Finally a compaction is performed to gather elements to the intersection list according to the scan result. In PART, since each block performs scan and compaction for a query independently, the intersection list occupies the same memory space as the shortest list. However, since the standard scan algorithm is used in PARA, all intersection lists are stored consecutively rather than separately. So we extract the positions of the intersection lists in memory from the scan result to help CPU fetch them.

Data Transferring: Since GPUs used in tests can hold the two test data sets, we upload the whole data set to GPU when initialization. In a large-scale search engine, we could put those inverted lists which are most frequently accessed in GPU memory. The transferring of data set is needed only once during the entire procedure, so we do not take this transfer time into account.

For each batch, necessary information, such as terms of each query are uploaded to GPU before processing and the intersection lists are downloaded to CPU after processing. We study data transfer during running time carefully, since data transfer is important to performance. Note that in PARA, we also download the positions array. In PART, some useless memory words are downloaded.

V. Experiments

A. Experimental Setup

Datasets: We used the following two data sets: the data set provided Baidu Inc. and TREC GOV data set. The GOV data set consists about 1.25 million pages crawled from web sites in the gov domain in 2002. The Baidu dataset contains 1.5 million web pages in English. The length distribution of inverted lists in GOV data set is illustrated in TABLE I. Baidu data set has similar characteristics.

TABLE I
Length distribution of inverted lists in GOV data set.

(0,1K]	(1K,5K]	(5K,10K]	(10K,50K]	(50K,100K]	(100K,∞)
64.32%	19.35%	5.93%	7.93%	1.52%	0.95%

Query Sets: For testing the Baidu dataset, we used a query set also provided by Baidu Inc. which contains $33,337$ queries. For testing GOV dataset, we used the Terabyte 2006 query set which contains $100,000$ queries.

Environment: All experiments were performed on a AMD Phenom II X4 945 machine with 2GB*2 DDR3 1333 memory. The GPU is Nvidia Tesla C1060, which has 30 streaming multiprocessors, 240 scalar processor cores in total. The frequency of processor cores is 1.3GHz. The card is equipped with 4GB DDR3 memory of bandwidth 102GB/s. The GPU programming platform is CUDA version 2.3.

B. Experimental Results

1) PARA on GOV dataset: We set the computation threshold according to the factors below:

i) The computing power of GPU. The more scalar processors and the higher bandwidth GPU has, the higher threshold, vice versa. Therefore the computation threshold formula contains two coefficients indicating GPU's computing power.
ii) Required system throughput. Higher computation threshold means bigger batch which makes full use of GPU's potential computing power.
iii) Required response time. Lower computation threshold means smaller batch which generally implies faster preprocessing, shorter data transferring means less kernel time and less transfer time.

We use the formula $(threshold = S * T * B)$ to determine the computation threshold. The first two factors denote the number of SMs and the number of threads per block respectively, and B denotes the number of GPU thread blocks we want a SM deals with. The product of the first two factors

denote the computing power of the GPU card, and the last factor is the computation complexity we want to put on one GPU SM. For example, if we set computation threshold as $30*256*128$, we mean that NVidia Tesla C1060 has 30 SMs, and we arrange 256 threads in every block. The last factor 128 denotes that we want put 128 thread blocks on every SM. Since S is a hardware parameter and we fixed T in our tests, B in fact can be regarded as the computation threshold.

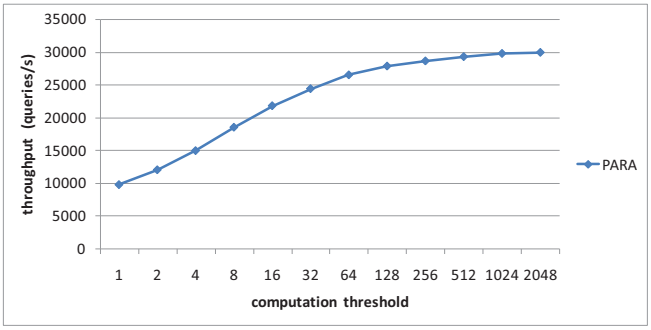

Fig. 3. The impact of computation threshold on system throughput.

As Fig. 3 illustrates, PARA reaches the peak throughput of about $30,000$ queries per second on the GOV dataset on a single Tesla C1060 (T is set to 256, same as follows). When B goes over 128, the growth rate of throughput slows down which implies that GPU is about fully loaded so that more blocks make no sense.

Besides throughput, we also record response time. In synchronous mode, since queries in a batch are processed together rather than one by one, their response time is the sum of CPU preprocessing time, GPU kernel time and transfer time. Fig. 4 shows a obvious uptrend of response time as B increases. However, when B is less than 256, the response time increases very slow as B increases and is no more than 10 ms. The response time increases sharply after B goes over 256.

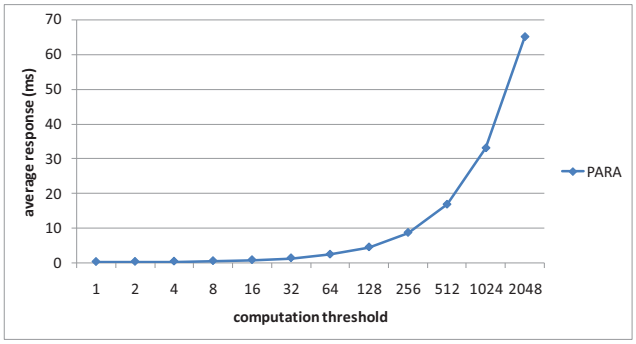

Fig. 4. The impact of computation threshold on response time.

Considering Fig. 3 and Fig. 4 jointly, we can select reasonable computation thresholds with acceptable response time and throughput. Certainly, we can change the threshold dynamically according to system load. If load is heavy, we could increase the threshold to obtain higher throughput. When load is light, lower threshold can be used to gain better response time.

2) PART vs. PARA on Baidu dataset: We test the two batching algorithms using Baidu dataset. Note that the batch size is not fixed in PARA, because we assemble a batch according to the computational complexity instead of the number of queries. So we compare the two algorithms under the similar average batch size. For example, PARA divides Baidu dataset into 65 batches in PARA when B is 256, so the average batch size is about 513 queries. To be fair, we set the batch size to 513 queries for PART.

Our prior work [18] shows that the load imbalance has a significant negative impact on PART. To balance the load, CPU has to perform a "semi-sort" on each batch, that is, assign queries to buckets according to the lengths of their shortest lists. Each bucket corresponds to a length range. Then buckets are arranged in ascending order. We can see that this method tries to assign queries with similar computational complexity to adjacent blocks to gain good load balance. However, this method works only for very large batches. If batch is small, the lengths of the shortest lists are sparse, therefore adjacent blocks most likely are assigned very different computation load. Unfortunately, we must use relatively small batches for reasonable response time.

As Fig. 5 shows, there exists a dramatic performance gap between PART and PARA. The reason is as mentioned above, PART most likely induces load imbalance. Moreover, PART has higher data transferring overhead, because intersection lists occupy extra space in PART.

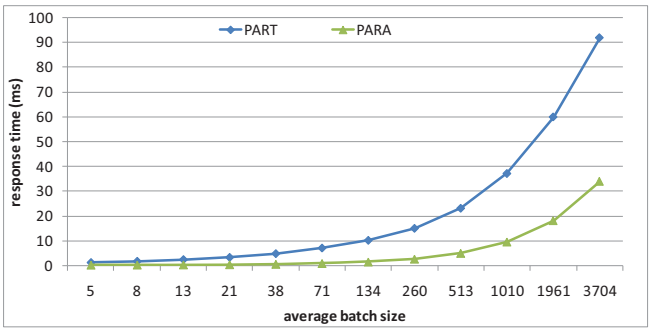

Fig. 5. Response time under different average batch sizes.

Fig. 6 shows the throughput of PART and PARA. PARA has a significant advantage over PART. When batch size goes over 513, PART maintains rising trend, while the growth pace of PARA slows down obviously. The reason is that larger batches mean better load balance in PART, and make no sense to PARA because GPU is saturated.

3) Fluctuation of response time on Baidu dataset: Response time fluctuation is bad to search engine. Violent fluctuations mean horrible user experience. A slow batch not only means bad current response time, but also affects succeed queries seriously. Moreover, it will be difficult for administrator to predict the system performance. Fig. 7 shows that the response time goes steady in PARA, while PART leads to

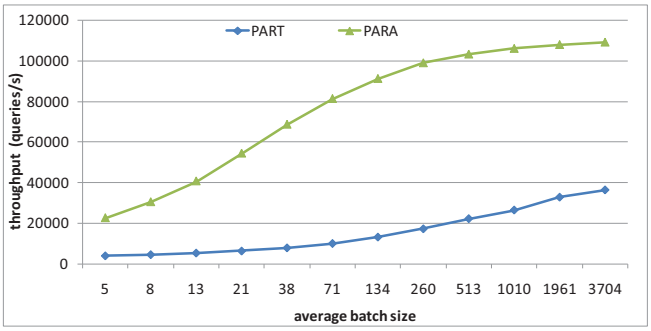

Fig. 6. Throughput under different average batch sizes.

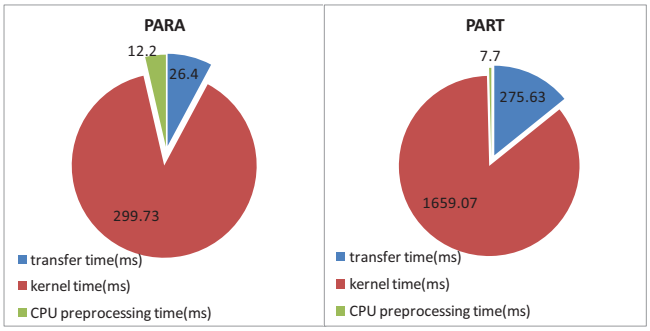

Fig. 8. Proportion of each step.

violent fluctuations. This result is expected, PARA assembles batches according to computational complexity, so all batches have almost the same computation load. PART groups a fixed number of queries into a batch mechanically, apparently can not guarantee consistent computational complexity.

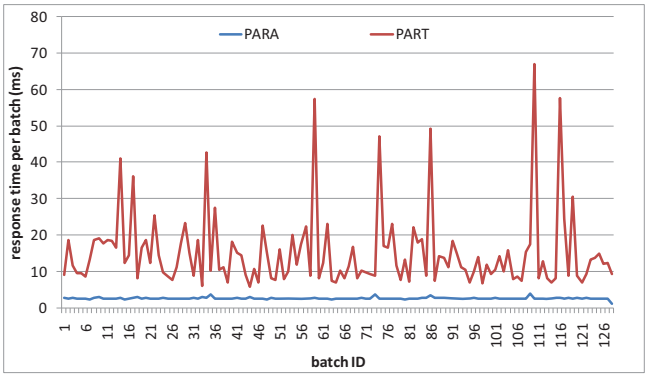

Fig. 7. Response time fluctuation.

4) The time proportion of two algorithms on Baidu dataset: Fig 8 shows the ratio of transfer time, kernel time and CPU preprocessing time of each step to the total execution time per batch respectively. The transfer time here is time of transferring necessary information, such as terms of each query before processing and the intersection results after processing. The batch size is 260. Because of the advantage of continuous storage, PARA decreases the size of the intersection lists needed to be downloaded to main memory dramatically. The proportion of effective kernel time in PARA is higher than that in PART which means that work is offloaded from CPU to GPU more effectively.

PARA sorts lists in every query by CPU, in order to avoid multiple GPU blocks sort the same query. In addition, for each query, CPU makes a unique copy for each block to avoid global memory access conflicts. These factors leads to a bit more CPU time in PARA. However, compared with kernel time, this extra CPU time is negligible.

5) Accelerating by multi-GPU: To improve the system throughput further, we deploy two NVidia C1060 cards on our test machine. Two CPU child threads are created; each of them controls one GPU card. We use dynamic load balancing policy to distribute the calculation tasks to the two cards. Threads scramble for queries in the query queue. Once a thread fetches enough queries, it will pump the batch to the GPU card it controls.

TABLE II shows the computational load each card undertakes on Baidu Dataset, where K stands for kernel time in ms; t stands for transfer time in ms; C stands for CPU preprocess time in ms; QN stands for the number of queries per batch. We can see that the two cards get similar number of queries though they are plugged in slots with different effective bandwidth. The kernel time and CPU processing time are close too. This means that load is nearly optimally balanced. When computation threshold is high, there is a growing proportion of interaction overhead between the two threads in the total running time. For example, if B is larger than 256, CPU processing time spent by each child thread is much more than the CPU processing time spent in single card mode. However, from Fig. 3 we can see that, B higher than 256 may be useless to most systems, since throughput grows slow and response time increases sharply. Therefore, the scheme of dynamic load balancing for double GPUs is suitable for most systems which are response time sensitive.

In each batch, GPU card0 and GPU card1 deal with similar amount of queries. Batch size on each card is similar to that in the case of single card. Since two cards work concurrently, the system throughput is nearly doubled.

6) Query scheduling under asynchronous mode: If system load is quite light, the system works in asynchronous mode. In this case, it may not be a good choice to process queries in batches by GPU because both CPU and GPU can offer enough throughput and processing queries by CPU may lead to better response time. This mode is also helpful to energy saving by letting GPU idle. We designed a CPU-GPU cooperative model, based on the statistics approach.

When queries arrive slowly (for example, average 200 queries per second) and the requirement of response time is still strict, we can set computation threshold to 0, which means that batch size is just 1. Many queries have little computational complexity, so they can be processed on CPU much faster than GPU because data transferring overhead is avoided. However, queries with high complexity still should be scheduled to GPU.

TABLE II
PERFORMANCE COMPARISON BETWEEN SINGLE CARD AND DOUBLE CARDS.

B(computation threshold)	single card			card0 (in a PCI-E x16 slot)				card1 (in a PCI-E x8 slot)			
	K	t	C	K	t	C	QN	K	t	C	QN
32	341.45	55.86	12.6	179.736810	29.1	8.607	15381	206.665931	34.9	10.279	17956
64	314.75	38.63	12.1	180.959591	20.4	10.29	17392	173.006625	21.8	9.630	15945
128	299.15	25.85	11.93	172.450409	14.8	8.14	16911	165.302573	15.5	7.7	16426
256	290.76	20.52	11.64	163.950166	12.07	11.4	16443	163.913291	13.13	11.35	16894
512	284.95	17.86	11.56	162.563118	10.46	38.9	17275	157.975945	13.21	38.6	16062
1024	281.95	15.22	11.68	160.941405	8.7	21.68	17591	156.653708	12.4	21.38	15746
2048	279.65	13.58	12.2	154.652893	11.47	75.9	16142	159.990445	14.5	75.85	17195

So, we need a criterion to determine where to deliver the single query. We deal with queries one by one on CPU and GPU separately. CPU response time and GPU response time for each query is recorded. We use the difference between CPU response time and GPU response time. If the response time difference is positive, GPU can process the query faster. Experimental results are under GOV dataset.

We use an efficient single-threaded CPU algorithm specially optimized for lists intersection, which is designed by LingJiang, Baidu Inc. In the algorithm, we equally divide the longer list into several segments. Every element in shorter list is searched in the longer list sequentially, while the step is the length of every segment. If the element being searched is greater than the last element of current segment in the longer list, we jump to the next segment. Otherwise, we must turn around to search this element in the current segment.

As histogram in Fig. 9 illustrates, CPU has advantage over GPU on most queries. However, the advantage is not significant. By contrast, as Fig. 10 shows, GPU is far superior to CPU in the queries whose computation complexity is high.

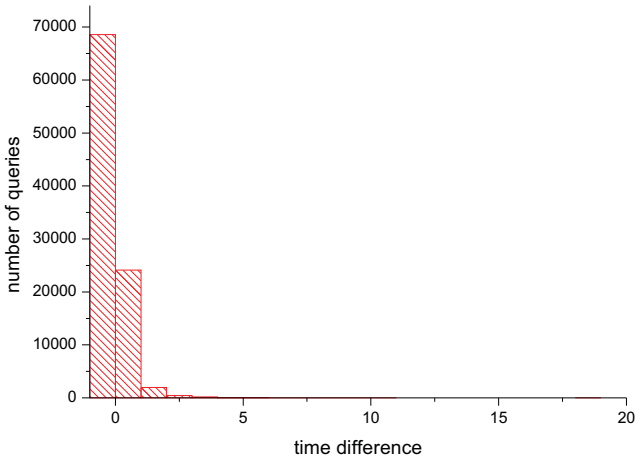

Fig. 9. The distribution of time difference.

We know the number of lists and the length of each list when CPU gets a query. However, the computational complexity is difficult to estimate precisely according to these parameters. The complexity of this problem is determined not only by the lengths of the lists but also the elements in the lists. Therefore, we introduce three metric to estimate the computational complexity.

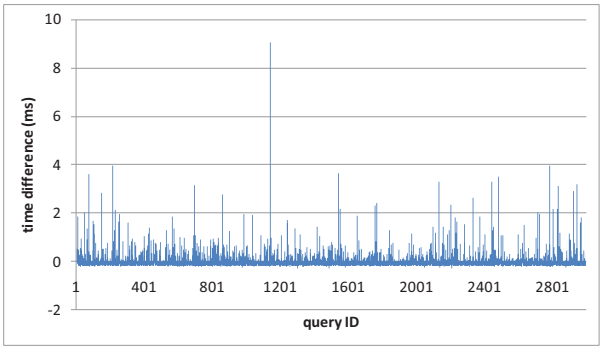

Fig. 10. Time difference in detail of first 3000 queries.

i) LOS: the length of the shortest list. We can see that the number of common docIDs is no more than LOS. The threads allocated for one query is determined by LOS. So LOS is an intuitive estimation of computational complexity.
ii) UBOC: the upper bound of the number of comparisons. $UBOC = LOS * (logL1 + logL2 + logL3 + \cdots)$, where LN is the length of Nth list (except the shortest list), as the computational complexity of binary search is logarithmic. This metric is more precise than LOS apparently.
iii) UBOCT: the upper bound of the number of comparisons per thread. This metric depicts the parallel running time well if the query is processed fully in parallel.

The scheduling algorithm boils down to the relationship between the time difference and each metrics. We adopt regression analysis here. If the p-value is smaller than 0.001, we can not reject the result of regression analysis at 1% significant level. R-square is the coefficient of determination, which is the proportion of variability in dataset that is accounted for by the statistical model [19].

As table III shows, on the same significant level, UBOC is accounted for 70.02% variability of time difference. It is an ideal metric reflecting the time difference on statistics. Therefore, we could regard the regression equation as the scheduling rule. We get the following regression formula:

TABLE III
REGRESSION ANALYSIS ON TIME DIFFERENCE AND SCHEDULING METRICS

metric	r-square	p-value	coefficient	Intercept
LOS	0.6312	< 0.0001	0.00001881	-0.13488
UBOC	0.7002	< 0.0001	3.784915E-7	-0.14164
UBOCT	0.0125	< 0.0001	0.00141	-0.04813

$$TimeDiff = -0.14164 + (3.7849E-7) * UBOC$$

When CPU gets a query under light load, it calculates UBOC first. Then $TimeDiff$ is calculated. If it is positive, CPU will delivery the query to GPU; otherwise, CPU will deal with the query by itself. Each data set has its unique features. For example, the distribution of inverted index lists is different among data sets. Therefore, we must generate different regression formulas for other data sets.

VI. Conclusion and Future work

In this paper, we propose a CPU-GPU cooperative model for efficient lists intersection in web search engines. Under light load, the system works in asynchronous mode. In this mode, queries are processed one-by-one by CPU or GPU. We design a scheduling algorithm to determine where the current query is to be delivered to. This algorithm is based on a regression formula concluded by a regression analysis on a lot of experimental results. On the other hand, if load is heavy, the system works in synchronous mode in which queries are sent to GPU in batches and processed by a query-parallel algorithm. Experimental results show that this element-thread mapping based algorithm balances load perfectly.

We believe that GPU will play an important role in actual large-scale distributed systems such as search engines in the future. However, a lot of work needs to do to make our algorithm more practical. Since the data set in real search engine systems may be much larger than our test set, effective compression algorithms are worth seriously studying. Streamed GPU algorithms for transfer overhead hiding are also valuable. Certainly, precise system load predicting is another important research direction.

VII. Acknowledgment

This paper is supported partly by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028), SRFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000).

We would like to thank Baidu Inc. to provide data set and other necessary materials. Many thanks to Guangjun Xie for his kindly help. Thank Caihong Qiu for helping us with statistics. Thank Ling Jiang for providing the efficient CPU algorithm.

References

[1] R. A. Baeza-Yates, "A Fast Set Intersection Algorithm for Sorted Sequences," in *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, CPM 2004*, Istanbul,Turkey, Jul 2004, pp. 400–408.

[2] R. A. Baeza-Yates and A. Salinger, "Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences," in *Proceedings of the 12th International Conference on String Processing and Information Retrieval, SPRIE 2005*, Buenos Aires, Argentina, Nov 2005, pp. 13–24.

[3] J. Barbay and C. Kenyon, "Adaptive intersection and t-threshold problems," in *SODA*, 2002, pp. 390–399.

[4] J. Barbay, A. López-Ortiz, and T. Lu, "Faster Adaptive Set Intersections for Text Searching," in *Proceedings of the 5th International Workshop on Experimental Algorithms, WEA 2006*, Cala Galdana, Menorca, Spain, May 2006, pp. 146–157.

[5] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *SODA*, 2000, pp. 743–752.

[6] F. K. Hwang and S. Lin, "Optimal Merging of 2 Elements with n Elements," *Acta Inf.*, vol. 1, pp. 145–158, 1971.

[7] ——, "A Simple Algorithm for Merging Two Disjoint Linearly-Ordered Sets," *SIAM J. Comput.*, vol. 1, pp. 31–39, 1972.

[8] C. Bonacic, C. García, M. Marín, M. Prieto, F. Tirado, and C. Vicente, "Improving Search Engines Performance on Multithreading Processors," in *Proceeding of 8th International Conference on High Performance Computing for Computational Science - VECPAR 2008,*, Toulouse, France, Jun 2008, pp. 201–213.

[9] D. Tsirogiannis, S. Guha, and N. Koudas, "Improving the Performance of List Intersection," *PVLDB*, vol. 2, no. 1, pp. 838–849, 2009.

[10] S. Ding, J. He, H. Yan, and T. Suel, "Using graphics processors for high-performance IR query processing," in *Proceedings of the 17th International Conference on World Wide Web, WWW 2008*, Beijing, China, Apr 2008, pp. 1213–1214.

[11] A. MacFarlane, J. A. McCann, and S. E. Robertson, "Parallel Search Using Partitioned Inverted Files," in *Proceedings of the 7th International Conference on String Processing and Information Retrieval, SPRIE 2000*, 2000, pp. 209–220.

[12] A. Moffat, W. Webber, and J. Zobel, "Load balancing for term-distributed parallel retrieval," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2006*, Seattle, Washington, USA, Aug 2006, pp. 348–355.

[13] S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras, "On efficient posting list intersection with multicore processors," in *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009*, Boston, MA, USA, Jul 2009, pp. 738–739.

[14] (2009, Oct) NVIDIA CUDA Compute Unified Device Architecture Programming Guide. [Online]. Available: http://developer.nvidia.com/cuda

[15] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008*, Austin, Texas, USA, Nov 2008, p. 31.

[16] Top 500. [Online]. Available: http://www.top500.org

[17] F. Chen, W. Ge, L. Guo, X. He, B. Li, J. Li, X. Li, X. Wang, and X. Yuan, "Multi-scale HPC system for multi-scale discrete simulationłDevelopment and application of a supercomputer with 1 Petaflops peak performance in single precision," *Particuology*, vol. 7, no. 4, pp. 332–335, Aug 2009.

[18] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang, "A Batched GPU Algorithm for Set Intersection," in *Proceedings of 2009 International Workshop on GPU Technologies and Applications (GPUTA 2009), in conjunction with the 10th International Symposium on Pervasive Systems, Algorithms, and Networks I-SPAN 2009*, Kaoshiung, Taiwan, Dec 2009, pp. 752–756.

[19] R. G. D. Steel and J. H. Torrie, *Principles and Procedures of Statistics*. New York: McGraw-Hill, 1960.