A Batched GPU Algorithm for Set Intersection

Di Wu, Fan Zhang, Naiyong Ao, Fang Wang, Xiaoguang Liu, Gang Wang
Nankai-Baidu Joint Lab, College of Information Technical Science, Nankai University
Weijin Road 94, Tianjin, 300071, China
Email: {wakensky,zhangfan555}@gmail.com, aonaiyong@163.com,
wangfang09@gmail.com, liuxg74@yahoo.com.cn, wgzwp@163.com

***Abstract*—Intersection of inverted lists is a frequently used operation in search engine systems. Efficient CPU and GPU intersection algorithms for large problem size are well studied. We propose an efficient GPU algorithm for high performance intersection of inverted index lists on CUDA platform. This algorithm feeds queries to GPU in batches, thus can take full advantage of GPU processor cores even if problem size is small. We also propose an input preprocessing method which alleviate load imbalance effectively. Our experimental results based on a real world test set show that the batched algorithm is much faster than the fastest CPU algorithm and plain GPU algorithm.**

I. Introduction

The Internet search market has undergone a booming expansion along with the rapid growth of Internet, which makes search engines faced with great performance challenges. To provide high throughput, search engine systems typically are built upon large-scale clusters. Nevertheless, per node workload is still quite heavy, especially CPU occupancy is high. Thus, in order to maximize overall throughout, we propose a new approach to offload the CPU workload to graphical processing units (GPUs).

In search engine systems, "list intersection" occupies a significant part of CPU time. A search query is composed of several keywords. Each keyword corresponds to an inverted index list which stores IDs of documents that contain this keyword. What search engine returns to user is just the intersection result of the inverted index lists involved.

The data set we used is provided by Baidu Inc. It is a real world test set instead of a synthetic one. It contains 33337 user search queries. Each query contains 2 to 8 keywords. Our aim is to offload all intersection calculation from CPU to GPU. However, our data set has a property: most of inverted index lists are short (contain less than 10,000 docIDs). That is to say, problem size is small generally. To the best of our knowledge, no previous work focuses on GPU algorithm for this type of input. However, we believe that small problem size is a common character of real search engine systems, because most of user queries contain multi-keyword. The intermediate result might become smaller after first one or two intersections.

Our contributions mainly deal with this property:

1) We design a batched GPU algorithm. The experimental results show that for small queries, this new algorithm is far superior to other GPU algorithms.
2) We design an input preprocessing method to get the best performance of our algorithm.
3) Tradeoff between throughout and response time is considered.

II. Related Work

List intersection is well studied in these years because of its importance in search engines. Some important results are published [1, 5, 6, 7]. But most of them perform well only for large lists.

Generally, document IDs in a list are sorted. So an intuitive GPU algorithm for intersection operation of two lists is: each thread fetches one element from the shorter list, then all threads do binary search in the longer list simultaneously and independently. However, if the two lists are both very large (say, both contain millions of elements), quite a lot compare operations are wasteful. Thus an improvement called partition is presented in [1]. The shorter list is divided into several segments, and the last element of each segment is searched in the longer list by a unique thread. Whether the representative element is found or not, its segment is confined to a subregion of the longer list. Fig. 1 illustrates this idea. After partition, each segment in the shorter list corresponds to a segment in the longer list. The number of compare operations decreases dramatically.

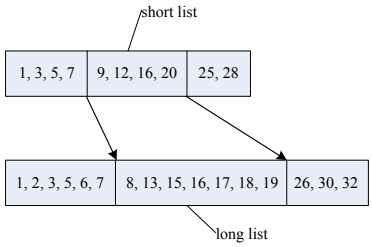

Fig. 1. Partition parallel algorithm.

As experimental results shown in [1], when the length of lists is larger than 100K, partition algorithm is about 4 times faster than simple binary algorithm.

However, the data set we deal with contains a lot of short lists, so the overhead of partition stage may cancel out its gain. Our experimental results confirmed this conjecture.

The CPU algorithm we use for comparison is designed by Ling Jiang, Baidu Inc. It is specially optimized for our data set. As Fig. 2 presents, the longer list is divided into several segments all of equal length. Each element of shorter list is searched in the longer list sequentially, however, the step is one segment instead of one element. If the element is greater than the last element of the current segment, we move to the next segment. Otherwise, we turn around to search backwards in the current segment.

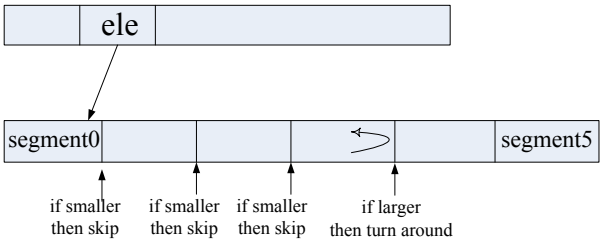

Fig. 2. Stride sequential algorithm.

III. Architecture

We start out with a typical CPU single stream model. Whenever a search query arrives, CPU puts it in the process queue. It is starting to be processed when the predecessor query is completed.

We can convert this model into GPU processing model simply by offloading each intersection calculation to GPU. CPU hands over the query which it receives to GPU, then waits until GPU returns the result. This simple GPU model is just that we used in our preliminary work, called "Serial GPU" model. However, this model can not take full advantage of GPU's computing power when queries are short.

Fig. 3 shows the improved model. In order to make hundreds of GPU shaders busy, queries are pumped to GPU in batches instead of one by one. In this model, CPU is only in charge of task scheduling and data transfering. All calculation tasks are offloaded to GPU.

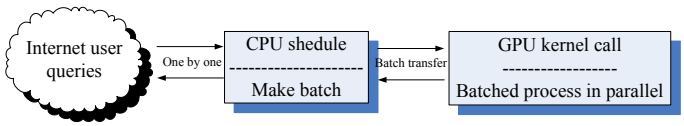

Fig. 3. Batched GPU model.

The schedule procedure consists of four steps:

1) Transfer inverted index lists to GPU memory. If the size of data set is smaller than the capacity of GPU memory, only one transfer operation is performed.
2) Prepare query batch. CPU waits *n* queries arrive, pack them into one batch, then calls the GPU function (GPU kernel) to calculate.
3) GPU runs the kernel function to perform list intersection, and then writes the result to a buffer.
4) The result is transferred back to main memory.

Our main work is to refine step 3. We assign each query to a unique block. So, n blocks all contain t threads, where t depends on the size of input. If all inverted index lists are large, we need more threads per block. So step 3 is refined as:

1) Find the corresponding inverted index lists according to the key words in the query. We sort these inverted index lists by length in ascending order. If perform intersection in this order, there is a strong possibility of achieving optimal computational complexity.
2) Intersection. First, we calculate the intersection of the shortest list and the second shortest list. The result is stored in a "result array". Then we calculate the intersection of the result array and the third shortest list, and so on.
3) Write the length of final result array to a predefined global memory location, so CPU can fetch the result.

The most time consuming part of the whole procedure is step 2. It is composed of three steps:

1) Search: Each thread fetches some elements in the shorter list, then do binary searches in the longer list. If an element is found, its corresponding cell in a temporary array is set to 1 (this array is zeroed when initializing).
2) Scan: We perform a scan (prefix sum) operation on the 0-1 array, so we get the locations of common elements of the two lists in the result array.
   There are some considerations for scan phase:
   a) The number of threads per block is fixed, so we must divide the whole 0-1 array into several segments. Each segment contains the same number of elements as the number of threads. The last segment may be an incomplete segment. We deal with it specially.
   b) An efficient scan algorithm must be adopted, as scan is a frequent operation in the whole procedure. The scan algorithm we adopted is based on the algorithm presented by Hillis and Steele[2], showed in Fig. 4.

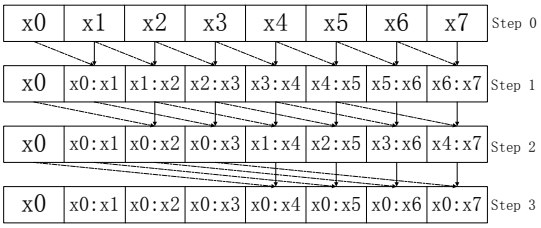

Fig. 4. Scan algorithm.

The main part of the algorithm is a loop. In each iteration, the i-th element is added to the $(i + offset)$-th element. Offset is initialized as 1, and is doubled after each iteration. We can see that this algorithm calculates "inclusive prefix sum". We can see that if 0-1 array contains N elements, the loop will finish after $log(N)$ steps. The j-th iteration performs $N - 2j - 1$ add operations. Thus the algorithm performs $Nlog(N) - (N - 1)$ add operations in total. In order to avoid memory copy, we allocate two buffers both of length N. They are both allocated in shared memory which is much faster and smaller than global memory. After finishing calculating the first segment, we copy the result from shared memory back to global memory. Next we copy the last element of each result segment to the first location in the next segment. Then this element is added to every other element in the second segment. Then the same operation is performed on the second segment and its successor segment, and so on. So the global scan result is calculated correctly. This algorithm is called "naive algorithm" [2]. Although other algorithms exist [2], our

experimental results show that this simple algorithm is the best for our data set.

3) Compacting: each thread is in charge of several elements in the shorter inverted index list. If the corresponding element in 0-1 array is 1, the thread should store the element into the result array. The corresponding element in the scan result designates the right position of this element in the result array.

Fig. 5 illustrates the whole procedure of step 2.

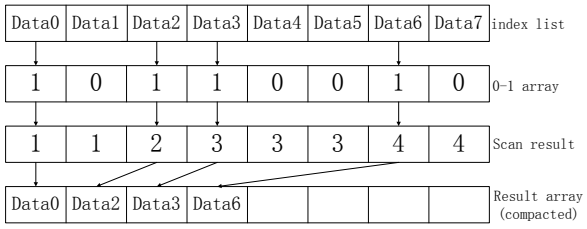

Fig. 5. Calculate final position in result array.

We also implemented an algorithm based on the serial GPU model for comparison. Namely, queries are delivered to GPU one by one. To maximize the degree of parallelism, each intersection operation is processed by multiple blocks, and each thread is in charge of only one element in the shorter list. For example, there are 1271 elements in the shorter list and 128 threads per block, so we will deploy 10 blocks when we call the GPU kernel function. The scan routine used in this algorithm is cudppScan from cudpp library. The compacting step is the same as that in batched algorithm.

IV. Experimental Results

Our experiments are based on CUDA platform version 2.3, detailed experimental results are listed below.

Host: Intel i7 920 CPU, 2GB×2 DDR3 1333 memory.

GPU: 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.

Bandwidth between GPU memory and main memory is about 5.13 GB/s.

The number of threads per block is fixed at 128.

The inverted index lists are stored in one file, named ind_data, which occupies 860MB disk space. As it is smaller than 4GB, so we upload it into GPU memory once. It takes 188.8ms. Host-GPU bandwidth occupancy is about 88.8%.

The test_data includes 33337 search queries, and we simulate it as a query stream comes into search server in short time. Since the test_data is a fraction of actual search query stream, the workload of each query is quite random, which is an important influence factor to GPU response time. CPU schedules the queries to GPU, and fetches the result after GPU finishes its work.

Fig. 6 shows the distribution of the length of shortest inverted index lists of queries. When we refer to "shortest", we mean that it is the shortest list in its query. For example, query0 has six key words, which are corresponding to six inverted index lists. We can select the shortest list out. So, we will have 33337 shortest inverted index lists in total. Since queries have been uploaded to GPU memory, the sort of each query is implemented with only one GPU thread. The number of lists in one query is less than 8, so we adopt insertion sort algorithm. While the procedure of sort, other threads are all waiting. Our experimental results show that the overhead of this step occupies only 20ms in total.

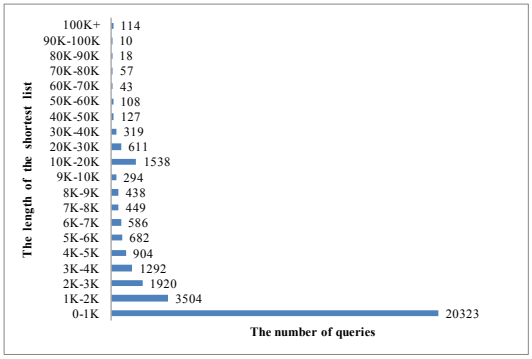

Fig. 6. Query size distribution.

The length of the shortest list reflects the workload of the query directly. For example, the length of the shortest list in query0 is 259, so after five intersections, the length of result array is smaller or equal to 259. We can see that 91% of shortest lists in our data set are shorter than 10k docIDs.

We now perform a preliminary evaluation of our two GPU models. The queries are delivered to GPU according to their original order in file test_data. In the "serial processing" model, we deploy 128 threads per block. Thus for 77.2% queries, at most 30 blocks are invoked. Tesla C1060 has 30 multiprocessors, so the workload can not feed it.

Fig. 7 shows the runtime of the two algorithms. We can see that the batched algorithm is obviously superior to the plain algorithm because it utilizes more processor cores. When the size of batch is larger than 8192, the runtime remain stable at about 3016ms.

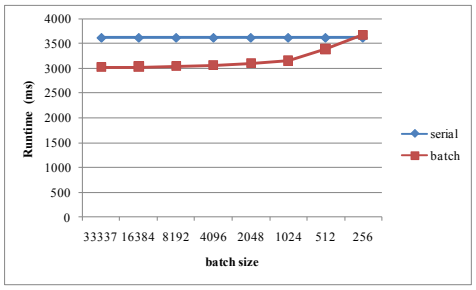

Fig. 7. Original order result.

TABLE I shows the performance gap between the CPU algorithm and the batched GPU algorithm. The CPU program is still much faster than the GPU one.

Further examination on the data set exposes serious load imbalance. If queries are fed to GPU in their original order in test_data, the queries in a batch will vary in size. Therefore when some queries have completed, other queries may be still

TABLE I
ORIGINAL ORDER RUNTIME ON CPU AND GPU.

	calculating time (ms)
GPU batch model (test_data)	3016
CPU (test_data)	1569

in process. Since a batch is processed by a single kernel call and current CUDA version can not execute multiple kernels simultaneously, perhaps only a minority of GPU processor cores are busy in the later stages of query processing. Besides, the different number of key words is also the reason for load imbalance. The main objective of batched algorithm - making full use of GPU processor cores - fails.

To solve the problem, we must pack the queries of similar size into a batch. We can estimate the problem size of each query by the length of its shortest list. Our experimental results shows that this estimation is reasonable. How to find queries of similar size? Sorting queries completely by size is unpractical. We adopted semi-sort: divide the range of list length into several buckets and assign each query to a bucket according to the length of its shortest list. In general, Here comes n queries which will be allocated into c buckets, each of them needs c-1 comparison times at most. This preprocessing is implemented by CPU, and is a linear time operation. Compared with complete sorting, this method introduces much lower overhead. Certainly, we will divide a big bucket into multiple batches.

We performed this preprocess on our data set. The length range is divided into 20 buckets. 0-10K is divided into 10 buckets all of interval 1K, and 10K-100K is divided into 10 buckets all of interval 10K. Then we test the CPU algorithm and the two GPU algorithms using this rearranged test data. For batched GPU algorithm, batch size is set to 16384. Fig. 8 shows the result. We can see that new input order improves the performance of batched GPU algorithm dramatically. GPU batched algorithm spent 421 ms on calculating and 150 ms on dynamic transfer. It decreases calculating time 6 times compared with the unordered input. Now, batched GPU algorithm is 2.7x faster than CPU algorithm, and 5.2x faster than plain GPU algorithm.

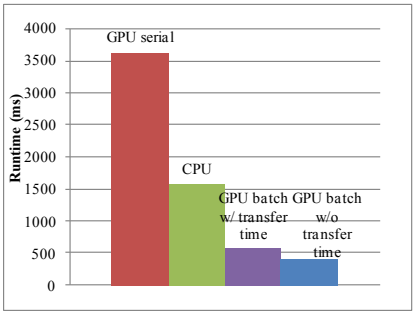

Fig. 8. Runtime on ordered input.

We also test CPU algorithm and batched algorithm using individual bucket. Each bucket is divided into batches of fixed length. The result is shown in Fig. 9. We can see that if the number of queries per batch is over 512, GPU algorithm is faster than CPU algorithm for all buckets. As the batch size decreases to 64, the runtime increases. The reason is that 64 queries can not take full advantage of GPU.

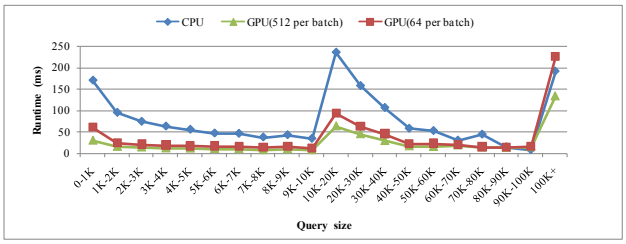

Fig. 9. Runtime on single bucket.

Since the batch size (equals to the number of blocks per GPU kernel call) influences the performance greatly, we tested batched GPU algorithm using different batch sizes. The whole data set instead of individual buckets is used. Fig. 10 shows the result. We can see that the algorithm always exhibits good performance unless the batch size is extremely small (32). It achieves peak performance at 1024 and holds the line as the batch size continues to increase.

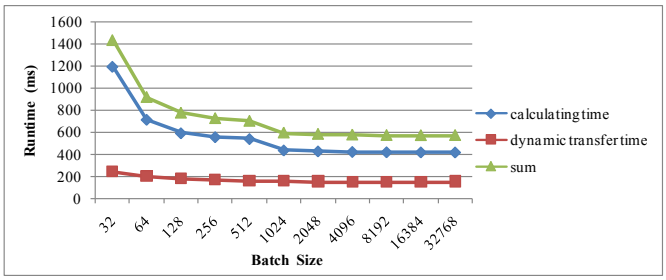

Fig. 10. Runtime of different batch size.

There are three data transfer steps in the batched algorithm:

1) Transfer inverted index lists to GPU memory. In our case, all lists can be stored in GPU memory, so this type of transfer needs once only. We call it “static transfer overhead”. Step 2 and 3 are called “dynamic transfer overhead”.
2) Transfer queries onto GPU memory. The transfer must be invoked before every GPU kernel call. Since the space occupancy of queries is quite small (all 33337 queries only occupy 549KB), so it induces a light overhead.
3) Transfer result arrays back to main memory. There are two methods for the task:
   a) Transfer result arrays separately. For example, there are 128 queries in the batch applied to GPU. When GPU kernel function returns, we invoke 128 times cudaMemcpy to fetch the result respectively.
   b) Transfer result arrays at a time. A large result array will be allocated on GPU memory before each GPU kernel call, whose size is the length sum of the size of all shortest lists in this batch. It is sufficient for GPU to use as the intersection result is shorter than the length of the shortest list. However, memory space is wasted. For example,

the length of shortest list is 1271, while the intersection result contains only 125 docIDs. Now we still need transfer 1271 docIDs and allocate 1271 docIDs memory space to receive the result.

The transfer time is not only determined by the size of data, but also be influenced significantly by the number of transfer operations. TABLE II compares the performance of these two transfer methods. The batch size is set to 2048. There exists dramatic performance gap between the two methods. If we transfer the result one by one, transfer overhead will cancel out the advantage of batched processing. So the best method is to transfer all results together, despite some memory space is wasted. Fig. 10 shows the detailed dynamic transfer time under different batch sizes. We can see that the transfer time is only a small part of total time.

TABLE II
TRANSFER TIME OF THE TWO METHODS.

	time used by step 2 and 3 (ms)
transfer separately	461.6
transfer as a whole	155.1

Besides the total calculating time and total dynamic transferring time, the response time of each batch is also an important factor we care about. For example, if batch size is 16384, there are 3 batches. The total calculation time is 421ms, and the total transferring time is 150ms. So, each batch takes about 190ms. It means that the response time is about 190ms which is obviously not acceptable. Fig.11 shows the response time curve when batch size varies.

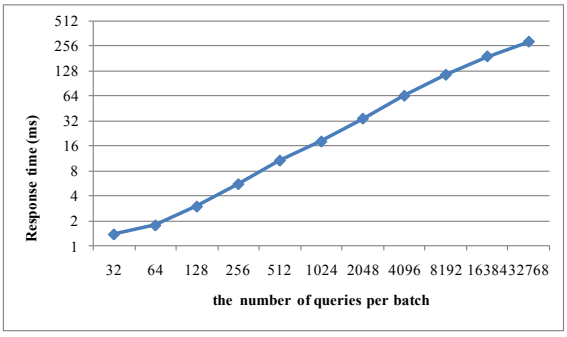

Fig. 11. Response time.

Compared with Fig. 10 and Fig. 11, we can determine the best choice of batch size. On our data set, 1024 or 2048 queries per batch will achieve the good tradeoff between throughput and response time.

V. CONCLUSION

We have proposed an efficient GPU algorithm for high performance intersection of inverted index lists. This algorithm fed enough queries to GPU in batch, thus takes full advantage of GPU processor cores. We also proposed an input preprocessing method which alleviates load imbalance effectively. Our experimental results based on a real world test set show that the batched algorithm is much faster than the fastest CPU algorithm and plain GPU algorithm.

We believe that GPUs 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 be done to make our algorithm more practical. Our experiments are all performed off line. Designing an online input preprocessing algorithm is an important future work. Since the data set in real search engine systems may be much larger than our test set, a streamed GPU algorithm should be developed to hide transfer overhead. Multi-GPU algorithm is also an interesting topic.

VI. 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 Ling Jiang for providing the efficient CPU algorithm. We also must thank Hao Yan for providing related code.

REFERENCES

[1] S. Ding, J. He, H. Yan, and T. Suel, "Using Graphics Processors for High Performance IR Query Processing," in *WWW 2009*, April 20-24, 2009, Madrid, Spain.

[2] M. Harris, "Parallel prefix sum (scan) with CUDA," , www.nvidia.com, April 2007.

[3] "Nvidia CUDA programming guide 2.1," http://www.nvidia.com/object/cuda_develop.html.

[4] S. Sengupta, M. Harris, and M. Garland, "Ecient Parallel Scan Algorithms for GPUs," *NVIDIA Technical Report*, NVR-2008-003, Dec. 2008.

[5] E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro, "adaptive set intersections,unions,and differences," *Eleventh ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[6] F. Samuel, J. Barbay, and M. McCool, "Implementation of Parallel Set Intersection for Keyword Search using RapidMind," *University of Waterloo Technical Report*, CS-2007-12.

[7] G. E. Blelloch, and M. Reid-Miller, "Fast Set Operations Using Treaps," *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'98.

[8] N. Satish, M. Harris, and M. Garland, "Designing Effcient Sorting Algorithms for Manycore GPUs," in *23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[9] "NVIDIA CUDA C Programming Best Practices Guide," http://www.nvidia.com/object/cuda_develop.html.