High Performance Approximate Sort Algorithm Using GPUs

Jun Xiao, Hao Chen, Jianhua Sun
College of Computer Science and Electronic Engineering
Hunan University
Changsha, China
xiaojun9081@gmail.com, haochen@aimlab.org, jhsun@aimlab.org

***Abstract*—Sorting is a fundamental problem in computer science, and the strict sorting usually means a strict order with ascending or descending. However, some applications in reality don't require the strict ascending or descending order and the approximate ascending or descending order just meets the requirement.**

Graphics processing units (GPUs) have become accelerators for parallel computing. In this paper, based on the popular CUDA parallel computing architecture, we propose high performance approximate sort algorithm running on multicore GPUs. The algorithm divides the distribution interval of input data into multiple small intervals, and then uses the processing cores of GPUs to map the data into the different intervals in parallel. Finally by combining the small intervals, we can make the data between the different intervals in order state and the data in the same interval is disorder state. Thus we can get the approximate sorting result and the result is characterized by a general order but local disorder. By utilize the massive core of GPUs to parallel sort data, the algorithm can greatly shorten the execution time. Radix sort is the fastest GPUs-based sorting and the experimental results show that our approximate sort algorithm is two times as fast as the radix sort and far exceeds all the GPUs-based sorting.

Keywords—sorting, parallel computing, high performance, GPUs, CUDA

I. INTRODUCTION

Sorting is one of most widely studied algorithmic problems in computer science, and has become a fundamental component in data structures and algorithms analysis. Many applications could be just classified as sorting problem, and the other applications depend on the efficient sorting as an intermediate step to accelerate the execution time [1], [2]. For example, search engine widely uses of sorting to select valuable information to users. Therefore, designing and implementing efficient sorting routine is important on any parallel platforms. As many parallel platforms spring up, we need to explore efficient sorting techniques for utilizing parallel computing power [3].

Recently, Graphics Processing Units have evolved into high performance accelerators and provide considerably higher peak computing and memory bandwidth than CPUs[4]. For instance, NVIDIA's GeForce GTX 780 GPUs contain up to 192 scalar processing cores (SPs) per chip. And, these cores are broken up into 12 Streaming Multiprocessors (SMs) and each SM comprises 16 SPs. A 3GB off-chip global memory is shared by the 192 on-chip cores. By introduction of CUDA, programmers could use C to program GPUs for general-purpose computation [5]. In consequence, it is an explosion of research on GPUs for high performance computing [6]. With the high computing power, advanced features such as atomic operations, shared memory and synchronization, also lead into modern GPUs.

Many researchers have proposed GPUs-based sorting algorithms and transit from the coarse-grained parallelism of multicore chips to the fine-grained parallelism of manycore chips. Quick sort is a popular sorting algorithm, and Cederman et al. [7] have adapted quick sort for GPUs to parallelization. Satish et al. [3] have designed efficient sorting algorithms to make use of the fast on-chip memory provided by NVIDIA GPU and change from a largely task-parallel structure to a more data-parallel structure. The studies of GPUs sorting mainly concentrate on bitonic sort, quick sort, radix sort and merge sort.

However, these GPUs-based sorting are belong to the strict sorting. The strict sorting usually means the strict order with ascending or descending after sorting. Some applications in the reality don't necessarily require the strictly ascending or descending order, and tolerate unsorted order to some extent. As a result, the approximately ascending or descending order already meets the requirement. In this situation, the overhead of the strict sorting is relatively high.

Our focus, in this paper, is to develop the approximate sort on manycore GPUs which is suitable for sorting data to reach the state of the approximately ascending or descending order. Our experimental results demonstrate that our approximate sort is fastest in all previously published GPUs sorting when running on current-generation NVIDIA GPUs. The radix sort is the fastest GPUs sorting for the large amount data[3] and our approximate sort could achieve at least more than twice compared with GPUs-based radix sort.

The rest of this paper is organized as follows: In Section 2 we will describe the background on GPUs architecture and the sorting on GPUs. In Section 3 we will elaborate the approximate sort in detail. In Section 4 we will present the

experimental evaluation of the approximate sort compared with GPUs-based sorting.

II. BACKGROUND

In this section, we will provide background information on GPU architecture and the GPU-based sorting.

A. *GPUs architecture*

Our approximate sort algorithm is designed and implemented on the NVIDIA GPUs architecture. GPUs have become high performance accelerators for parallel computing, which are massively multi-threaded data-parallel processor. GPUs contain two major components: the processing component and the memory component.

A certain number of streaming multiprocessors comprises the processing component. At the same time, each streaming multiprocessor includes a series of simple cores that execute the in-order instructions. For high performance, a few tens of thousands of threads are launched and these threads carry out the same instruction on the different data sets. Threads in GPUs have three-level hierarchy: each block includes hundreds of threads mapped to a streaming multiprocessor and a grid contains a set of blocks executed on a kernel [8].

In the memory component, the off-chip global memory in GPUs is accessible across all streaming multiprocessors. The data transfer between host and device memory is at the means of DMA. A 16KB on-chip cache equipped in each streaming multiprocessor, which has very high bandwidth and very low access latency.

Our approximate sort algorithm leverages the CUDA Data Parallel Primitives library [9], specifically its scan and reduce. By using the CUDPP library, we avoid do tedious work that the CUDPP has done for us.

B. *Sorting on GPUs*

We here present only the most relevant work because sorting on GPUs has always been the research hotspot.

Early GPUs-based sorting algorithms were primarily based on Batcher's bitonic sort[10]. Barajlia et al. [11] presented a practical bitonic sorting network implemented in CUDA when bringing in the new general-purpose parallel platform. Cederman et al. [7] developed an efficient implementation of GPUs quick sort to make use of the highly parallel nature and its limited cache memory. Satish et al. designed efficient parallel radix sort and merge sort for GPUs, and their radix sort is the fastest GPU sort [3].

Above mentioned sorting can be viewed as a feasible alternative to sort a large amount of data on GPUs. However, these sorting routines are all belong to the strict sorting. We define the strict sorting that the strict order with ascending or descending after sorting, otherwise call as the approximate sorting. For example, we have an input array of (10, 8, 2, 9, 3, 1) and sort in ascending order. If the output is (1, 2, 3, 8, 9, 10) with strict order, the sorting algorithm used is part of the strict sorting. If the output is (1, 3, 2, 10, 9, 8) or others with unsorted within the interval and sorted between the intervals, the sorting algorithm used belongs to the approximate sorting. The length of the interval controlled by the users and the length of the interval is 3 in this case. For further explanation, (1, 3, 2) and (10, 9, 8) are two intervals. (1, 3, 2) or (10, 9, 8) is unsorted but every element in (1, 3, 2) is less than the one in (10, 9, 8), that is the ascending order between the intervals and it means the approximately ascending order.

Some applications in the reality don't necessarily require the strictly ascending or descending order, and tolerate unsorted order to some extent. As a result, the approximately ascending or descending order already meets the requirement. In this situation, the overhead of the traditional sorting is relatively high. We propose lightweight approximate sort on manycore GPUs to address the above problem.

III. APPROXIMATE SORT ON GPUS

In the following section, we present the detail of approximate sort algorithm on GPUs to parallelism.

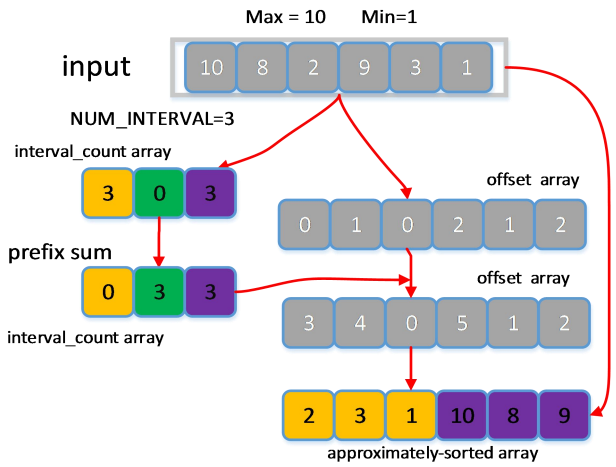

Fig. 1. Illustration of approximate sort on GPUs

As shown in Figure 1, our algorithm on GPUs operates in three steps. First, each data element in the input array is mapped into a smaller interval (the number of the smaller intervals is a pre-defined parameter and typically much less than the input size, NUM_INTERVAL=3 in our case). In this step, we use offset array to maintain an ordering among all data elements that are mapped into the same interval. At the same time, the interval counter array is use to record the number of data elements falling into each interval. Second, an exclusive prefix sum operation is performed on the interval counter array. In the third step, the results of the above two steps are combined to produce the final coordinates that are then used to transform the input array to the approximately-sorted form.

Step 1: Similar to many parallel sort algorithms that subdivide the input into the equally-sized intervals and then sort each interval in parallel, we first map each data element of the input array into an interval. As shown in Listing 1, the number of the interval is a fixed value NUM_INTERVAL, and the mapping procedure is a linear projection of each data element of the input vector to one of the NUM_INTERVAL

intervals. The linear projection is demonstrated at lines 10 and 11 in Listing 1. The variables of min and max represent the minimum and maximum value in the input respectively, which can be obtained when using the CUDPP's reduce tool on GPUs. In this way, each interval represents a partition of the interval [min, max], and all intervals have the same width of *(max − min) / NUM_INTERVAL*. The data elements in the input array are assigned to the target interval whose value range contains the corresponding data element, and for brief illustration we use interval_index array to record the target interval. In addition, another array interval_count is maintained to record the number of data assigned to each interval. As shown at line 13, the offset array is based on an atomic function provided by CUDA, atomicInc, to avoid the potential conflicts incurred by concurrent writes. The function atomicInc returns the old value located at the address presented by its first parameter, which can be leveraged to indicate the local ordering among all the data elements assigned to the same interval. The Kepler GPUs have substantially improved the throughput of atomic operations compared to Fermi GPUs, which also demonstrated in our implementation.

Listing 1: Assigning elements to intervals.

```
1  __global__  void assign_interval (uint *input, uint lenght, uint max, uint min,
2                                    uint *offset, uint *interval_count, uint *interval_index )
3  {
4       int  idx = threadx.x+blockDim.x*blockIdx.x;
5       uint interval_idx;
6       for ( ; idx<lenght ; idx+=total_threads)
7       {
8            uint value = input[idx];
9
10           interval_idx = (size − min) * (NUM_INTERVAL − 1) / (max − min);
11           interval_index[idx] = interval_idx;
12
13          offset[idx] = atomicInc (&interval count[interval_idx],length);
14      }
15 }
```

Listing 2: the key step of approximate sort.

```
1  __global__ void appr_sort (uint *key, uint *key_sorted, void *value, uint length,
2                             void *value_sorted, uint *offset, uint *interval_count,
3                                uint *interval_index)
4 {
5       int idx = threadIdx.x + blockDim.x * blockIdx.x;
6       uint count=0;
7       for (;idx < length;idx += total threads)
8       {
9           uint  Key = key[idx];
10          uint  Value = value[idx];
11
12          uint Interval_index = interval_index[idx];
13          count = interval_count[Interval_index];
14          uint  off = offset[idx];
15          off = off + count;
16
17          key_sorted[off] = key;
18          value_sort[off] = value;
19      }
20 }
```

Step 2: Having obtained the counters for each interval and the local ordering within a specific interval, we perform a prefix sum operation on the interval_count array to determine the address at which each interval's data would start. Given an input array, the prefix sum, also known as scan, is to generate a new array B from original array A in which each data B[*i*] is the sum of data from A[0] to A[*i*] (inclusive and exclusive prefix sum respectively). Because the length of the interval count_array (NUM_INTERVAL) is typically less than that of the length of the input, performing the scan operation on CPU is much fast than the GPUs counterpart. However, due to the data transfer overhead (in our case, two transfers), and the fact that we observed devastating performance degradation when mixing the execution of the CPU-based scan with other GPUs kernels in a CUDA stream, the parallel prefix sum is performed on GPUs using the CUDPP library.

Step 3: By combining the atomically-incremented offsets generated in step 1 and the interval data locations produced by the prefix sum in step 2(as shown at lines 12-15 in Listing 2), it is straightforward to scatter the key-value pairs to proper locations (see lines 17-18).

Choosing a suitable value for the number of intervals may have important implications for the efficiency and effectiveness of our sorting algorithm. As the number of intervals increases, if the input data exhibiting uniform distribution of elements, our algorithm would approximate more closely to the ideal sorting, while the overhead of performing the prefix sum may increase accordingly. When decreasing the number of intervals, we will get a coarse-grained approximation for the input array. We will present empirical evaluations on this in Section IV.

IV. EXPERIMENTAL EVALUATION

A. *Experiment setup*

We ran the experiments on an eight-processor Intel Xeon E5 2648L 1.8GHz machine. At the same time, the machine equipped with a high-end NVIDIA GeForce GTX 780 GPUs with 12 multiprocessors and 192 GPUs processing cores.

We compared approximate sort on GPUs with the following state-of-the-art GPUs sorting algorithms: Satish et al.'s[3] merge sort and radix sort. Because the version of radix sort is the fastest GPUs sort and the version of merge sort is the fastest comparison-based GPUs sort according to the reference. At the same time, the source code of that merge sort and radix sort is available in the NVIDIA CUDA SDK[12].

The data sets we automatically generated for the benchmark test conform to Uniform distribution or Gaussian distribution. Values that are picked randomly from 0 to 231 produce Uniform distribution. The Gaussian distribution is created by always taking the average of four randomly picked values from the uniform distribution [7]. We choose the two distributions for the representative.

B. *Performance analysis*

We compare our approximate sort with merge sort and radix sort on GPUs. First, we generate respectively three data sets on Uniform distribution and Gaussian distribution. The size of the data set we evaluate is 1M, 2M, 4M (M means 10^6

in this paper) and we set the NUM_INTERVAL = 10000. As shown in Figure 2 and Figure 3, the performance on the two distributions is roughly the same. When the data volume is doubling, the cost of approximate sort slowly increases compared with merge sort. Our approximate could achieve at least more than twice compare with radix sort.

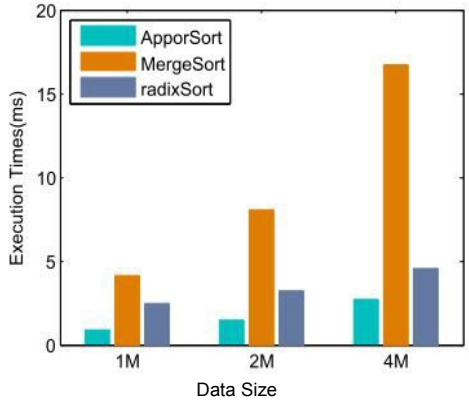

Fig. 2. Data sets on Uniform distribution

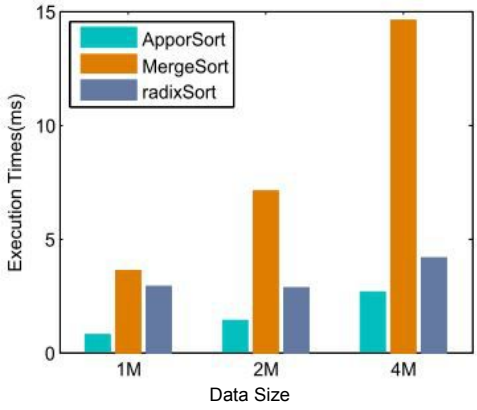

Fig. 3. Data sets on Gaussian distribution

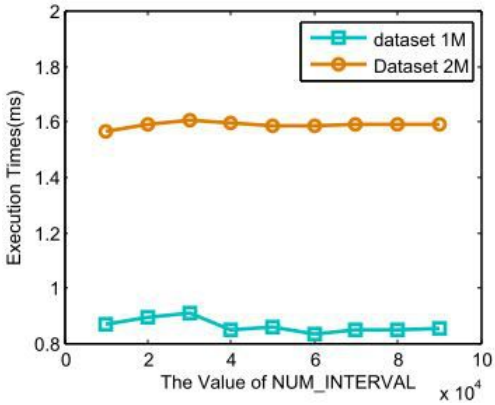

Fig. 4. The parameter of NUM_INTERVAL

In the Figure 4, we evaluate how the parameter of NUM_ INTERVAL effects on performance. We prepare two data set on Uniform distribution and the size of the data set respectively 1M and 2M. The values of NUM_INTERVAL is (10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000). As the NUM_INTERVAL increased, the execution time of approximate sort almost the same. When the NUM_INTERVAL is small, the cost of atomic operation is high because multiple elements are assigned to the same interval concurrently and the overhead of prefix sum is small. When the NUM_INTERVAL is large, the cost of atomic operation is low because fewer elements are assigned to the same interval concurrently but the overhead of prefix sum is expensive. It is suggested that the performance almost keep same when the NUM_INTERVAL changes within a certain range.

V. CONCLUSIONS

This paper, we propose approximate sort on manycore GPUs to parallelism. The approximate sort could obtain the approximate order with ascending or descending by controlling the parameter of NUM_INTERVAL. The radix sort is the fastest GPUs sort and our approximate sort could achieve at least more than twice compared with GPUs-based radix sort.

As for future, our work is to integrate our approximate sort into the application in the reality.

VI. ACKNOWLEDGMENT

This research was supported in part by the National Science Foundation of China under grants 61272190 and 61173166, the Program for New Century Excellent Talents in University, and the Fundamental Research Funds for the Central Universities of China.

REFERENCES

[1] D. E. Kauth, “The art of computer programming: Volume 3/sorting and searching,” 1973.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein et al., Introduction to algorithms. MIT press Cambridge, 2001, vol. 2.

[3] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore gpus,” in Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009 , pp. 1–10.

[4] C. Nvidia, “Nvidia cuda c programming guide,” NVIDIA Corporation, vol. 120, 2011.

[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” Queue, vol. 6, no. 2, pp. 40–53, 2008.

[6] S. Bandyopadhyay and S. Sahni, “Grsgpu radix sort for multifield records,” in High Performance Computing (HiPC), 2010 International Conference on. IEEE, 2010, pp. 1–10.

[7] D. Cederman and P. Tsigas, “A practical quicksort algorithm for graphics processors,” in Algorithms-ESA 2008. Springer, 2008, pp. 246–258.

[8] L. Chen and G. Agrawal, “Optimizing mapreduce for gpus with effective shared memory usage,” in Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. ACM, 2012, pp. 199–210.

[9] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, “Cudpp: Cuda data parallel primitives library,” 2007.

[10] K. E. Batcher, “Sorting networks and their applications,” in Proceedings of the April 30–May 2, 1968, spring joint computer conference. ACM, 1968, pp. 307–314.

[11] R. Baraglia, G. Capannini, F. M. Nardini, and F. Silvestri, “Sorting using bitonic network with cuda,” in the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR), Boston, USA, 2009.

[12] “Nvidia cuda sdk,” (http://www.nvidia.com/cuda), 2014.