GPUMemSort: A High Performance Graphics Co-processors Sorting Algorithm for Large Scale In-Memory Data

Yin Ye[1], Zhihui Du[1+], David A. Bader[2] , Quan Yang [1] and Weiwei Huo[3]
[1] *National Laboratory for Information Science and Technology*
Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China
[2] *College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332, USA*
[3] *School of Information and Communication Engineering, Beijing University of Posts and Telecommunications, China*
[+]*Corresponding Author's Email: duzh@tsinghua.edu.cn*

Abstract

In this paper, we present a GPU-based sorting algorithm, GPUMemSort, which achieves high performance in sorting large-scale in-memory data by take advantage of GPU processors. It consists of two algorithms: an in-core algorithm, which is responsible for sorting data in GPU global memory efficiently, and an out-of-core algorithm, which is responsible for dividing large-scale data into multiple chunks that fit GPU global memory. GPUMemSort is implemented based on NVIDIA's CUDA framework and some critical and detailed optimization methods are also presented. The tests of different algorithms have been run on multiple data sets. The experimental results show that our in-core sorting can outperform other comparison-based algorithms and GPUMemSort is highly effective in sorting large-scale in-memory data.

Keywords: Parallel Sorting Algorithm, GPU, CUDA

1. Introduction

With the improvement of CPU performance and multi-core CPUs, bandwidth between CPU and memory becomes the bottleneck of large-scale computing. Many hardware vendors, such as AMD, IBM, and NVIDIA integrate co-processors to offload tasks from the CPU, and this can alleviate the effects of low CPU-memory bandwidth. Meanwhile, high performance computers have ever-increasing amounts of memory, so it is very important to develop efficient co-processor algorithms to deal with large-scale in-memory data.

Recently, GPUs (Graphics Processing Units) have become the best-known co-processor. They have been utilized in many different sorts of general purpose applications. GPUs are suitable for highly parallel, compute-intensive workloads because of higher memory bandwidth and thousands of hardware thread contexts with hundreds of parallel compute pipelines executing programs in a SIMD (single instruction multiple data) fashion. The peak performance of GPUs has been increasing at the rate of 2.5–3.0 times a year, much faster than the performance of CPUs.

Several GPGPU (General Purpose computing on GPUs) languages, such as OpenCL [2] and NVIDIA CUDA [1] are proposed for developers to use GPUs with extended C programming language, instead of graphics API. In CUDA, threads are organized in a hierarchy of grids, blocks, and threads executed in a SIMT (single-instruction, multiple-thread) manner; threads are virtually mapped to an arbitrary number of streaming multiprocessors (SMs) through warps. There exist several types of memory, such as register, local memory, shared memory, global memory, constant memory, *etc*. Different types of memory have different characteristics. Therefore, how to organize the memory access hierarchy is very important for improving programs' performance. In this paper, the GPU part of our algorithm is implemented with CUDA and we will show how we design and optimize memory access pattern in details.

Main Contribution: We propose a novel graphics co-processor sorting algorithm to sort large-scale in-memory data. Our idea is to split a large-scale sorting task into a number of disjoint ones which can fit GPU memory. In general, our contributions are as follows:

(1) We provide a method which can efficiently divide the large-scale in-memory data into disjointed subsets so that they can be sorted by GPU quickly.

(2) We improve the performance of GPU Sample Sort [12] algorithm and the enhanced algorithm outperforms the existing GPU sorting algorithms.

Table 1 summarizes this paper's notation. The paper is organized as follows. Sections 2 will introduce the background and the related work. In section 3, the proposed algorithm is introduced. Detailed implementation and optimization will be presented in section 4. Our experimental results are shown in section 5. In section 6, we will give the conclusion and future work.

Table1. NOTATION

NOTATION	DESCRIPTION
N	number of elements in the input data set
n	size of elements which can fit into the global memory
d	number of chunks
s	number of sample points
s[i]	the i^{th} sample point
e[i]	the i^{th} input element
list[i]	the i^{th} sorted list

2. Background and Related Work

2.1. Parallel Sorting Algorithm

Parallel sorting has been studied extensively during the past 30 years. Generally, parallel sorting algorithms can be

DOI: 10.5176_2010-2283_1.2.34

divided into two categories [3]:

• Partition-based Sorting: First, use partition keys to split the data into disjoint buckets. Second, sort each bucket independently, and then concatenate the sorted buckets.

• Merge-based Sorting: First, partition the input data into data chunks of approximately equal size and sort these data chunks in different processors. Second, merge the data across all of the processors.

Each category has its own potential bottleneck. Partition-based algorithms have to deal with problem of how to keep load balanced across all the processors. Merge-based sorting algorithms perform well only for a small number of processors.

To solve the load balance problem, Parallel Sorting by Regular Sample (PSRS) [5] guarantees that the size of data chunk assigned to processor is less than $(2n/p - n/p^2 - p + 1)$. A newer approach [4] can guarantee that each processor will have at most $(n/p + n/s - p)$ elements, where $p \le s \le n/p^2$ and s is a parameter.

2.2. GPU Programming with CUDA

The NVIDIA CUDA programming model is created for developing applications on GPUs. Some major principles [6] on this platform are: (1) Leverage zero-overhead thread scheduling to hide memory latency. (2) Optimize the use of on-chip memory to reduce bandwidth usage and redundant execution. (3) Group threads to avoid SIMD penalties and memory port/bank conflicts. (4) Threads within a thread block can communicate via synchronization, but there is no built-in global communication mechanism for all threads.

2.3. Parallel Sorting Algorithm based on GPU

Since most sorting algorithms are bound by memory bandwidth, sorting on the high-bandwidth GPUs becomes a popular topic. Purcell [7] introduced bitonic merge sort, while Kipfer and Westermann [8] improved it to odd-even merge sort. Greß and Zachmann [9] introduced the GPUABiSort based on adaptive bitonic sorting. Naga K. Govindaraju [3] presented a GPUTeraSort algorithm to sort billion record wide-key databases. Also, some CUDA-based sorting algorithms have been proposed recently. Erik Sintorn [10] introduced a hybrid algorithm combining bucket sort and merge sort, but can only sort floats as it uses a float4 in merge sort. Cederman [11] proposed Quicksort in CUDA, which is sensitive to the distribution of the input data. The comparison-based Thrust Merge method by Nadathur Satish, et al. combines odd-even merge and two-way merge to balance the load. Satish et al.[13] presented GPU radix sort for integers. [12] is a randomized sample sort that significantly outperforms Thrust Merge. Because of its random selection, the load balancing does not perform well.

However, most of these algorithms are designed for small-scale data sorting and are ineffective when data size is larger than the global memory size.

3. GPUMemSort Algorithm

In this section, we will present GPUMemSort which consists of two major parts, the out-of-core algorithm and the in-core algorithm. The aim of the algorithm is to sort large-scale data on the CUDA platform to achieve parallel sorting, and to find new ways to solve the loading balancing problem. Specifically, the out-of-core sorting can divide large-scale data into multiple disjointed subsets and assign them to GPU. The In-core sorting aims at sorting the subsets efficiently. Two aspects that will influence the performance of GPUMemSort algorithm. One is how to implement coalesced Memory Access in the share memory, which can be an important factor in deciding the sorting time. The other is the tradeoff between the splitters-finding algorithm and the bucket sorting algorithm, which will be discussed in the following statements. We make efforts on these aspects and the results show that our algorithm has significant improvement on the original Sample Sort.

3.1 Out-of-core algorithm

In this section, we introduce the out-of-core algorithm to tackle with large-scale data on the platform of multi-core GPUs. The main idea behind the algorithm is to cut the huge task into a number of subtasks, whose sizes do not have significant differences. This will guarantee the data will be put into the global memory, and minimize the balancing problem as well. This is the first step of the GPUMemSort and lays an ideal foundation for the in-core sort.

We adopt the ideas of Deterministic Sample-based Parallel Sorting (DSPS) in our out-of-core sorting algorithm. The idea behind DSPS algorithm is to find s-1 samples as splitters to partition the input data set into several data chunks. Elements in the $(i+1)^{th}$ chunk are no smaller than those in the i^{th} chunk. The sizes of these chunks have a deterministic upper bound in order to avoid the loads of streams differ greatly and causing the load balancing problem. The chunks can be put into GPU global memory by adjusting a parameter in the algorithm according to the value of the whole data size.

The out-of-core algorithm can be described as follows:

Step1: Divide the input data set into d chunks, each contains (n/d) elements, assuming that d divides n evenly.

Step 2: Copy the chunks to the GPU's global memory one by one, and sort them by in-core algorithm. Then split each chunk into d buckets. The x^{th} element in chunk i will be put into bucket $Bin[i][\lfloor x/d \rfloor]$. Copy the sorted chunks back to main memory one by one.

Step 3: Swap buckets among chunks in main memory, for $i \in [0,d-1]$, $j \in (i,d-1]$, switch $Bin[i][j]$ and $Bin[j][i]$. So that new chunk i consists of $\{Bin[0][i],Bin[1][i],...,Bin[d-1][i]\}$.

Step 4: In the $(d\text{-}1)^{th}$ chunk, selects the $((x+1)\times n/(d^2\times s))^{th}$ element as a sample candidate from $Bin[i][d-1]$, for $x \in [0,s-1]$ and $i \in [0,d-1]$.

Step 5: Sort the sample candidate, pick the $(k+1)\cdot s$ sample point as $s[k]$, $k \in [0,d-2]$, let $s[d-1]$ be the largest. Copy the sample array from main memory to GPU global memory.

Step 6: Copy each chunk to GPU global memory again and split the chunk into d buckets based on d sample points. The bucket j of chunk i is called $NS[i][j]$, for $0 \le j \le d-1$. After splitting, all the elements in $NS[i][j]$ should be no larger than $s[j]$. At last, copy these buckets back to main memory.

Step 7: Swap buckets among chunks again in main

memory, new chunk *i* consists of {*NS[0][i], NS[1][i],...,NS[d-1][i]*}. $i \in [0, d-1]$. All the elements in chunk *i* are no larger than *s*[*i*].

Step 8: $\forall i \in [0,d-1]$, calculate the total length of chunk *i*. If the length is less than the threshold Θ, copy the whole chunk to GPU global memory and sort it using our in-core sorting algorithm. Otherwise, copy *NS[0][i], NS[1][i],...,NS[d-1][i]* to the GPU one by one. For *NS*[*j*][*i*], split it into two parts, *part[j][i][0]* and *part[j][i][1]*, where *part[j][i][0]* contains elements equal to *s[i]* while *part[j][i][1]* contains the rest. Copy back the *part[j][i][1]* to the main memory, then merge all the *part[j][i][1]*, $0 \le j \le d-1$ into one array. Finally, sort this array on the GPU and write it back to the result set. Fill out the rest part of result set using *s[i]*.

According to the condition in [5], we can easily get $(n/d + n/s - d) < \theta$,so $d \ge n/(\theta/2 - n/(2s) + \sqrt{(n/s-\theta)/4+n})$ .This means that if every chunk's size is guaranteed to be less than Θ, the number of chunks split in Step 1 must be larger than $\lceil n/(\theta/2 - n/(2s) + \sqrt{(n/s-\theta)/4+n}) \rceil$.

Suppose that GPU is able to store and sort a 128MB data set, then the sample number *s=64*, the *N = 1* GB, according to the approach above, $d \ge 8.47$, so that *d* must be larger than or equal to 9.

3.2 In-core algorithm

Our in-core algorithm is based on GPU Sample Sort, which is currently the fastest comparison-based sorting algorithm. However, it has a load balancing problem. The key to make subsets well-balanced in a sample sorting algorithm is to find appropriate splitters, like in as PSRS (Parallel Sorting by Regular Sample) and DSPS. However, if they are directly ported to GPU, the overhead of generating splitters to get balanced subsets will be much larger than that of directly sorting on imbalanced subsets. So it is important to find the tradeoff point between them. Let us review the procedure of PSRS. Suppose that the size of data set is *n*. First, split the data set into *p* subsets. Then, for each subset, select (*s*−1) equidistant points as sample candidate points. Finally, merge the (*s*−1) ·*p* sample candidate points, sort them and select (*s* − 1) equidistant points as splitters. The overhead of splitter generation in PSRS is splitting the whole data set and sorting all the subsets, and it is proportional to the data size.

Our in-core sorting algorithm uses a novel strategy to select sample points. First pick up a subset from the whole data set randomly. The size of this subset is equal to (*s*− 1)·*k*·*M*, ($k \le p$), where *M* is the maximum size of array that can be sorted in share memory of one SM. Then, split the set into *k* subsets and assign *k* blocks to sort these subsets in parallel. Afterward, for each subset, select (*s*−1) equidistant points as sample candidate points. At last, merge the (*s*− 1)·*k* samples, sort them and select (*s*−1) equidistant points as splitters. The parameter *k* should be assigned at runtime depending on data size.

4. Key Optimization Methods

Here we present the detailed implementation and optimization of GPUMemSort. First, we describe the task execution engine, which can overlap data transfer with GPU computation based on pipelining. Second, we indicate how to swap buckets in chunks. Finally, we show the compensation algorithm based on optimistic mechanism.

4.1 Task Execution Engine based on Pipeline

The data transfer between CPUs and GPUs is a significant overhead in our GPUMemSort algorithm. Without optimization, more than 30% of the time would be spent on transferring data between CPUs and GPUs. But the GPU may remain idle when data transfer operations are doing. Also, the bandwidth between CPU and GPU is fully-duplex, so only 50% of the total bandwidth resource can be used if only one way is used to transfer data between CPUs and GPUs. So overlapping data transfer from CPU to GPU, GPU computation, and data transfer from GPU to CPU will bring remarkable performance improvement. Thus a task execution engine is implemented based on pipeline mechanism. First, divide a sorting task into three subtasks: CPU-GPU data transfer, kernel sorting, and GPU-CPU data transfer. Then, pipeline these three types of subtasks based on streaming with CUDA's asynchronous memory copy. Streaming maintains the dependencies, while the asynchronous memory copy parallelizes data transfer operations and sorting operation. Fig. 1 shows the comparison between the GPU classic computation pattern and our pipeline-based one.

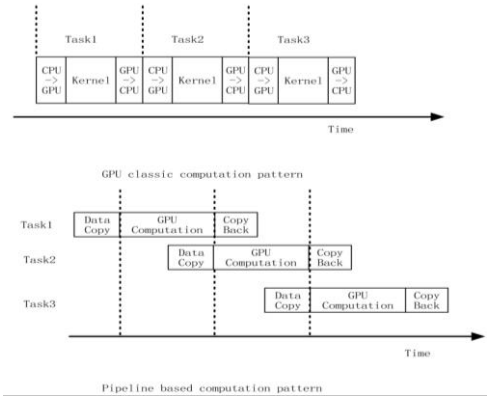


Figure1. Comparison between the GPU classic computation pattern and our pipeline based computation pattern

4.2 Implementation of Buckets Swap

In the implementation of DSPS, different buckets are swapped through network communication because different chunks are scattered in distributed memory. Pointers are used to avoid hard memory copy.

In Algorithm 1, we present our data structure of pointer arrays to swap buckets, and the procedure of transferring data from main memory to GPU global memory. Assign to each data chunks a *TransposeChunk* structure, including a vector of *TransposeBlock* to record the start address and the

size of a bucket. Then swap the start address and size in the corresponding *TransposeBlock* structures. In the coming data transfer, traverse the buckets and copy them from main memory to GPU global memory, avoiding a copy on the host.

Algorithm 1 Data Structure for buckets swap and coming data transfer algorithm

```
Struct TransposeBlock{
        int* block ptr;
        long size;
};
Struct TransposeChunk{
        TransposeBlock blocks[d];
};
procedure memcpyFromHostToDevice(Transpose
   Chunk& chunk, int* dvalue)
        offset ← 0;
        for q = 0 to d do
        TransposeBlock& tmpBlock ← chunk.blocks[q];
        cudaMemcpyHostToDevice (dvalue + offset,
        tmpBlock.block ptr, sizeof(int) * tmpBlock.size);
        offset ← offset + tmpBlock.size;
end for
```

4.3 Optimistic Mechanism based Compensation Algorithm

In the Step 4 of DSPS, *Est[i]* is calculated to record the size of elements equal to *s[i]* in sample candidate list. In the following splitting operation for chunks, it should be guaranteed that in *NS[i]*, the number of elements equal to *s[i]* is smaller than $Est[i] \cdot n/(p^2 \cdot s)$. If not, we should try to shift this element to the adjacent buckets when splitting.

To add the comparison logic above into chunks splitting module, a global variable should be maintained for each bucket to record the number of elements equal to corresponding splitter. Atomic FAA (Fetch and Add) method will be called a few times to keep consistency, thus decreasing the performance. Otherwise, the size of the chunks in the last step may exceed the threshold θ.

In order to solve this problem, we propose a novel compensation algorithm based on optimistic mechanism. Assume that $\forall i \in [0,d-1]$, the number of elements in *NS[i]* equal to *s[i]* exceeding $Est[i] \cdot n/(p^2 \cdot s)$ is a low-probability event. We add logic to Step 8 to compensate for exceptions. First, decide whether the size of each chunk is no larger than the given θ. If yes, copy this chunk to GPU global memory and sort it with our in-core algorithm. Otherwise, copy *NS[0][i]*,*NS[1][i]*,...,*NS[d-1][i]* to the GPU one by one. For *NS[j][i]*, split it into two parts: *part[j][i][0]* and *part[j][i][1]*, the former contains elements equal to *s[i]* while the latter contains the rest. Copy back the *part[j][i][1]* to the main memory, merge all the *part[j][i][1]*, $0 \le j \le d-1$ into one array, then sort this array on the GPU and write it back to the result set. Finally, fill the rest part of result set with *s[i]*. Algorithm 2 presents the pseudo code of our compensation algorithm.

Algorithm 2 Compensation algorithm on the CPU

```
# chunk: [input] TransposeChunk of chunk which will be
processed,
# splitter: [input] the corresponding splitter value
# outputBlock: [output] the pointer where results will be
written back
# splitterSize: [output] the number of elements which
equal to splitter in the chunk
procedure handleLongArrayException(const
TransposeChunk& chunk, const int splitter, int* &
outputBlock, int& splitterSize)
int boundary[d]; // splitter of each bucket
        struct TransposeChunk m chunk;
alloc memory whose size equal to d in dBoundary
    and copy boundary to dBoundary;
    for q = 0 to d do
        // handle blocks in chunk one by one.
        int* dBucketValue = NULL;
        int* dBucketOutputValue = NULL;
        const TransposeBlock& tmpBlock =
                chunk.blocks[q];
        alloc memory whose size equal to tmpBlock.size in
            dBucketValue and copy tmpBlock.block ptr to
            device memory;
        malloc tmpBlock.size length array to
             dBucketOutputValue;
        splitEquality kernel
        <<<BLOCK NUM,THREADS NUM>>>
        (dBucketValue, tmpBlock.size, splitter,
                dBoundary);
        boundary[q] ←ΣdBoundary[i];
        i∈[0, BLOCK N U M);
        prefixSum(dBoundary);
        divide kernel
        <<<BLOCK NUM,THREADS NUM>>>
             (dBucketValue, tmpBlock.size, splitter,
              dBoundary);
        copy dBucketOutputValue back to outputBlock in
             main memory;
    end for
    copy all buckets in m chunk to global memory;
    employ in-core sorting algorithm to sort them;
    copy sorted buckets back to outputBlock;
    pad the rest of outputBlock using splitter;
    free memory in device and main memory;
```

5. Experimental Results

In this section, we introduce our hardware environment and compare our in-core sorting with GPU Sample Sort, GPU Quick Sort and Thrust Merge Sort based on six different data sets and show the performance and scalability of GPUMemSort.

5.1 Hardware Environment

Our system consists of two NVIDIA GPU GTX 260 co-processors, 16GB DDR3 main memory and an Intel Quad Core i5-750 CPU. Each GPU connects to the main memory through exclusive PCIe 16X data bus, providing 4GB/s bandwidth with full duplex. Experiments have shown that data transmissions between each GPU and

main memory will not be affected too much. Also, time consumed by data transmission between GPU and main memory can be almost overlapped by GPU computation. Table 2 shows the bandwidth measurement results in different scenarios.

Table2. GPU to host bandwidth measurement

Test Cases	Single GPU	Two GPUs
Device to Host	3038.5MB/s	2785.1MB/s
Host to Device	3285.5MB/s	2802.1MB/s
Device to Device	106481.5MB/s	106377.1MB/s

The GTX 260 with consists of 16 SMs (Streaming Multiprocessor), each having 8 processors executing the same instruction on different data. In CUDA, each SM supports up to 768 threads, owns 16KB of share memory, and has 8192 available registers. Threads are logically divided into blocks and are assigned to a specific SM. Depending on how many registers and how much local memory the block of threads requires, there could be multiple blocks assigned to a SM. GPU Data is stored in 512MB of global memory. Each block can use share memory as cache. Hardware can coalesce several read or write operations into a big operation, so it is necessary to keep threads visiting consecutive memory locations.

5.2 Performance Evaluation

In this section, we first compare the performance of in-core sort, GPU Sample Sort, GPU Quick Sort and Thrust Merge Sort based on different data sets of unsigned integers. Six different types of data sets include Uniform, Sorted Zero, Bucket, Gaussian, and Staggered [11]. Fig. 2 shows the result on data of different array sizes: our in-core sorting outperforms the others because it can achieve good load balancing with little cost.

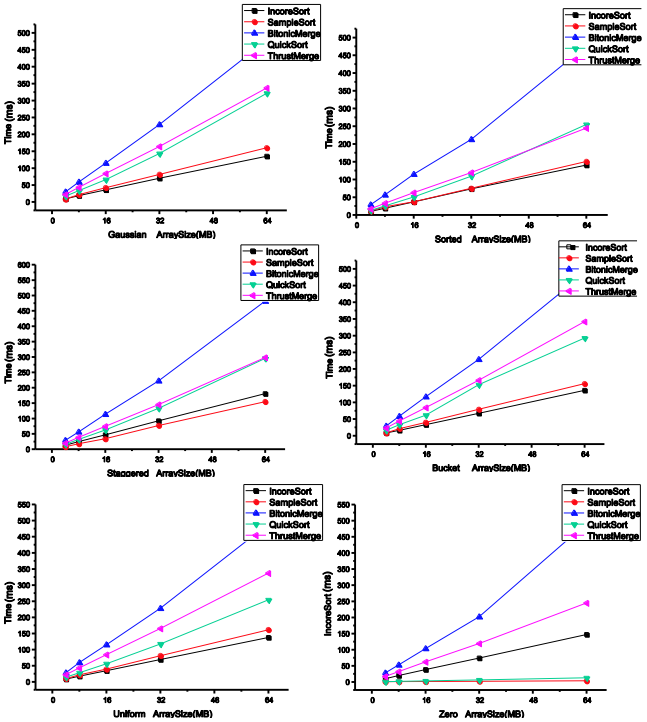


Figure2. Performance comparison between in-core sort and other existing sorting algorithms

The performance evaluation of our out-of-core algorithm on a single GPU is shown in Fig. 3, indicating that our out-of-core algorithm is robust and is capable of handling data efficiently with different distributions and sizes.

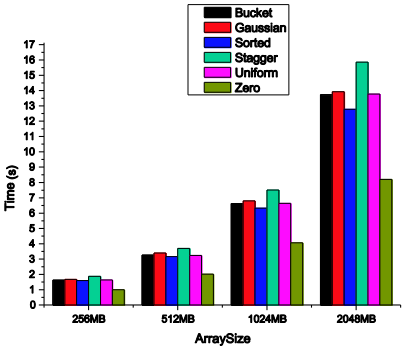


Figure3. Performances of out-of-core algorithm on different data distributions

Finally, the scalability of our out-of-core algorithm from one GPU to two GPUs is shown in Fig. 4. It is clear that our out-of-core sorting algorithm can reach near-linear speedup in two GPUs, showing that our out-of-core algorithm has good scalability when the bandwidth between main memory and GPU memory is not a bottleneck.

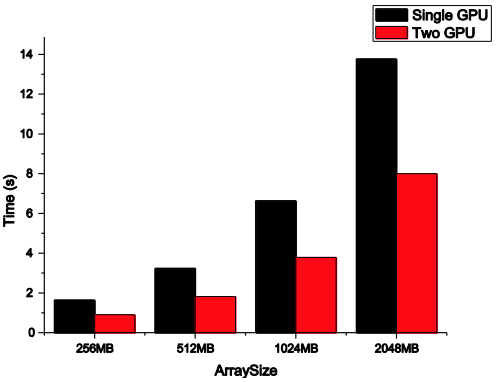


Figure 4. Performance comparison of our out-of-core scaling from one GPU to two GPUs

6. Conclusion and Future Work

In this paper, we present GPUMemSort: a high performance graphics co-processor sorting framework for large-scale in-memory data by exploiting high-parallel GPU processors. We test the performance of the algorithm based on multiple data sets and it shows that GPUMemSort sorting algorithm outperforms other multi-core based parallel sorting algorithms. A significant conclusion drawn from this work is that our GPUMemSort can break through the limitation of GPU global memory and can sort large-scale in-memory data efficiently.

We have found that for some special distributed inputs, some special sorting method can be developed by taking advantage of the special data distribution information. We will include this optimization method into our in-core sorting algorithm. The values of different parameters in our sorting method now are set based on intuition and some simple tests. But we know that those value have significant

impact on the performance and we will develop method to find suitable values for those parameters automatically. The scalability of our out-of-core algorithm only tested on two GPUs, we will do more experiments on more GPUS in the next steps to verify our algorithm. Now we use double buffers in main memory and this method will occupy more memory. We will further optimize our implementation method which needs less memory in the future. Furthermore, we will try to extend our algorithm to a GPU cluster system and optimize our algorithm on this kind of distributed heterogeneous architecture.

ACKNOWLEGEMENT

This research is supported in part by National Natural Science Foundation of China ((No. 61073008, 60773148 and No.60503039), Beijing Natural Science Foundation (No. 4082016), NSF Grants CNS-0708307, IIP-0934114, OCI-090446 (Bader), and the Center for Adaptive Supercomputing Software for Multithreaded Architectures (CASS-MT)

References

[1] NVIDIA CUDA (Compute Unified Device Architecture) http://developer.NVIDIA.com/object/cuda.html

[2] OPENCL, http://www.khronos.org/opencl/

[3] N. Govindaraju, J. Gray, R. Kumar and D. Manocha. GPUTeraSort: high performance graphics coprocessor sorting for large database management. SIGMOD , pp325-336, 2006

[4] D. R. Helman, J. JaJa, D. A. Bader. "A New Deterministic Parallel Sorting Algorithm with an Experimental Evaluation". ACM Journal of Experimental Algorithmics (JEA), September 1998, Volume 3.

[5] H. Shi and J. Schaeffer, Parallel Sorting by Regular sampling, Journal of Parallel and Distributed Computing 14, pp361–372, 1992

[6] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W-m. W. Hwu: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. PPoPP 2008: pp73–82

[7] T.Purcell, C.Donner, M.Cammarano, H.Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. ACMSIGGRAPH/Eurographics Conference on Graphics Hardware, pp41–50,2003

[8] P.Kipfer, M.Segal,and R.Westermann. Uberflow: A GPU-based particle engine. SIGGRAPH/Euro graphics Workshop on Graphics Hardware, 2004.

[9] A. Greß, and G. Zachmann, GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures, The 20th IEEE International Parallel and Distributed Processing Symposium, Rhodes Island, Greece, 2006, pp1–10.

[10] E. Sintorn, and U. Assarsson, Fast Parallel GPU- Sorting Using a Hybrid Algorithm, Journal of Parallel and Distributed Computing, Volume 68, Issue 10, pp1381–1388.

[11] D. Cederman, and P. Tsigas, A Practical Quicksort Algorithm for Graphics Processors, Technical Report, Gothenburg, Sweden, pp 246–258.

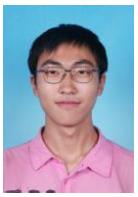
[12] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In IEEE International Parallel and Distributed Processing Symposium, Atlanta, GA, 2010

[13] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for many-core GPUs. In IEEE International Parallel and Distributed Processing Symposium, Rome, Italy, 2009.

Yin Ye got his Master degree from the Department of Computer Science and Technology at Tsinghua University, China. He received his Bachelor Degree in 2003 and then joined the High Performance Computing Institute at Tsinghua University. His research interests include Parallel and Distributed Computing, GPU Computing.

Zhihui Du is an associate professor in the Department of Computer Science and Technology at Tsinghua University, China. He received his BE degree in Computer Science from Tianjin University in 1992 and his MS and Ph.D. degrees in Computer Science from Peking University in 1995 and 1998, respectively. His research interests cover high performance computing and grid computing.

David A. Bader received his PhD degree in 1996 from the University of Maryland. From 1998- 2005, he served on the faculty at the University of New Mexico. He is a professor in computational science and engineering, a division within the College of Computing, at the Georgia Institute of Technology. His main areas of research are in parallel algorithms, combinatorial optimization, and computational biology and genomics.



Quan Yang is a senior student in the Department of Computer Science and Technology at Tsinghua University, China. He is doing research in the High Performance Computing Institute at Tsinghua University. His research interests include parallel sorting algorithm on muti-core GPU.

Weiwei Huo is an undergraduate student in the Information and Communication Engineering School at Beijing University of Posts and Telecommunications, China. She joined the High Performance Computing Institute at Tsinghua University as an intern student. Her research interests include Parallel and Distributed Computing on GPU, Parallel Algorithms and Optimizations.