GPUMemSort: A High Performance Graphic Co-processors Sorting Algorithm for Large Scale In-Memory Data [*]

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

Abstract

In this paper, we present a GPU-based sorting algorithm, GPUMemSort, which achieves high performance in sorting large-scale in-memory data by exploiting high-parallel GPU processors. It consists of two algorithms: in-core algorithm, which is responsible for sorting data in GPU global memory efficiently, and 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 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 CPU, bandwidth between CPU and memory becomes the bottleneck of large-scale computing. Many hardware vendors, such as AMD, IBM, NVIDIA integrate co-processors to offload tasks from CPU and this can alleviate the effect caused by low CPU-memory bandwidth. Meanwhile, high performance computers own much larger memory than before, so it is very important to develop efficient co-processors algorithm to deal with large-scale in-memory data.

Recently, GPU has become the best-known co-processor. It has been utilized in many different sorts of general purpose applications. GPU is suitable for highly parallel,compute intensive workloads, because of higher memory bandwidth, thousands of hardware thread contexts with hundreds of parallel compute pipelines executing programs in a SIMD fashions. 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 based on Moore's law.

Nowadays, 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, which are executed in a SIMT (single-instruction, multiple-thread) manner; threads are virtually mapped to an arbitrary number of streaming multiprocessors (SMs) through warps. There exists several types of memory, such as register, local memory, shared memory, global memory, constant memory, etc. Different type of memory owns different characteristics. Therefore, how to organize the memory access hierarchy is very important to improve 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 proposes 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 the design, detailed implementation and tests of a graphics co-processors sorting algorithm, which can sort large-scale in-memory data.

(2) We enhance GPU Sample Sort[12] algorithm, the fastest comparison-based GPU sorting algorithm. The enhanced algorithm outperforms the others because it can achieve better load balancing.

The notations in this paper are summarized in Table 1.

The paper is organized as follows. Section 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.

[*]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).

Table 1. NOTATIONS

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 divided into two categories[3]:

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

Each category has its own potential bottleneck. Partition-based algorithms have to deal with problem of how to keep load balancing among 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 $(\frac{2n}{p} - \frac{n}{p^2} - p + 1)$. A new one [4] can guarantee that each processor will have at most $(\frac{n}{p} + \frac{n}{s} - p)$ elements , where $p \leq s \leq \frac{n}{p^2}$ and s is a parameter.

2.2. GPU Programming with CUDA

The NVIDIA CUDA programming model is created for developing applications on GPU. 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 bounded 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. Greβ and Zachmann[9] introduced the GPUABiSort based on adaptive bitonic sorting. Naga K. Govindaraju[3] presented an 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, etc combines odd-even merge and two-way merge to balance the load. Satishet.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 is bad.

However, most of them 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 the two parts of GPUMemSort.The Out-of-core sorting can divide large-scale data into multiple disjointed subsets and assign them to GPU. The In-core sorting can sort the subsets efficiently.

3.1 Out-of-core algorithm

We adopt the idea of Deterministic Sample-based Parallel Sorting (DSPS) in out-of-core sorting algorithm. The idea behind DSPS algorithm is to find *s-1* samples to partition the input data set into several data chunks. Elements in *(i+1)-th* chunk are no less than those in *(i)-th* chunk. The sizes of these chunks has a deterministic upper bound. They can be put into GPU global memory by adjusting parameter *d* in the algorithm according to the value of *n*.

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

Step 1: Divide the input data set into d chunks, each contains $\frac{n}{d}$ elements. Without loss of generality, we assume that *d* divides *n* evenly.

Step 2: Copy the chunks to GPU global memory one by one, sort them by in-core algorithm. Then split the chunk into *d* buckets, bucket *j* of chunk *i* is called *Bin[i][j]*. The *x th* data in chunk *i* will be put into bucket *Bin[i]*[$\lfloor \frac{x}{d} \rfloor$]. Copy these buckets back to main memory.

Step 3: Swap buckets among chunks, $\forall i \in [0, d\text{-}1], j \in (i, d\text{-}1]$, 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-1 th* chunk, $\forall i \in [0, d\text{-}1]$,*Bin[i][d-1]* selects the *((x+1)*$\frac{n}{d^2 * s}$*) th* element as a sample candidate. *0*$\leq$ *x* $\leq$ *s-1*. The sample candidates list should contain *s*d* sample points.

Step 5: Sort the sample candidate list, select each *(k+1)*s* sample point, *k*$\in$*[0,d-2]* as *s[k]*, where *s[d-1]* is the largest. Copy the sample points array from main memory to GPU global memory.

Step 6: Copy each chunk to GPU global memory again, split the chunk into d buckets based on d sample points. The bucket j of chunk i is called *NS[i][j]* $0 \leq j \leq d-1$. 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, 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, use in-core sorting algorithm to sort it. Otherwise, we will copy *NS[0][i],NS[1][i],...,NS[d-1][i]* to GPU one by one. For *NS[j][i]*, split it into two parts called *part[j][i][0]* and *part[j][i][1]*, *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]*, $0 \leq j \leq d-1$ into one array. At last, sort this array by GPU and write it back to the result set, fill out the rest part of result set using *s[i]*.

In step 8, the threshold Θ is the maximum size of array that can be sorted on GPU. According to conduction in [5], we can easily get that: $\frac{n}{d} + \frac{n}{s} - d \leq \theta$, so $d \geq \frac{n}{\frac{\Theta}{2} - \frac{n}{2s} + \sqrt[2]{(\frac{\frac{n}{s}-\Theta}{4} + n)}}$. This means that if every chunk's size is guaranteed to be less than Θ, the number of chunks splitted in step1 must be larger than $\left\lceil \frac{n}{\frac{\Theta}{2} - \frac{n}{2s} + \sqrt[2]{(\frac{\frac{n}{s}-\Theta}{4} + n)}} \right\rceil$.

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

3.2 In-core algorithm

In-core algorithm is based on GPU Sample Sort, which is currently the fastest comparison-based sorting algorithm. However, it encounters load balancing problem. The key to make subsets well-balanced in sample sorting algorithm is to find appropriate splitters, such as PSRS(Parallel Sorting by Regular Sample) and DSPS. However, if they are directly ported to GPU, overhead of generating splitters will be even much larger than that of sorting imbalanced subsets. So it is important to find the tradeoff point between them.

Let's review the procedure of PSRS. Supposed 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.

Overhead brought by splitters generation in PSRS is splitting the whole data set and sorting all the subsets and it is proportional to the data size.

In-core sorting algorithm uses an innovative strategy to select sample points. First pick up a set from whole data set randomly. The size of set equals to $(s-1)*k*M$ $(k \leq p)$, 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. Detailed Implementation and Optimization

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

4.1 Task Execution Engine based on Pipeline

The Data transfer between CPU and GPU is a significant overhead in GPUMemSort algorithm. Without optimization, more than 30% of the time would be spent on data transfer operations between CPU and GPU. On one hand, GPU can not be fully used because it will remain idle when data transfer operations are performed. On the other hand, since the bandwidth between CPU and GPU is fully-duplex, only 50% bandwidth resource can be used at most simultaneously. 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 type of subtasks based on streaming with asynchronous memory copy technology proposed by CUDA.Streaming maintains the dependency, while asynchronous memory copy parallelizes data transfer operations and sorting operation. Fig. 1 shows the comparison between GPU classic computation pattern and pipeline-based one.

4.2 Implementation of Buckets Swap

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

In Algorithm 1, we present the data structure of pointer array to swap buckets, and the function of data transfer from main memory to GPU global memory. Assign each data chunks *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, thus hard memory copy is avoided.

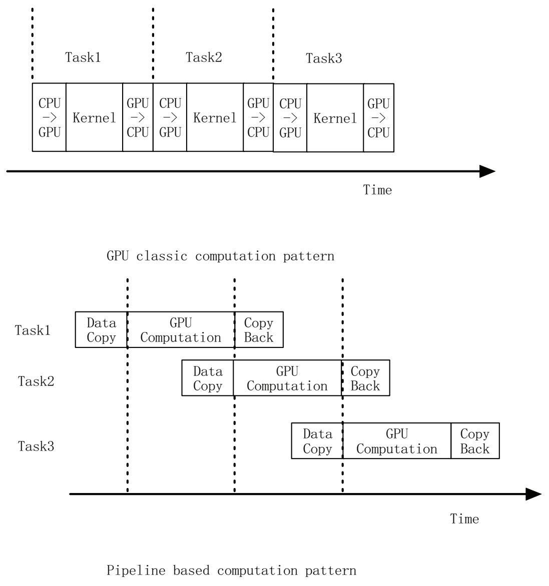

Figure 1. Comparison between GPU classic computation pattern and pipeline based computation pattern

4.3 Compensation Algorithm based on Optimistic Machanism

In the step 4 of DSPS, *Est[i]* is calculated to record the size of elements equals to *s[i]* in sample candidate list. In the coming splitting operation for chunks, it should be guaranteed that in *NS[i]*, the number of elements equal to *s[i]* is smaller than $Est[i] * \frac{n}{p^2 * s}$. 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 keep record of the number of elements equal to corresponding splitter. Atomic FAA (Fetch And Add) method will be called a few times to keep consistency, thus deteriorating the performance. Otherwise, the size of chunks in the last step may exceed the threshold θ.

In order to solve this problem, we propose an innovative compensation algorithm based on optimistic mechanism. Assume that $\forall i \epsilon [0, d\text{-}1]$, the number of elements in *NS[i]* equal to *s[i]* is no less than $Est[i] * \frac{n}{p^2 * s}$ is a small probability event and a compensation logic is added in step 8. First, judge 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 in-core algorithm. Otherwise, copy *NS[0][i],NS[1][i],...,NS[d-1][i]* to 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, then merge all the *part[j][i][1], 0≤j≤d-1* into one array, then sort this array employing GPU and write it back to the result set. At last, fill the rest part of result set with *s[i]*. In Algorithm 2, presented is the pseudo code of compensation algorithm.

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 $memcpyFromHostToDevice(TransposeChunk \& chunk, int * dvalue)$
    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
```

5. Experimental Results

In this section, we introduce our hardware environment and compare our in-core sorting with GPU SampleSort, GPU QuickSort 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 GPU 260GTX co-processors, 16GB DDR3 main memory and an Intel Quad Core i5-750 CPU. Each GPU connects 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 1 shows the bandwidth measurement results in different scenarios.

Table 2. GPU-Memory bandwidth measurement results

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

GTX 260 with CUDA consists of 16 SMs(Streaming Multiprocessor), each has 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

Algorithm 2 Compensation algorithm in CPU Side

```
# chunk: [input] TrasnposeChunk of chunk which will be processed,
# splitter: [input] the corresponding splitter value
# outputBlock: [output] the pointer of array in which results will be written back
# splitterSize: [output] the number of elements which equals 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] $\Leftarrow \Sigma dBoundary[i]; i\epsilon[0, BLOCK\_NUM)$;
    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 incore 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;
```

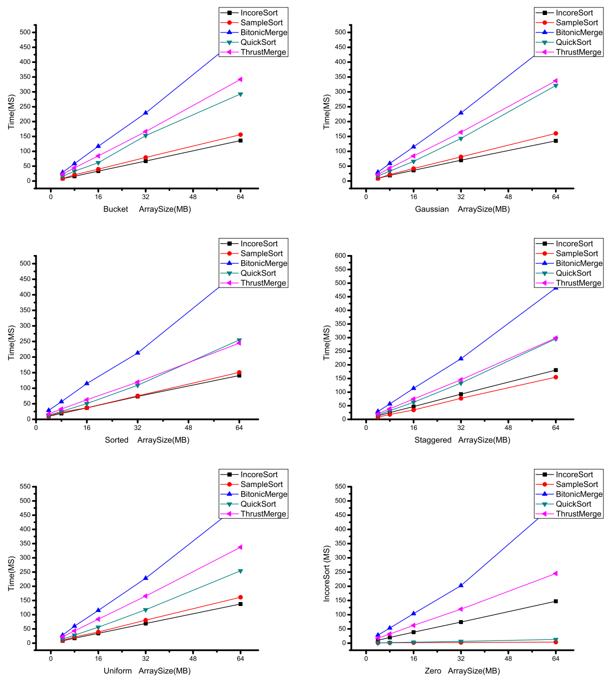

Figure 2. Performance comparison between in-core sort and other existing sort algorithms

8192 available registers. Threads are logically divided into blocks assigned to a specific SM. Depending on how many registers and how many local memory the block of threads requires, there could be multiple blocks assigned to a SM. GPU Data is stored in a 512MB global memory. Each block can use share memory as cache. Hardware can coalesce several read or write operations into a big one, so it is necessary to keep threads visiting memory consecutively.

5.2 Performance Evaluation

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

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

At last, the comparison of the out-of-core algorithm performance between single GPU and two GPUs is shown in Fig.4. It is clear that 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.

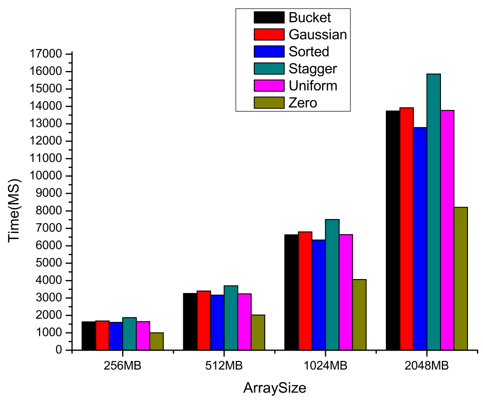

Figure 3. Performance for different data distributions of out-of-core algorithm

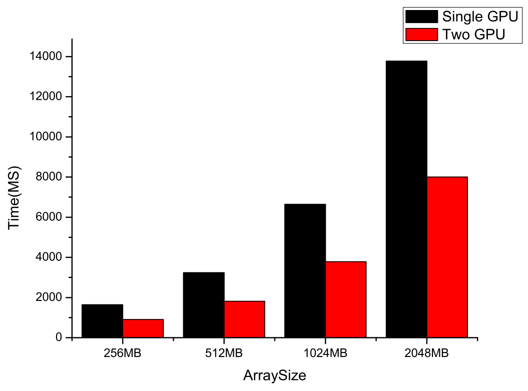

Figure 4. Performance comparison of out-of-core between single GPU and two GPUs

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.

In the future, we will try to extend our algorithm to multiple GPUs augmented cluster, implement and optimize this algorithm in distributed heterogeneous architecture. In addition, we will try to enhance in-core sorting algorithm to help GPUMemSort reach even higher performance.

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.

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*, 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, pp. 361-372, 1992

[6] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, Wen-mei W. Hwu: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. PPOPP 2008: 73-82

[7] T.Purcell,C.Donner,M.Cammarano,H.Jensen,and P.Hanrahan.Photon mapping on programmable graphics hardware. ACMSIGGRAPH/Eurographics Conference on Graphics Hardware,pages 41C50,2003

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

[9] Gre,A., and Zachmann, 2006,GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures, The 20th IEEE International Parallel and Distributed Processing Symposium, Rhodes Island, Greece, pp. 1-10.

[10] Sintorn, E., and Assarsson, U. 2007, Fast Parallel GPU-Sorting Using a Hybrid Algorithm, Journal of Parallel and Distributed Computing, pp. 1381-1388.

[11] Cederman, D., and Tsigas, P., 2008, 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,2010 (currently available at http://arxiv1.library.cornell.edu/abs/0909.5649).

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

[14] Felix Putze, Peter Sanders, Johannes Singler. The Multi-Core Standard Template Library (Extended Poster Abstract). Symposium on Principles and Practice of Parallel Programming (PPoPP) 2007