An efficient sorting algorithm with CUDA

Article *in* Journal of the Chinese Institute of Engineers · November 2009

DOI: 10.1080/02533839.2009.9671578

6 authors, including:

Jing Qin
The Hong Kong Polytechnic University
315 PUBLICATIONS **15,545** CITATIONS
SEE PROFILE

Junping Zhao
Chinese PLA General Hospital (301 Hospital)
13 PUBLICATIONS **122** CITATIONS
SEE PROFILE

Pheng-Ann Heng
The Chinese University of Hong Kong
696 PUBLICATIONS **29,863** CITATIONS
SEE PROFILE

AN EFFICIENT SORTING ALGORITHM WITH CUDA

Shifu Chen*, Jing Qin, Yongming Xie, Junping Zhao, and Pheng-Ann Heng

ABSTRACT

An efficient GPU-based sorting algorithm is proposed in this paper together with a merging method on graphics devices. The proposed sorting algorithm is optimized for modern GPU architecture with the capability of sorting elements represented by integers, floats and structures, while the new merging method gives a way to merge two ordered lists efficiently on GPU without using the slow atomic functions and uncoalesced memory read. Adaptive strategies are used for sorting disorderly or nearly-sorted lists, large or small lists. The current implementation is on NVIDIA CUDA with multi-GPUs support, and is being migrated to the new born Open Computing Language (OpenCL). Extensive experiments demonstrate that our algorithm has better performance than previous GPU-based sorting algorithms and can support real-time applications.

Key Words: parallel sorting, parallel merging, CUDA.

I. INTRODUCTION

Fast and robust sorting algorithms are essential to many applications where ordered lists are needed. With the progress of general-purpose computing on GPUs (GPGPU), a lot of efforts have been dedicated to developing high-performance GPU-based sorting algorithms, especially after programmable vertexes and fragment shaders were added to the graphics pipeline of modern GPUs. The early GPU-based sorting algorithms directly employ graphics application programming interfaces (APIs). For examples, Purcell *et al*. (2003) reported an implementation of bitonic merge sort on GPUs. Kipfer *et al*. (2004) presented an improved bitonic sort using the name odd-even merge sort. (Greβ and Zachmann, 2006) introduced the GPUABiSort, a sorting algorithm based on the adaptive bitonic sorting technique proposed by Bilardi *et al*. (1989). Govindaraju *et al*. (2005) implemented a library named GPUSort with the capability of sorting floating points with both 16 and 32-bit precision. Since the graphics APIs cannot give developers direct access to the native instruction set and memory of the parallel computing units on graphics devices, one of their main disadvantages is they cannot efficiently handle the sorting of elements in structure, and the computing capacity of graphics processors was not fully exploited.

As the acronym for Compute Unified Device Architecture, CUDA is a parallel computing architecture developed by NVIDIA Corporation. Compared to traditional GPGPU techniques, CUDA has several advantages, such as scattered reads, shared memory, faster downloads and readbacks to or from the GPU, and full support for integer and bitwise operations. These features make CUDA an efficient parallel computing architecture, which can easily drain the computing capacity of modern GPUs. A full introduction to programming with CUDA can be found in (NVIDIA Corporation, 2008).

Recently, some CUDA-based sorting algorithms have been proposed. For instances, Erik Sintorn *et al*. (2007) introduced a CUDA-based hybrid algorithm which combines bucket sort and merge sort, but can only sort floats as it uses a float4 for internal merge sort to achieve high performance (sorting integers would be possible if one replacs float4 with int4). Cederman *et al*. (2008) proposed the implementation of Quicksort in CUDA, but the performance of this algorithm is sensitive to the distribution of the input list. University of California and NVIDIA collaboratively developed

*Corresponding author. (Email: sf.chen@siat.ac.cn)

S. Chen and P. A. Heng are with the Shenzhen Institute of Advanced Integration Technology, Chinese Academy of Science/The Chinese University of Hong Kong, China.

J. Qin, Y. Xie and P. A. Heng are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, China.

J. Zhao is with the Institute of Medical Informatics, Chinese PLA General Hospital & Postgraduate Medical School, China.

a fast CUDA-based sorting algorithm named Global Radix Sort reported by Harris *et al*. (2008). However, as it uses the bit information of the elements, the complexity of this algorithm is in scale with the number of bits of the elements, and explodes when being employed to sort floats or even structures.

Although these CUDA-based algorithms show high performance in some cases, the lack of ability to sort floats or structures limits their application to resolve many practical problems. In this paper, we propose a new fast and flexible sorting algorithm with CUDA, which not only has high performance, but also has the capability to sort elements in various data types, and is flexible to deal with disorderly lists and nearly-sorted lists respectively. In this algorithm, we first cut the input list into slices, and then combine these slices into buckets. The buckets are then sorted on a single stream multi-processor, and the outputs are imploded into an ordered list.

In addition, a novel merging method is provided for multi-GPUs support, and for sorting large lists, which cannot be sorted by one pass due to hardware limitations. Merging two ordered lists into one list is a trivial job on a CPU, but it's traditionally difficult on GPUs, as it's not appropriate for the GPU programming model and is hard to parallelize. We alter this problem by applying two level task divisions to give the possibility for this problem to parallelize. Level I division is block-based (multi-processor based), while level II division is thread-based. Shared memory is used to avoid uncoalesced memory access, which will seriously reduce the memory bandwidth and affect the performance.

We conducted a series of experiments to compare the performance of our algorithm to other CUDA-based algorithms and CPU-based STL::qsorts(), the results demonstrated that our algorithm has better performance than all the previously evaluated GPU-based sorting algorithms, such as GPU Quicksort, Hybrid sort and Global Radix Sort, and shows acceleration over STL qsort with a factor of 10 to 20 depending on different graphics cards.

II. SORTING ON GPU: ALGORITHM OVERVIEW

Task division is critical for parallel computing. Generally a good task division should distribute the original task to sub tasks of nearly the same size, and keep the sub tasks independent so that they can be assigned to different processors with little or no communication. We divide the input lists into ascending ordered buckets so that they can be sorted independently and can be merged to be entirely sorted lists after they are all sorted. In order to get an almost equal bucket division, we first divide the input list into slices which have much fewer elements than a bucket, and then combine those slices into buckets. The bucket sequences needs to meet two criteria:

1) Each element in bucket i should be smaller than the elements in bucket $i + 1$, assuming that we're going to obtain an incremental sequence (we always assume this in this paper).
2) Each bucket has no more elements than a fixed size M; otherwise it will be split into sub-buckets.

M is determined by the graphics hardware, indicating the maximum threads that can be contained in a thread block. Typically it is 512 for NVIDIA graphics cards.

1. Main Steps

The four main steps of this algorithm are briefly introduced as follows:

1) **Slice division:** Firstly we split the input list into small slices, each slice is expected to have fewer elements than M , and each element in slice i should be smaller than the elements in slice $i + 1$. This part will be discussed in detail in section III.3.
2) **Slice merge:** Then the slices will be merged into buckets. This operation should make each bucket have as close as possible to, but no more than M elements, unless that the first slice has already more elements than M. This part will be discussed in section III.4.
3) **Internal bitonic sort:** Thirdly the buckets will be sorted individually in a parallel manner. In order to obtain the best performance, the data of these buckets will be loaded into the shared memory, and written back to the global memory after the sorting is completed. This part will be discussed in section IV.
4) **Recursion:** Finally, if there exist some buckets containing more elements than M, so they are not fully sorted, each bucket will be considered as a new input list and return to the entrance of the sorting pipeline. This part is implemented as a recursion. According to our experiments, the recursion procedure is rarely called for. Mostly we get no buckets or just one bucket exceeding M while sorting uniformly distributed or sorting a Gaussian distributed list with 10M elements. So this recursion won't raise a performance problem.

III. BUCKET DIVISION

The bucket division contains two steps: slice division and slice merge. Slice division has three substeps: (1), find the maximum and minimum element of the input list; (2), calculate a step width for the slices; (3), calculate the slice index of each element and assign each element to a corresponding slice.

1. Find the Maximum and Minimum

Firstly, a reduction method introduced by

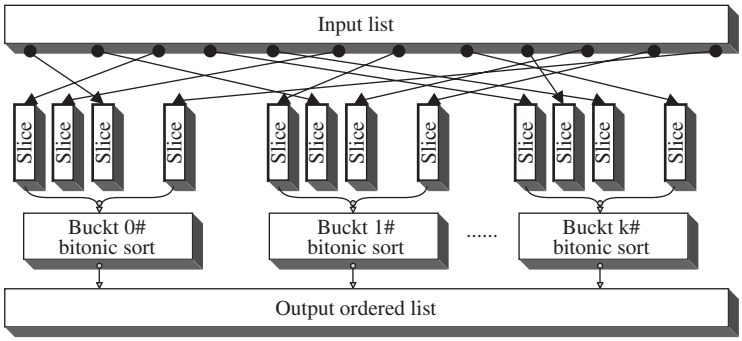

Fig. 1 Pipeline of our CUDA-based sorting algorithm. The input list is firstly split into slices; these slices are then merged to buckets. Finally, an internal bitonic sort is performed for each bucket

Harris (2008) is used to find the maximum and minimum elements of the input list in parallel.

2. Calculation of Step Width and Slice Count

Once we get the maxValue and minValue, we can calculate the step width of slices using a parameter E, which represents the expected slice size. The step width can be calculated as Eq. (1):

$$stepWidth = (maxValue - minValue) * E/n, \quad (1)$$

where n is the size of the input list. Then, using step width of slice list, we could calculate the slice count directly.

3. Assign Elements to Slices

In this step, we use four arrays: (1), sliceSizes, storing the size of each slice; (2), sliceOfElements, storing the slice index of each element; (3), offsetInSlice, storing the offset of each element in its corresponding slice; (4), sliceOffsets, storing the global offset for each slice.

The process of assigning elements to slices is to fill sliceOfElements with the correct slice index for each element, to calculate the size of each slice, and to find the offset of each element in its corresponding slice.

For old graphics hardware, it's not easy to make this process efficient, as it's hard to obtain consistent data. Fortunately, for modern GPUs, synchronization mechanisms such as atomic functions are available. By employing these atomic functions, threads actually run serially when they attempt to increase the size of slice i. Although this would decrease the parallelism and bring some unfavorable factors to the performance, it is still critical for some applications.

With these atomic functionalities, we describe the process of assigning elements to slices as the following pseudo code shows:

Algorithm 1: Slice Division
```
FOREACH element i in parallel
  sliceIndex=(value[i]-minValue)/stepWidth
  offset=atomicInc(sliceSizes[sliceIndex])
  sliceOfElements[i]=sliceIndex
  offsetInSlice[i]=offset
END
```

The function atomicInc() increases the value of given memory unit by 1 atomically, and returns the old value of this unit. The old values can be taken as the offset of this element directly, as it is initially 0, increases as sliceSize increases, and is never the same for different elements in the same slice.

After the slice list being completely built, an algorithm called SCAN introduced by Harris *et al*. (2007) is used to calculate the prefix sum in parallel. The array sliceSizes is used to build the array sliceOffsets while employing SCAN. After this procedure is done, each element of sliceOffsets will have the global offset for the corresponding slice.

4. Merge Slices to Generate Bucket

In this step, we use two arrays, which are bucketSizes and bucketOffsets. The bucketSizes stores the size of each bucket, and the bucketOffsets stores the offset of each bucket.

The following pseudo codes give a description of this process:

Algorithm 2: Slice Merge
```
FOR i=0 to sliceCount-1
  IF bucketSize[bucketCount]+sliceSize[i]>M AND
     bucketSize[bucketCount]!=0
  generateNewBucket()
  END
  addSliceToBucket(bucketCount,i)
END
```

The function addSliceToBucket() handles the process of adding a slice to a bucket, where the bucketSize would be updated.

After we complete the slice division and slice merge, we can build a new list. The following pseudo codes give a description:

Algorithm 3: move elements to new list
FOREACH element *i* in parallel
  sliceIndex=(value[i]-minValue)/stepWidth
  sliceOffset=sliceOffsets[sliceIndex]
  globalOffset=sliceOffset+offsetInSlice[i]
  newList[globalOffset]=oldList[i]
END

---

5. Flexible Strategies for Flat or Nearly Sorted List

A list is flat if its maxValue is equal to its minValue. We can discover if it is a flat list or not by comparing the maxValue and minValue. If they are the same, the program can exit right away. For a nearly sorted list, we have to solve at least two problems. One is how we know whether the input list is nearly sorted. The other is how to handle it if we know it is nearly sorted.

We define

$$disorderDegree = \frac{\sum_{k=1}^{n} |a_i - a_{i-1}|}{maxValue - minValue}. \qquad (2)$$

The *disorderDegree* is the degree of disorder of the input list, and a_i is the value of element *i*. A threshold *C* is used to determine whether the input list is nearly sorted or disorderly. If *disorderDegree* < *C*, then the input list is considered to be nearly sorted, otherwise disorderly. In our experiments, *C* = 10 works well for sorting elements from 1M to 24M.

The parallel reduction algorithm is also used here to calculate the *disorderDegree* effectively. It uses very little time, less than 2% of the time used by the sorting process. The user can simply skip this step if he already knows whether the input list is nearly sorted or not.

While doing the slice division, we use different strategies to calculate the index of elements for nearly ordered lists and disordered lists, respectively. For disorderly lists, we use

$$elementIndex = blockDim * blockId + threadId. \qquad (3)$$

While *threadId* is the thread index, *elementIndex* indicates the element that the current thread will handle, the *blockDim* is the size of the blocks, and *blockId* is the index of the active block. In this case, the threads in the same block will read consecutive elements so that the memory access is coalesced and efficient.

And for nearly sorted lists, we use

$$elementIndex = blockCount * threadId + blockId. \qquad (4)$$

Although this involves discontinuous memory access and causes a non-coalesced problem, the time used in this part drops significantly. In our experiments, sorting 24M nearly- sorted floats on a Geforce 280 GTX card, takes nearly 1050 ms while using the first strategy, but only takes 420 ms while using the other.

IV. BITONIC SORT IN GPU

Since the expected size *E* of each slice is much smaller than *M* (in our implementation, *M* is 512 by default, and *E* is 80), the vast majority of the buckets should have elements totaling less than but close to *M*. The buckets which have elements totaling more than *M* will be on the way to a recursion.

The internal bitonic sort includes three steps.

1. Data Copying and Padding

Firstly, we copy the elements from global memory to shared memory to reduce the global memory accesses. Then we pad the bucket to make it have *M* elements if it is not full. The padding operation makes the coming compulation much more efficient, as *M* is always a power of 2.

There is more than one method to pad elements in non-full buckets, but the most simple and safe way is to use the greatest number, which is considered as the infinite in computer usage.

2. Internal Bitonic Sort

After data copying and padding, we've got *M* elements stored in shared memory. We then use an internal bitonic sort which is one of the fastest sorting networks. The parallel bitonic sorting works effectively only when the length is a power of 2, that's why we do the padding. A detailed introduction to parallel bitonic sorting may be found in (Kider, 2005).

3. Write Back the Result

This step is pretty simple, we just have to make sure that the padding will be removed.

V. EFFICIENT MERGING ON GPU

We have developed an efficient GPU-based merging method, so that a longer list can be cut into sub-lists which have length less than 30M, and can

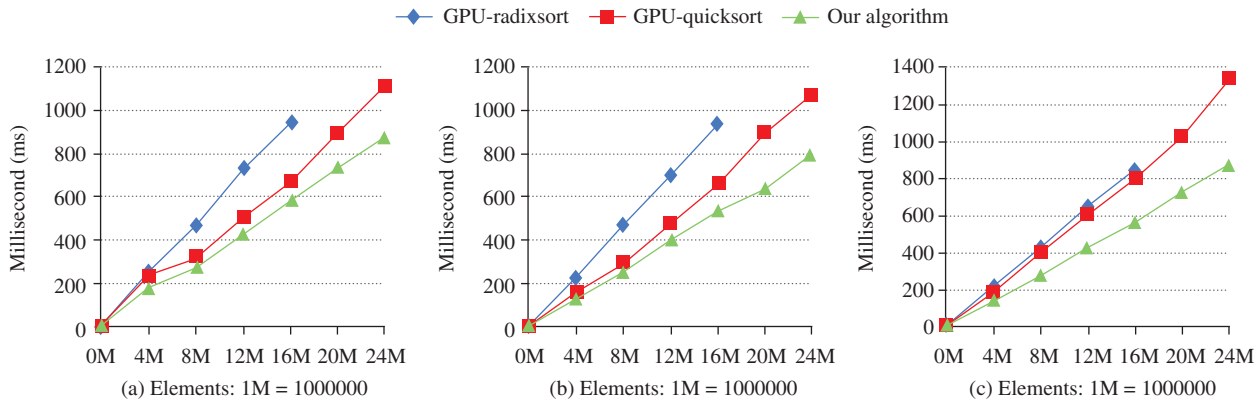

Fig. 2 The timing performance of GPU-RadixSort, GPU-Quicksort and our algorithm while sorting 4M to 24M integral elements: (a) Sorting uniformly distributed, disorderly list, (b) Sorting uniformly distributed, nearly-sorted list, (c) Sorting Gaussian distributed, disorderly list

be sorted independently and merged into an entirely sorted list using our developed merging method.

Two lists can be merged into one list trivially in CPU by a simple scan. However, to resolve the same problem in parallel architecture is difficult. If we apply the same solution like CPU's on GPU, only one thread will be active , while other processors will just be waiting. To avoid wasting computing resources, we employed a divide and conquer technique to resolve this problem in parallel.

1. Task Division for List Merging

For two ordered sequences A and B, the problem is to produce a sequence C, which is also ordered and is exactly a combination of A and B. We first cut A into small segments as $A_0, A_1 \cdots A_{n-1}$. Each segment has a size of N (2048 in our implementation) except the last one. We'll get $n + 1$ elements for this division as $e_0, e_1 \cdots e_k \cdots e_n$, while e_0 and e_n represent the first and the last element of sequence A.

For these $n + 1$ elements in A, we apply a binary search to find their positions in B. The search is independent for each element, so can be executed in parallel. After this process, we'll get $n + 1$ pivots as $p_0, p_1 \cdots p_k \cdots p_n$ representing ranks of $e_0, e_1 \cdots e_k \cdots e_n$ in B.

2. Merge in one Block

After the pivots are prepared, an in-block merge is launched for each segment of A. Each thread in a block is responsible for the merging task of a fixed number of continuous elements. For instance, in our implementation, one thread in block (segment) k is accounting for four elements by default, we name them as x_0, x_1, x_2 and x_3. Firstly we find the position of x_0 in B using a binary search between p_k and p_{k+1}, and then we can find the position of x_1 using a linear search from the position of x_0, the same for x_2 and x_3. After we get the four positions, we can simply calculate the global offsets of x_0 to x_3 and the elements between their two positions in sequence B. Finally moving these elements to global results, using these global offsets is a trivial task.

VI. MULTI-GPUS SUPPORT

With this merging algorithm, the presented sorting algorithm can sort more than 30M elements. Since the sorting process for each sub-list is independent, we also applied multi-GPUs support for the proposed sorting algorithm. We implemented one thread context for each CUDA-enabled device to support multi-GPUs acceleration. However, since creating, switching, synchronizing and killing threads are not time free, the multi-GPUs based implementation won't be faster than single GPU based ones when sorting small lists. We will show the performance comparison between single and multi-GPUs.

VII. PERFORMANCE EVALUATION

We conducted a series of experiments to evaluate the time usage of our algorithm. The experiments were performed on three graphics cards: a GeForce 9600GT card, a GeForce 280 GTX card and a Geforce 295 GTX card, while the CPU used in these experiments was a Quad Q6600 at 4 ×2.4 GHz. Results are shown in Fig. 2 to Fig. 5.

Firstly, we compare our algorithm with GPU-Quicksort,and GPU RadixSort (from CUDPP V1.0), as they are also implemented with CUDA. Note that both the current implementations of GPU-Quicksort and GPU RadixSort can only sort elements of integers, while our algorithm is not limited to sorting integers. Experimental results show that our algorithm achieves better

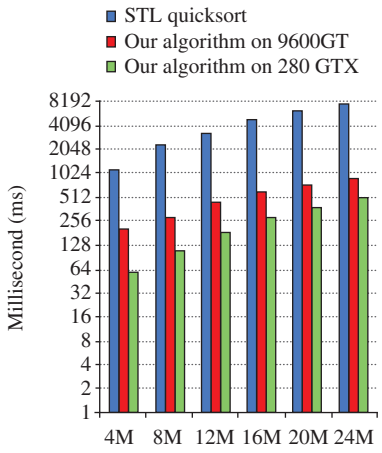

Fig. 3 The time use of STL Quicksort and our algorithm, while sorting 4M to 24M elements. The list is uniformly distributed and disorderly: (a) Sorting list in float, (b) Sorting list in float4

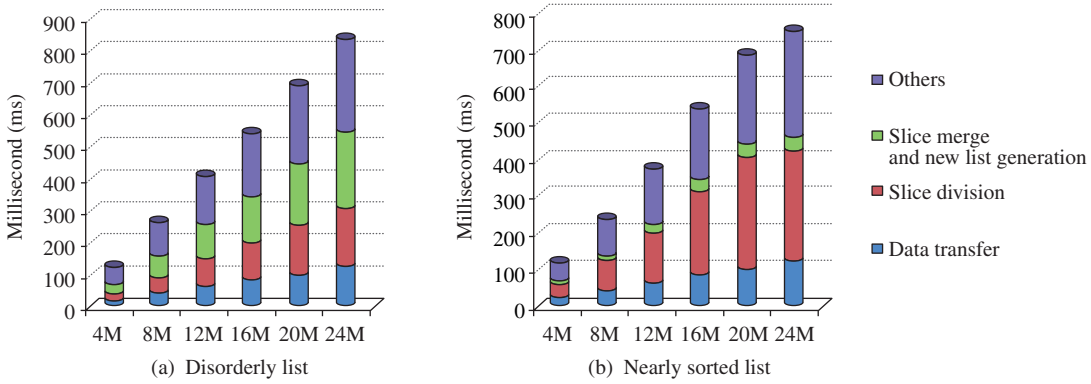

Fig. 4 The time consumption of each part of our algorithm. We could find out the bottlenecks while sorting nearly-sorted or disorderly ones: (a) Sorting nearly-sorted list in float, (b) Sorting disorderly list in float

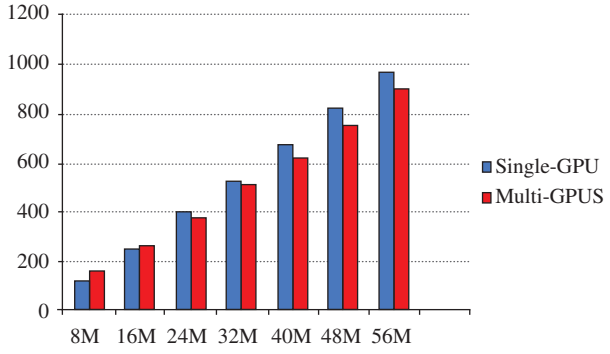

Fig. 5 The performance comparison between implementation on single and multi- GPUs while using Geforce 295 GTX for this experiment. We can find that multi-GPUs based implementation is not faster than single GPU based ones while element amount is less than 24M

performance than both the GPU-Quicksort and the GPU RadixSort. Fig. 2 shows the comparison of our algorithm with GPU-Quicksort and GPU RadixSort. The results include time cost of data transfer between host and graphics device. Since the GPU- RadixSort can only support sorting less than 16M elements, time uses for it sorting 20M and 24M elements are not presented.

Secondly, we make a comparison of our algorithm and STL quicksort. The results show our algorithm is 10 to 20 times faster than STL quicksort (released version and highly optimized). Results are shown in Fig. 3.

Figure 4 shows the time consumption of each part of our algorithm, while sorting nearly-sorted or disorderly lists. Finally Fig. 5 shows the performance comparison between implementation on single and multi-GPUs

VIII. CONCLUSIONS

We've proposed an efficient GPU-based sorting algorithm, which can handle the sorting of elements in integers, floats, or structures. An efficient merging

method is also presented to support sorting large lists and to support multi-GPUs based implementation. The experiments demonstrated our algorithm achieves performance 10 to 20 times faster than STL quicksort and has better performance than the GPU-Quicksort and GPU-RadixSort.

ACKNOWLEDGEMENT

The work described in this paper was supported by the National Natural Science Foundation of China (Grant No. 60873067).

REFERENCES

Bilardi, G., and Nicolau, A., 1989, "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines," *SIAM Journal on Computing*, Vol. 18, No. 2, pp. 216-228.

Bilardi, G., and Nicolau., A., 1989, "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines," *SIAM Journal on Computing*, Vol. 18, No. 2, pp. 216-228.

Cederman, D., and Tsigas, P., 2008, "A Practical Quicksort Algorithm for Graphics Processors," *Technical Report*, Gothenburg, Sweden, pp. 246-258.

E., Blelloch., C., Greg Plaxton., Charles, E. Leiserson., Stephen, J. Smith., Bruce, M., Maggs., Marco, Zagha., 1998, "An Experimental Analysis of Parallel Sorting Algorithms," *Theory of Computing Systems*, Vol. 31, No. 2, pp. 135-167.

Greβ, A., and Zachmann, G.., 2006, "GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures," *The 20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, pp. 1-10.

Govindaraju, N. K., Raghuvanshi, N., and Manocha, D., 2005, "Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors," *ACM SIGMOD International Conference on Management of Data*, Illinois, USA, pp. 611-622.

Harris, M., 2008, "Optimizing Parallel Reduction in CUDA," *Technical Report*, California, USA, pp. 1-38.

Harris, M., Sengupta, S., and Owens, J. D., 2007, "Parallel Prefix Sum (Scan) with CUDA," *GPU Gems 3, Addison-Wesley, Reading*, Miami, USA.

Joseph T. Kider., 2005, "GPU as a Parallel Machine: Sorting on the GPU," *Lecture of University of Pennsylvania*, Pennsylvania, USA.

Kapasi, U. J., Dally, W. J., Rixner, S., Mattson, P. R., Owens, J. D., and Khailany, B., 2000, "Efficient Conditional Operations for Data-Parallel Architectures," *The 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, California, USA, pp. 159-170.

Kipfer, P., Segal, M., and Westermann, R., 2004, "UberFlow: A GPU-Based Particle Engine," *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, Grenoble, France, pp. 115-122.

NVIDIA Corporation., 2008, "NVIDIA CUDA Programming Guide," *Technical Report*, California, USA, pp. 1-125.

Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W., and Hanrahan, P., 2003, "Photon Mapping on Programmable Graphics Hardware," *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, California, USA, pp. 41-50.

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

Manuscript Received: Jun. 26, 2009
Revision Received: Aug. 07, 2009
and Accepted: Oct. 15, 2009