A Fast and Flexible Sorting Algorithm with CUDA

Shifu Chen[1], Jing Qin[2], Yongming Xie[2], Junping Zhao[3], and Pheng-Ann Heng[1,2]

[1] Shenzhen Institute of Advanced Integration Technology, Chinese Academy of Sciences/The Chinese University of Hong Kong
sf.chen@siat.ac.cn
[2] Department of Computer Science and Engineering, The Chinese University of Hong Kong
{jqin,ymxie,pheng}@cse.cuhk.edu.hk
[3] Institue of Medical Informatics, Chinese PLA General Hospital & Postgraduate Medical School
kyc301@yeah.net

Abstract. In this paper, we propose a fast and flexible sorting algorithm with CUDA. The proposed algorithm is much more practical than the previous GPU-based sorting algorithms, as it is able to handle the sorting of elements represented by integers, floats and structures. Meanwhile, our algorithm is optimized for the modern GPU architecture to obtain high performance. We use different strategies for sorting disorderly list and nearly-sorted list to make it adaptive. Extensive experiments demon- strate our algorithm has higher performance than previous GPU-based sorting algorithms and can support real-time applications.

Keywords: Parallel sorting algorithm, CUDA, GPU-based sorting algorithm.

1 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 vertex and fragment shaders were added to the graphics pipeline of modern GPUs. The early GPU-based sorting algorithms directly employed graphics application programming interfaces (APIs). For examples, Purcell [1] reported an implementation of bitonic merge sort on GPUs. Kipfer [2][3] presented an improved bitonic sort with the name of odd-even merge sort. Greβ [4] introduced the GPUABiSort, a sorting algorithm based on the adaptive bitonic sorting technique proposed by Bilardi [5]. Govindaraju [6] implemented a library named GPUSort with the capability of sorting floating point in both 16 and 32-bit precision. Although encouraging

A. Hua and S.-L. Chang (Eds.): ICA3PP 2009, LNCS 5574, pp. 281–290, 2009.

performance improvements are reported in these algorithms, one of the main disadvantages of them is they cannot efficiently handle the sorting of elements in structure, since the APIs cannot give developers access to the native instruction set and memory of the parallel computational elements in GPUs, therefore the computing capacity of GPU cannot be fully exploited.

CUDA is a parallel computing architecture developed by NVIDIA. Comparing with traditional GPGPU techniques, CUDA has several advantages, such as scattered reads, shared memory, faster downloads and readbacks to or from the GPU, and fully 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 of programming with CUDA can be found in [7].

Recently, some CUDA-based sorting algorithms have been proposed. For instances, Erik Sintorn [8] introduced a CUDA-based hybrid algorithm which combines the bucket sorting and merge sorting, but can only sort floats as it uses a float4 for internal merge sorting to achieve high performance (sorting integers would be possible if replace float4 with int4). Daniel Cederman [9] 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 a fast CUDA-based sorting algorithm named Global Radix Sort [10]. However, as it uses the bit information of the storage unit, the complexity of this algorithm is in scale with the number of bits, and explodes while being employed to sort floats or structures.

Although these CUDA-based algorithms show high performance in some cases, the lack of ability of sorting floats or structures limits their application to many practical problems. In this paper, we propose a novel fast and flexible sorting algorithm with CUDA, which not only has high performance, but is able to handle the sorting of elements represented by various data types. In addition, flexible strategies are provided for sorting disorderly list and nearly-sorted list, respectively. Extensive experiments demonstrated that our algorithm has higher performance than previous GPU-based sorting algorithms, such as GPU Quicksort, Hybrid sort and Global Radix Sort, and showed its capability of supporting real-time applications.

2 Algorithm Overview

The main idea of this algorithm is: we first split the input list to buckets, and then sort each bucket concurrently in one stream multi-processers.

The buckets sequence need to meet two criteria:

1) Each element in the 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 to 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 the NVIDIA cards.

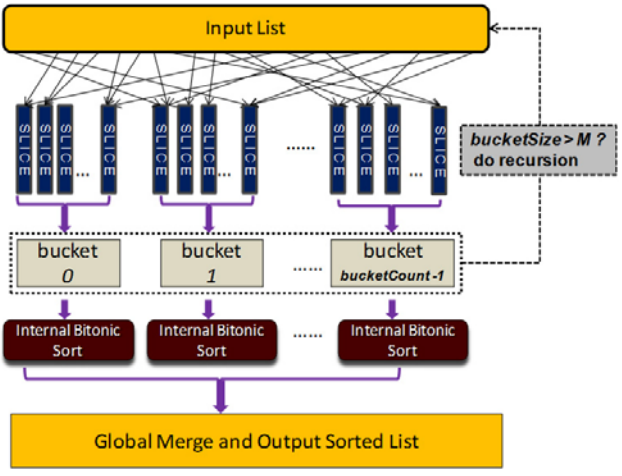


Fig. 1. The pipeline of our CUDA-base sorting algorithm. The input list is firstly split into slices, and these slices are then merged to buckets. Finally, an internal bitonic sort is performed for each bucket.

2.1 Main Steps

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

Slice division. Firstly we split the input list into small slices, each slice is expected to have less elements than M , and each element in the slice i should be smaller than the elements in slice i + 1. This part will be discussed in detail in section 3.3.

Slice merge. Secondly 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, except that the first slice has already make the bucket exceeded. This part will be discussed in section 3.4.

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 wrote back to the global memory after the sorting is completed. The whole sorting process of one bucket is entirely executed on single stream multi-processers. Since the shared memory supplies horizontal accesses, this process is very fast. This part will be discussed in section 4.

Recursion. Finally, if there exists some buckets contain more elements than M, so that it's not fully sorted, the 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 actually rarely called. Mostly we get none or just one bucket exceeded, while sorting uniformly or Gaussian distributed list with 10M elements. So it won't raise a performance problem.

3 Bucket Division

The bucket division contains three steps: slice division and slice merge. Slice division has three sub-steps: 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 corresponding slice.

3.1 Find the Maximum and Minimum

Firstly, a reduction method introduced by Mark Harris [11] is used to find the maximum and minimum elements of the input list.

3.2 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 Ep. 1:

$$\text{stepWidth} = (\text{maxValue} - \text{minValue}) * E/n \tag{1}$$

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

3.3 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 the slice.

For old graphics hardware, it's not easy to make this process efficient, as it's hard to obtain data consistent. Fortunately, for the modern GPUs, the 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 out 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 following pseudo code shows:

Algorithm 1: Slice Division

```
foreach element i in parallel
   sliceIndex=(int)(elementValue[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 a same slice.

After the slice list being completely built, an algorithm called SCAN [13] is used to calculate the prefix sum in parallel. Mark Harris also introduced the CUDA implementation of SCAN at [17]. 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 corresponding slice.

3.4 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.

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 if
   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=(int)(elementValue[i] - minValue)/stepWidth;
   sliceOffset=sliceOffsets[sliceIndex];
   globalOffset=sliceOffset + elementOffsetInSlice[i];
   newList[globalOffset] = oldList[i];
end
```

4 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 less than but close to M . The buckets which have elements more than M will be on the way to a recursion.

The internal bitonic sort includes three steps.

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. This makes the coming up operations much more efficient, as M is always a power of 2 .

There are more than one methods to pad elements into those non-full buck- ets, but the most simple and safe way is to use the biggest number, which is considered as infinite in computer.

Internal bitonic sort. After data copying and padding, we've got M elements stored in shared memory. We then use an internal bitonic sort introduced by W. Christopher [12]. Bitonic sort 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 of parallel bitonic sorting could be found in [15].

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

5 Bottleneck and Optimizations

Memory access is usually considered to be the biggest bottleneck for sorting algorithms, especially for the GPU-based ones.

For our algorithm, the memory access bottleneck could probably be found at two sections: one is the slice division described at Algorithm 1, the other is the new list generation which is described at Algorithm 3.

When the input list is nearly sorted (either ascending or descending), it comes the worst situation. Since the atomic functions are actually run serially at hardware level, if the input list is nearly sorted, the nearby elements may be assigned to the same slice, so assigning an element to a slice could be very time consuming as there might be a long queue waiting to access the same memory unit. We use a simple method to check the input list is nearly sorted or not to avoid this worst situation. This will be discussed at 5.1.

Moving elements to new places is not efficient, because there exists discontinuous reads and writes, both result in a non-coalesced memory access. We utilize the texture mechanism to alleviate this problem. This will be discussed at 5.2.

5.1 Flexible Strategies for Flat or Nearly Sorted List

A list is flat while its maxValue is equal to its minValue. We could know it is a flat list or not by comparing the maxValue and minValue, if yes, the program could exit right away.

For nearly sorted list, we have to solve at least two problems . One is how we know the input list is nearly sorted or not, the other is how to handle it if we already know that.

We define:

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

The disorderDegree is on behalf of the degree of disorder of the input list, and a_i is the value of element i. A threshold C is then used to determine 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 [11] is also used here to calculate the disor- der-Degree effectively. It only takes less than 2% of the total time. And the user can simply skip this step if they already know 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:

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

While threadId is the thread index, elementIndex indicates the element that 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:

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

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

5.2 Utilize the Texture Mechanism

As mentioned at section 3.4 and 3.5, they are two arrays, which are bucketSizes, sliceOffsets. These arrays don't need to be modified after built, and accesses on them could be very heavy. These two characteristics meet the features of texture mechanism well.

Since the textures memory space is cached, this step will be accelerated a lot if the texture fetches hit the cache frequently.

6 Performance Evaluation

We conduct a series of experiments to evaluate the timing performance of our algorithm. The experiments are performed on two graphics cards, one is a GeForce 9600GT card and the other is a GeForce 280 GTX card, while the CPU employed in these experiments is a Quad Q6600 at 4x2.4 GHz. Results are shown in Fig.2 to Fig. 4.

Firstly, we compare our algorithm with GPU-Quicksort,and GPU Radix-Sort, as they are also implemented with CUDA. Note that both the current implementation 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 algrithm achieves better 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 The GPU-RadixSort can

only support sorting less than 16M elements, time use of its sorting 20M and 24M elements are not presented.

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

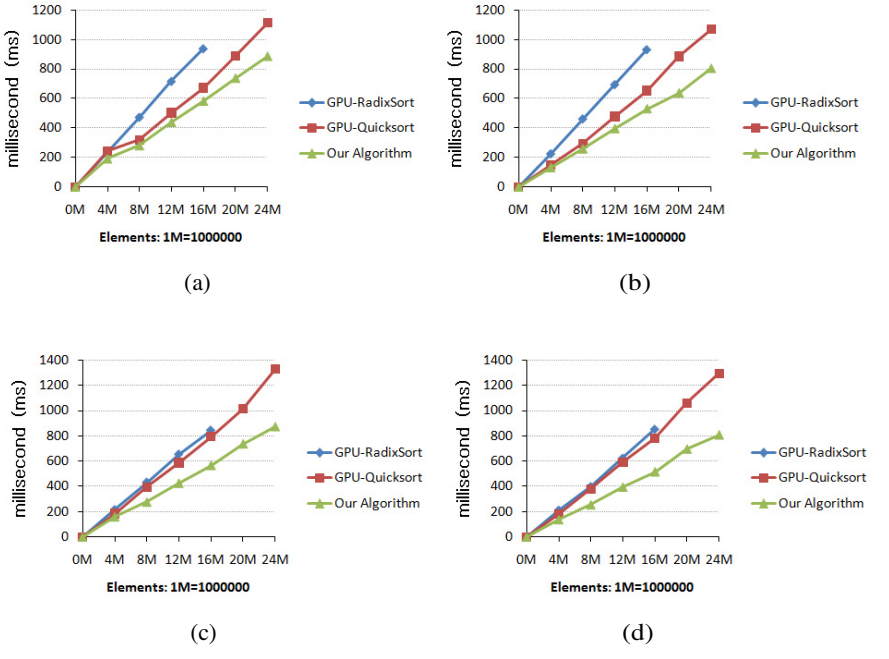


Fig. 2. The timing performance of GPU-RadixSort, GPU-Quicksort and our algorithm while sorting 4M to 24M integral elements. (a) Sorting uniformly distributed, disorderly list. (b) Sorting uniformly distributed, disorderly list. (c) Sorting Gaussian distributed, disorderly list. (d) Sorting Gaussian distributed,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

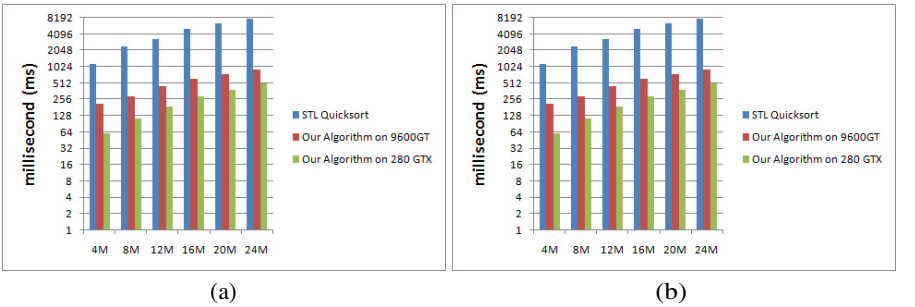
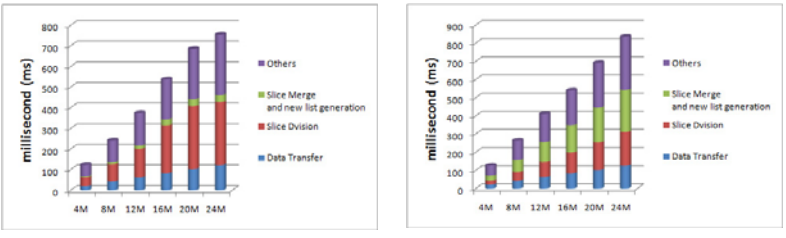


Fig. 3. The time use of STL Quicksort and our algorithm, while sorting 4M to 24M elements. The list is uniformly distributed and disorderly. (a) Sorting list in float. (b) Sorting list in float4.

At last, Fig. 4 shows the time consumption of each part of ou algorithm, while sorting nearly-sorted or disorderly lists. Finally Fig. 5 shows an application of our algorithm for real-time bleeding simulation.



(a) (b)

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

Fig. 5. In a SPH-based real-time blood simulation project, our sorting algorithm was used for neighbor particle searching

7 Conclusion

We've proposed a fast and flexible GPU-based sorting algorithm. It can handle the sorting of elements in integers, floats, or structures. It's a combination of bucket sort and internal bitonic sort. The experiments demonstrated our algorithm achieves a 10x to 20x acceleration over STL quicksort. Our algorithm also has higher performance than the GPU Quicksort and GPU RadixSort.

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

References

1. Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W., Hanrahan, P.: Photon Mapping on Programmable Graphics Hardware. In: Proceedings of the ACM Siggraph Eurographics Symposium on Graphics Hardware (2003)
2. Kapasi, U.J., Dally, W.J., Rixner, S., Mattson, P.R., Owens, J.D., Khailany, B.: Efficient Conditional Operations for Data-parallel Architectures. In: Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture, pp. 159–170 (2000)

3. Kipfer, P., Segal, M., Westermann, R.: UberFlow: A GPU-based Particle Engine. In: Proceedings of the ACM Siggraph/Eurographics Conference on Graphics Hardware, pp. 115–122 (2004)
4. Greβ, A., Zachmann, G.: GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (2006)
5. Bilardi, G., Nicolau, A.: Adaptive Bitonic Sorting. An Optimal Parallel Algorithm for Shared Memory Machines. SIAM Journal on Computing 18(2), 216–228 (1989)
6. Govindaraju, N.K., Raghuvanshi, N., Manocha, D.: Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 611–622 (2005)
7. NVIDIA Corporation. NVIDIA CUDA Programming Guide (2008)
8. Sintorn, E., Assarsson, U.: Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In: Workshop on General Purpose Processing on Graphics Processing Units (2007)
9. Cederman, D., Tsigas, P.: A Practical Quicksort Algorithm for Graphics Processors. Technical Report 2008-01, Computer Science and Engineering Chalmers University of Technology (2008)
10. Harris, M., Satish, N.: Designing Efficient Sorting Algorithms for Manycore GPUs. NVIDIA Technical Report (2008)
11. Harris, M.: Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology (2008)
12. Blelloch, E., Greg Plaxton, C., Leiserson, C.E., Smith, S.J., Maggs, B.M., Zagha, M.: An Experimental Analysis of Parallel Sorting Algorithms (1998)
13. Harris, M., Sengupta, S., Owens, J.D.: Parallel Prefix Sum (Scan) with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, Addison-Wesley, Reading (2007)
14. Bilardi, G., Nicolau, A.: Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines. SIAM J. Comput. 18(2), 216–228 (1989)
15. Kider, J.T.: GPU as a Parallel Machine: Sorting on the GPU, Lecture of University of Pennsylvania (2005)
16. Knuth, D.: Section 5.2.4: Sorting by merging. In: The Art of Computer Programming, Sorting and Searching, vol. 3, pp. 158–168 (1998) ISBN 0-201-89685-0
17. Harris, M.: Parallel Prefix Sum(Scan) with CUDA (2008)