# A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess

Keliang Zhang
School of Computer Science,
Fudan University
No. 825 Zhang Heng Road
Shanghai, 201203, P.R. China
klzhang@fudan.edu.cn

Baifeng Wu
School of Computer Science,
Fudan University
No. 825 Zhang Heng Road
Shanghai, 201203, P.R. China
bfwu@fudan.edu.cn

***Abstract*—Radix sort is an important sorting algorithm which is widely used in applications such as binary search and database. The most important advantage of radix sort is its time complexity is O($n$), lower than other sorting algorithms based on comparison operation. However, a factor that hampers its application is long execution time of its loop body. In this paper, based on data level parallelism idea, we present a new parallel radix sort algorithm. In this algorithm, each iteration of loop can be fully parallelized via thounds of concurrent threads, eliminating the original performance bottleneck. Furthermore, it is a scalable algoirthm, which means the performance of algorithm can be linearly improved with the increase of core number of modern many-core processors. Experiment results show the efficiency of the algorihtm.**

## I. Introduction

Sorting algorithms (e.g. radix sort, quick sort) are widely used in modern applications. The time complexity of radix sort is O($n$), obviously superior to quick sort. However, due to radix sort's long execution time of loop body, quick sort is usually the choice in actual applications, such as the sort function implementation [9] in the STL library for Microsoft Developer Studio (we use STL sort for short). One method for eliminating the performance obstacle of radix sort algorithm is parallelization.

Generally, there are plenty of data dependency in radix sort. To resolve these data dependency, one approach is to use large amount of mutex operations. But the approach usually remarkablely reduces the degree of parallism, and limits the scalability of the algorithm. Another approach is the privatization solution used in [1], [6], [14] which eliminates the data dependency by allocating private histogram bins to each processing element. The size of memory needed by histogram bins will increase with the number of processing elements. The most weakness of privatization is that the size of memory for histogram bins is relate to the number of processing elements, which makes the exact size of memory needed by histogran bins diffcult to be determined, even leading to excessive memory usage. This drawback may prohibit performance promotion of the algorithm especially when the number of processing elements is very large. This paper proposes a new parallel approach, which first converts the original data array $input$ to preprocess array $pInput$ by preprocess method based on bucket partition, then the preprocess array $pInput$ is sorted by the counting sort (the commonly used sub-sorting procedure of radix sort). Our method can eliminate all data dependency as the privatization method does. It divides the original data sequence into multiple intervals which is monotonically increasing, which means that the elements belonging to different intervals are different. Then it deal with the preprocess array $pInput$ with multiple iterations. And each iteration processes a group of independent elements from all intervals with one element coming from one interval. This makes each iteration become a fully data parallel process because all element values are completely different, which means each iteration may only parallelly access the different positions of histogram bins without any access conflict. Such a massive data level parallel radix sort algorithm is attractive in acutal applications because of its fast exectuion time, especially in the case when the number of processing elements is large and the scale of data is huge.

During our experiments, the hardware and software platforms are the AMD ATI Radeon HD 5850 GPU processors and OpenCL respectively. Experiment results show that the performance of our bucket partition preprocess parallel approach of radix sort algorithm is one order of magnitude higher than the performance of STL sort under AMD Athlon X2 64 Dual Core CPU processor. And it can obtain the comparable performance with others parallel sorting algorithms [3], [4], [5], [6], [10], [12], [13] published recently years.

## II. Related Work

Some previous vector machines (e.g. Cray Y-MP, CM-2) do not support conflict detection mechanism. In those machines data dependency can only be solved via the software method which affects the performance, programmability, and scalability. Blelloch et al. [1] present a privatization parallel approach based on vector machine CM-2, it avoids the conflict problem via allocating private histogram bins to each processing element, and then combining those private histogram bins through reducing operation. Zagha et al. [14] propose a same parallel approach based on vector machine Cray Y-MP with some hardware-optimized techniques to fully exploit the hardware resources, such as virtual processors, loop raking, and processor memory layout. Ha et al. [6] implement the algorithm under modern GPUs with some optimization methods. There are two problems for privatization solution. One is that the number of required histogram bins is $p$ times that required by the sequential algorithm where $p$ is the number of processing elements. Since $p$ is related to physical architecture, the size of memory required by histogram bins can not be determined. Furthermore, as the number of the processing elements in a parallel processing system increases, the size of memory required by histogran bins may exceed the amount the system can provide and result in memory allocation failure. The other is the reduction operation used to combine the private histogram bins. The reduction operation is a time consuming operation for a parallel processing system, and becomes the major performance bottleneck in the privatization solution as Lee et al. [8] point out. Although most modern GPUs support atomic operations with the ability of solving data access conflict. But such operation limits the memory access, affects memory access coalescing, results in a serious inefficient memory bandwidth useage.

Sintorn et al. [13] present a hybrid GPU-sorting algorithm. First, they split the input list into enough sublists by bucket sort sub-procedure. Then they use the merge sort sub-procedure to sort these sublists. The merge sort sub-procedure consists of two parts which are 2-by-2 bitonic sort for float4 vectors and 4-by-4 odd-even merge sort for two internally sorted float4 vectors. The sorted output list of bucket sort has two properties. One is that the values of elements in the $i$th bucket are all smaller than those in the $j$th buckets when $i$ is smaller than $j$. The other is that the elements of each bucket are sorted. Sintorn et al. [13] only make use of the first of the above two properties, so we rename it as bucket partition preprocess in this paper.

The advantage of the method proposed in this paper is that the amount of histogram bins can be determined, which is the same as that of sequential radix sort algorithm, that is the radix value $r$. Comparing to the radix sort algortihm in NVIDIA sample library, although our method has no advantage in performance, the simple, clear, independent algorithm structure makes the method be more easy to be adopted, modified, optimized according to actaul applications.

## III. Bucket Partition Preprocess

In this section, we will give an simple review of preprocess method based on bucket partition, we recommend readers to refer [13] for the details of the bucket partition preprocess.

As shown in Fig. 1, the original array $input$ is converted to the logical two domains array $pInput$, which is stored in a continuous memory space.

To acheive a load balance among parallel threads, the number of elements in each row of array $pInput$ should be approximately same. As pointed out in [13], it can be solved by selecting good pivot points for the bucket partition preprocess to improve the probability of getting equal sub-list length by computing histogram. Given that the value range of array $input$ is [0, $r$], the distribution of its value is uniform,

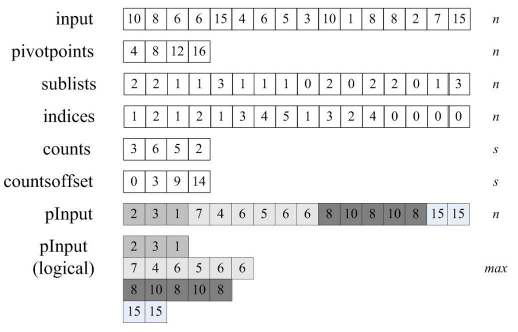

input 10 8 6 6 15 4 6 5 3 10 1 8 8 2 7 15 n
pivotpoints 4 8 12 16 n
sublists 2 2 1 1 3 1 1 1 0 2 0 2 2 0 1 3 n
indices 1 2 1 2 1 3 4 5 1 3 2 4 0 0 0 0 n
counts 3 6 5 2 s
countsoffset 0 3 9 14 s
pInput 2 3 1 7 4 6 5 6 6 8 10 8 10 8 15 15 n
pInput (logical) 2 3 1
7 4 6 5 6 6 max
8 10 8 10 8
15 15

Fig. 1. The original array $input$ is converted to preprocess array $pInput$ by bucket partition preprocess with some accessorial array.

then the initial pivot points can be chosen simply as a linear interpolation of range [0, $r$], and it is generally not needed to adjust anymore. The length of each array can be found in Fig. 1. Experiment results show that when the $input$ is 32-bits unsigned integer, the best performance will be reached for our parallel radix sort algorithm when $r$ is $2^{16}$ and $s$ is $2^{10}$.

We use both the bucket partition preprocess and counting sort to replace the counting sort used in original radix sort algorithm as the sub-sorting procedure. Such a parallel optimization may lead to problems when the radix sort needs repeatedly calling of sub-sorting procedures, because the original order of elements with the same value can not be kept. Therefore, the whole sub-sorting procedure of radix sort is not stable anymore. It is a necessary requirement for the sub-sorting procedure of radix sort, although there is no problem in [13] because it does not need the stable property. For example, as shown in Fig. 1, there are two elements have the same value 10 in array $input$, and their index are 1 and 10 respectively, but it is undetermined that the last order of two elements pair will be $<idx_1, idx_{10}>$ or $<idx_{10}, idx_1>$ in the $3th$ row of array $pInput$.

To solve the problem, in our method we use the insert sort algorithm to process the intermediate result from the sub-sorting procedure. Although the insert sort algorithm is generally considered as a slow sorting method with high time complexity, for a almost sorted data sequence (elements are not correctly sorted only when they have the same high digits) such as the above intermediate result, the execution of the algorithm can be very fast with linear time complexity. Experiment results show that the performance declines only about 10%.

## IV. Hybrid Parallel Approach of Radix Sort Algorithm

Radix sort algorithm is different from those sorting algorithms based on comparing operation, such as merge sort, quick sort, and insert sort. It is a linear time complexity O($n$) based on random access machine (RAM) computational model. It breaks the input keys to $b$-digits, and utilizes a stable sub-sorting procedure to sort one digit at one-pass iteration. The whole radix sort algorithm sorts the input keys by executing the sub-sorting procedure $b$ passes. Counting sort is generally used as the sub-sorting procedure because it is stable.

In our method we take a different approach, we use preprocess counting sort as the sub-sorting algorithm which contains both bucket partition preprocess and counting sort. The bucket partition preprocess is used to converts the array $input$ to the preprocess array $pInput$ which has a particular property that the values of elements in the $i$th row are all smaller than those in the $j$th row when $i$ is smaller than $j$. The preporcess array $pInput$ is then further sorted by the counting sort in parallel.

The parallel implementation of bucket partition preprocess is based on [13], and the parallel implementation of scan-histogram-bins sub-procedure of counting sort is based on [7], [12]. The details of histogram-keys and rank-and-permute sub-procedure of counting sort is in the following sections.

### *A. Parallel of Histogram-Keys Sub-procedure*

Histogram-keys sub-procedure is used to compute the value distribution of the digits sequence extracted from input keys. For example, assuming the radix is $r$ and the range of histgoran bins is [0, $r$), then the histogram-keys sub-procedure is used to compute the size of digits which have value between 0 (including) and $r$ (excluding).

Although the number of processing elements of many-core GPUs is about $10^3$ (we use notation $p$ to denote), the number of actually concurrent threads (or work-item) can be $10^5$ or even high (we use notation $w$ to denote) based on context switching and single instruction multiple data (SIMD) architecutre. These threads are light-weight with hardware context switching, leading to almost zero-overhead thread switching, and huge concurrent thread number.

The method to fully exploit the huge number of concurrent threads is to try to operate $w$ elements simultaneously. The simplest approach for operating $w$ elements simultaneously is partitioning the array $input$ to many consecutive sub-sequence which has length $w$ as shown in Fig. 2. Then the histogram-keys sub-procedure updates the $histbins$ by the consecutive $w$ elements at once iteration. However, the data access conflict problem of writting the same position of array $histbins$ will occur when there is more than one elements have the same digits in the sub-sequence.

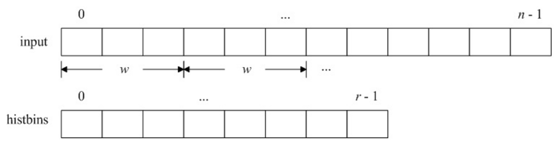


Fig. 2. A simplest but not correct partition method of array input

We use bucket partition preprocess to convert the array $input$ to the preprocess array $pInput$ as shown in Fig. 3. Then we can avoid the data access conflict problem during processing the whole column of array $pInput$ in parallel. We use notation $s$ to denote the number of rows in array $pInput$, because it is the same as the length of array $pivot\!-\!points$ as pointed out in previous section. For processing the whole column of array $pInput$, it is only needed to set the size of global work items to $s$, and only $max$ (we use notation $max$ to denote the maximum size of rows) times of iterations to deal with the whole array $pInput$. The pseudocode of parallel histogram-keys is shown in Fig. 4.

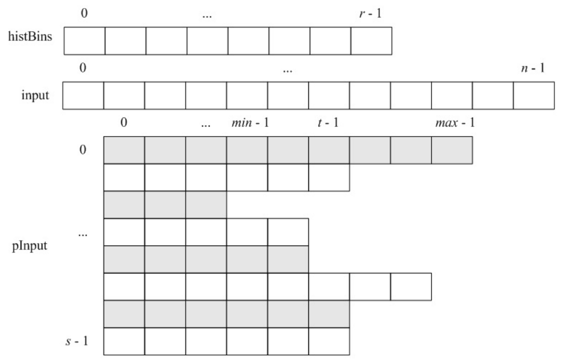
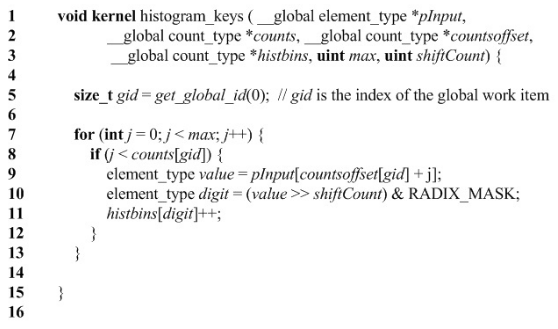


Fig. 3. Bucket partition preprocess of uniform distribution



```
void kernel histogram_keys ( __global element_type *pInput,
        __global count_type *counts, __global count_type *countsoffset,
        __global count_type *histbins, uint max, uint shiftCount) {

    size_t gid = get_global_id(0);  // gid is the index of the global work item

    for (int j = 0; j < max; j++) {
        if (j < counts[gid]) {
            element_type value = pInput[countsoffset[gid] + j];
            element_type digit = (value >> shiftCount) & RADIX_MASK;
            histbins[digit]++;
        }
    }

}
```

Fig. 4. Pseudocode for parallel histogram-keys sub-procedure

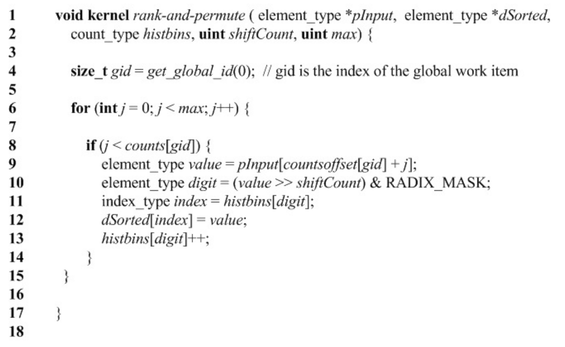
As shown in Fig. 3, the general partition of array $pInput$ is not perfect distribution. We use the notation $min$ to denote the minimum size of rows, and use notation $t$ to denote the average size of rows, so $t$ = $n/s$. For achieving load balance, the value $(t - min)$ and $(max - t)$ should be smaller than a threshold value. Because there are a lot of columns which do not reach maximum size, we can idle the correspondence work items. For example, assuming the $i$th (equals $gid$ here) row of $j$th column is null, then the $i$th work item should be idle. We can use a test statement $j < counts[gid]$ to determine whether the $i$th row of $j$th column is null or not. And the index of element in the $i$th row and the $j$th column is denoted by $countsoffset[gid]$ + $j$, it means the index of element in array $pInput$ processed by the $i$th work item at the $j$th iteration.

From parallelism viewpoint, we should select a big value $s$ for the kernel histogram-keys, because GPUs can support more than $10^5$ work items to execute concurrently. When $r$ is $2^{16}$, the experiment result shows that the performance of histogram-keys sub-procedure when $s$ is $2^{16}$ is over 10 times of that when $s$ is $2^{10}$. However, the performance of bucket partition preprocess will reach the best value when $s$ is $2^{10}$ as pointed out in [13], and it is consistent with our experience result. Because the time of preprocess counting sort is a major time consuming procedure in the bucket partition preprocess, occupying more than 70% execution time during the bucket partition preprocess as pointed out in later section, so we set $s$ to $2^{10}$ in our implementation of whole radix sort algorithm.

The time complexity of histogram-keys sub-procedure is O($max$) because we only need max pass iterations. And generally speaking, $max$ = O($r$), $s$ = O($r$), $r$ = O($w$), so O($max$) = O($t$) = O($n/s$) = O($n/r$) = O($n/w$).

### B. Parallel of Rank-and-Permute Sub-procedure

After histogram-keys and scan-histogram-bins sub-procedure, the array $histbins$ contains the final positions of all digits in array $dSorted$, if there is more than one digits has the same value, then array $histbins$ contains the position of the first one among those digits. Then the sub-procedure rank-and-permute can place the elements of array $pInput$ into the correct position in array $dSorted$. It can be parallel implemented using the same method of parallel histogram-keys sub-procedure, that is, placing the whole column of array $pInput$ to the correctly position in array $dSorted$. Because all digits are different in the same column of array $pInput$, so the value of $histbins[digit]$ (we use notation $index$ to denote it, and it means the position of digit in array $dSorted$) are different, so the two updating statements (the line 12 and 13 of Fig. 5) will not result in data access conflict because they will not write to the same position of array $dSorted$ and $histbins$ respectively. The pseudocode of parallel rank-and-permute sub-procedure is shown in Fig. 5.



```
1    void kernel rank-and-permute ( element_type *pInput,  element_type *dSorted,
2      count_type histbins, uint shiftCount, uint max) {
3
4      size_t gid = get_global_id(0);  // gid is the index of the global work item
5
6      for (int j = 0; j < max; j++) {
7
8        if (j < counts[gid]) {
9          element_type value = pInput[countsoffset[gid] + j];
10         element_type digit = (value >> shiftCount) & RADIX_MASK;
11         index_type index = histbins[digit];
12         dSorted[index] = value;
13         histbins[digit]++;
14       }
15     }
16
17   }
18
```

Fig. 5. Pseudocode for parallel rank-and-permute sub-procedure

The size of global work items of kernel rank-and-permute is $s$ too, which is also the number of rows of array $pInput$. The algorithm only needs $max$ times of iterations because it can process the whole column of array $pInput$ concurrently. It extracts the corresponding digit sequences from the input keys, then gets the position in array $dSorted$ by value $index$, and writes the keys to the position index of array $dSorted$. Finally it uses the statement $histbins[digit]$++ to deal with the case when more than one keys have the same digit.

## V. Experience Result and Performance Analysis

Our algorithm was tested on an AMD Athlon X2 64 Dual Core CPU System with a single AMD ATI Radeon HD 5850 GPU card. The key parameters of our hybrid parallel approach of radix sort algorithm were chosen from many experiments to achieve the best performance: the radix $r$ was set to $2^{16}$, and the number of buckets $s$ was set to $2^{10}$. If the value range of input keys is [0, $2^{32}$), the whole parallel radix sort algorithm (called two-pass radix sort) requires two-pass preprocess counting sort procedure and one pass insert sort procedure which is used to solve the non stable problem of preprocess counting sort procedure. However, as pointed out in [1], the radix sort is not a best choice for large keys, because the multi-pass sub-sorting algorithm will cause multiple drop in performance comparing with one-pass sub-sorting procedure. We also tested the performance of our hybrid radix sort algorithm for the case with small keys, which only needs one-pass preprocess counting sort and it does not need insert sort algorithm anymore. The value range of one-pass preprocess counting sort (or called one-pass radix sort) is [0, $2^{16}$) when the radix $r$ is set to $2^{16}$. The input keys of both two-pass and one-pass hybrid radix sort are generated by rand function of the library for Microsoft Developer Studio 2008. Meanwhile, for comparison the selection of CPU version algorithm is the STL sort. In order to highlight the parallel processing effect, the time for transferring data between host memory and device memory was not taken into account during comparison. In all of our experiments, the measurements were done several times and the average running time was calculated, and the results of two-pass radix sort and one-pass radix sort are shown in Fig. 6.

As shown in Fig. 6, the performance of both one-pass radix sort and two-pass radix sort are much better than that of STL sort. Actually, our two-pass radix sort gains more than 5X speedups than STL sort, and one-pass radix sort obtains more than 20X speedups than STL sort.

Actually, as shown in Fig. 7, the percentage of time occupied by bucket partition preprocess in the one-pass preprocess counting sort is about 70%, and that of counting sort is only about 30%.

We also compare the performance of our one-pass hybrid radix sort algorithm with others parall radix sort algorithms on NVIDIA GPU platform. The result shows the performance of our algorithm is comparable with those approaches in [2], [3], [6], [10], [11], [12], [13]. Among those parallel approaches, all other algortihms have comparison performances with the algorithm in [13]. Since our algorithm use the same bucket partition preprocess sub-procedure as the algorithm in [13], and the ratio value $rat$ (where we define the ratio value $rat$ be the time of bucket partition preprocess sub-procedure devided by the whole execution time of algorithms) is about 70% - 80% whereas that of the algorithm in [13] is about 29.6%. It can be concluded that our algorithm is faster than the algorithm in [13]. Finally, we can find that the performance of our algorithm is better than [4], [5], and comparable with [2], [3], [6], [11],

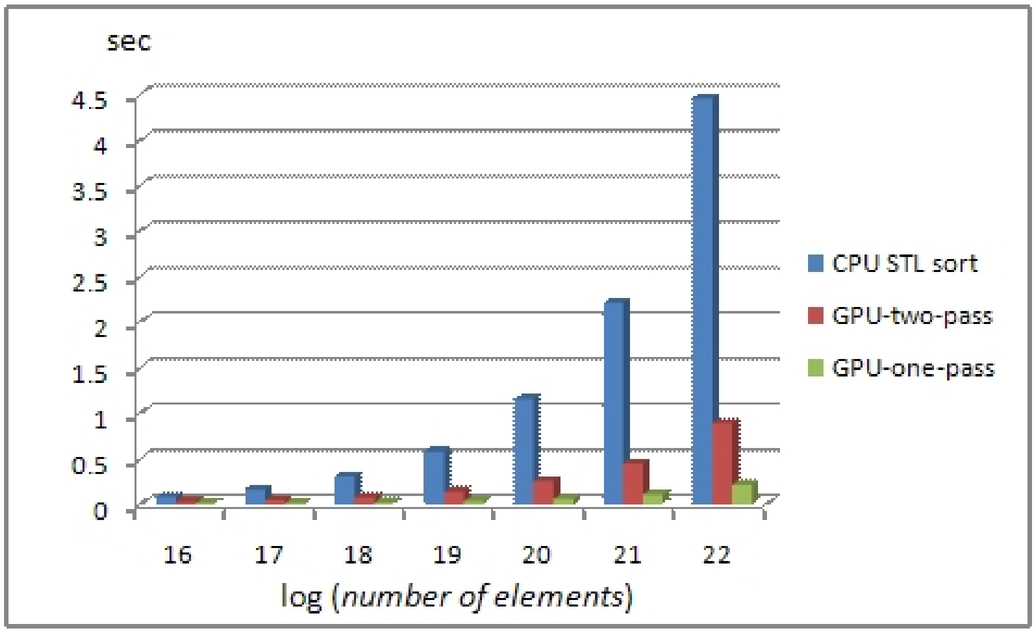


Fig. 6. The performance comparison between our radix sort and STL sort

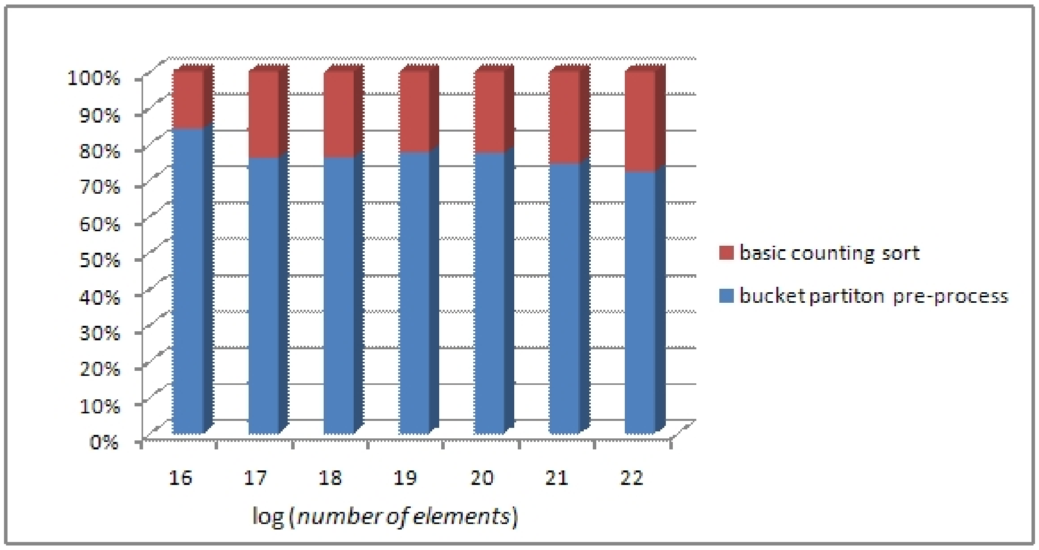


Fig. 7. The time proportion of pre-process counting sort

[12], [13] and the merge sort of [10].

## VI. Conclusion and Future Work

In this paper, we present a hybrid parallel radix sort algorithm based on bucket partition preprocess. Our parallel approach has two advantages. First, comparing with the method based on atomic operations, our solution completely eliminates the data dependency of radix sort and has much better scalability. Second, our approach only requires fixed size of memory for histogran bins, while the size of memory for histogran bins required by other parallel implementations of radix sort based on privatization solution is related to the number of processing elements.

In the future work, we will implement our parallel algorithm based on CUDA framework and make a more straightforward performance comparison vervus those parallel sorting algorithm based on CUDA. And we intend to consider the parallel implementation based on GPU cluster.

## Acknowledgments

We would like to thank anonymous reviewers for insightful feedbacks that improve the quality of this paper. Meanwhile we appreciate AMD, Inc. for offering the AMD ATI Radeon HD 5850 GPU card. This work is funded by Huawei Innovation Research Program, State Key Laboratory of ASIC & System (Fudan University), Fudan University Graduate Innovation Fund and AMD University Cooperation Program.

## References

[1] Guy E. Blelloch and Charles E. Leiserson and Bruce M. Maggs and C. Greg Plaxton and Stephen J. Smith and Marco Zagha, A comparison of sorting algorithms for the connection machine CM-2, Proceedings of the third annual ACM symposium on Parallel algorithms and architectures, 3-16, 1991.

[2] Jatin Chhugani and Anthony D. Nguyen and Victor W. Lee and William Macy and Mostafa Hagog and Yen-Kuang Chen and Akram Baransi and Sanjeev Kumar and Pradeep Dubey, Efficient implementation of sorting on multi-core SIMD CPU architecture, PVLDB 1(2), 1313-1324, 2008.

[3] Daniel Cederman and Philippas Tsigas, A Practical Quicksort Algorithm for Graphics Processors, Proceedings of the 16th annual European symposium on Algorithms, 246-258, 2008.

[4] Naga Govindaraju and Jim Gray and Ritesh Kumar and Dinesh Manocha, GPUTeraSort: high performance graphics co-processor sorting for large database management, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 325-336, 2006.

[5] A. Greb and G. Zachmann, GPU-ABiSort: optimal parallel sorting on stream architectures, The 20th International Parallel and Distributed Processing Symposium, 2006.

[6] Linh Ha and Jens Kruger and Claudio T Silvay, Fast 4-way parallel radix sorting on GPUs, Computer Graphics Forum, 28, 2368-2378, 2009.

[7] M. Harris and S. Sengupta and J. D. Owens, Parallel prefix sum (scan) with cuda, GPUGems 3, 851-876, 2007.

[8] Victor W. Lee and Changkyu Kim and Jatin Chhugani and Michael Deisher and Daehyun Kim and Anthony D. Nguyen and Nadathur Satish and Mikhail Smelyanskiy and Srinivas Chennupaty and Per Hammarlund and Ronak Singhal and Pradeep Dubey, Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, Proceedings of the 37th annual international symposium on Computer architecture, 451-460, 2010.

[9] David R. Musser, Introspective sorting and selection algorithms, Softw. Pract. Exper, 27(8), 983-993, 1997.

[10] Nadathur Satish and Mark Harris and Michael Garland, Designing efficient sorting algorithms for manycore GPUs, Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing, 1-10, 2009.

[11] Nadathur Satish and Changkyu Kim and Jatin Chhugani and Anthony D. Nguyen and Victor W. Lee and Daehyun Kim and Pradeep Dubey, Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort, 2010 SIGMOD Conference, 351-362.

[12] Shubhabrata Sengupta and Mark Harris and Yao Zhang and John D. Owens, Scan primitives for GPU computing, Proceedings of the 22nd ACM SIGGRAPH EUROGRAPHICS symposium on Graphics hardware, 97-106, 2007.

[13] Erik Sintorn and Ulf Assarsson, Fast parallel GPU-sorting using a hybrid algorithm, Journal of parallel and distributed computing, 68(10), 1381-1388, 2008.

[14] Marco Zagha and Guy E. Blelloch, Radix sort for vector multiprocessors, Proceedings of the 1991 ACM IEEE Conference on Supercomputing, 712-721, 1991.