International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 18 (2015) pp 39315-39319
© Research India Publications. http://www.ripublication.com

Performance Evaluation of Merge and Quick Sort using GPU Computing with CUDA

Neetu Faujdar
Ph.D Scholar Department of Computer Science and Engineering, Jaypee University of Information Technology, Solan District neetu.faujdar@mail.juit.ac.in

Satya Prakash Ghrera
Professor Department of Computer Science and Engineering, Jaypee University of Information Technology, Solan District sp.ghrera@juit.ac.in

Abstract
The Sorting can be of two ways first is sequential sorting and second is parallel sorting. Now a day's it is not a good idea to do the sorting of data sequentially as in the present life we require the huge data and many applications with large computational requirements and data-intensive applications are rapidly evolving in many scientific domains. In this way, we have to do parallel computation in the sorting field also. Sorting algorithms can be parallelized in many ways, but the most efficient way is to parallelize the sorting algorithms is GPU computing. So in this paper we are using the GPU computing to parallelize the merge and quick sort. So the paper presents the algorithms for fast sorting of large lists by using modern GPUs. The goal of this paper is to test the performance of merge and quick sort using GPU computing with CUDA on a dataset and to evaluate the parallel time complexity and total space complexity taken by merge and quick sort on a dataset. Merge and quick sort is evaluated on four cases of the dataset. The four cases of the dataset are random data, reverse sorted data, sorted data, and nearly sorted data, and we compare the performance with sequential with merge and quick sort.

Keywords: GPU (Graphics Processing Unit), CUDA (Compute Unified Device Architecture), Merge Sort, Quick Sort.

Introduction
The parallel execution of sorting algorithms on the graphics processing unit (GPU) is allowed by general purpose computing, on graphics processing unit (GPGPU) [1]. Sorting algorithms having highly parallel code are handled by GPU as a co-processor. The processing power of the GPU is easily available for GPGPU, The framework of NVIDIA's compute unified device architecture (CUDA) release that provides free programmability of GPUs [3]. CPUs with number of stream processors are used for floating point calculations. Parallelism is limited in a stream processor like ALUs [5]. Stream processor acts as an ideal in parallelism of floating point operations. For example, NVIDIA GTX 260 contains 250 on-board stream processors. The GPU has a much greater computation throughput compared with a CPU. NVIDIA implemented their GPGPU architecture through extensions to the C programming language to allow for simple integration with existing applications. Unlike for code running on the host supplied by full ISO C++ through NVIDIA's CUDA [2] compiler, functions executed on the device only supports CUDA C. Many general purpose applications require high-performance sorting algorithms; therefore many sorting algorithms have been explicitly developed for GPUs [4]. So we are going to develop the parallel version of merge and quick sort using GPU in the framework of NVIDIA's CUDA C.
The goal of this paper is to test the performance of merge and quick sort using GPU computing with CUDA on a dataset and to evaluate the parallel time complexity and total space complexity taken by merge and quick sort on a dataset. Both parallel and sequential versions of merge and quick sort are evaluated on four cases of the dataset [6] which are random data, reverse sorted data, sorted data and nearly sorted data. The remaining paper consists of following sections. The detail of merge and quick sort is given in section 2 and the parallel time complexity of the merge and quick sort is calculated in section 3. The comparison of sequential and parallel execution time of merge and quick sort on a dataset is given in section 4. The space complexity based testing and comparison of merge and quick sort on a dataset is done in section 5 and we present the conclusion and related work with a few comments in section 6.

Sorting Algorithms
Parallel Merge Sort using GPU computing with CUDA
Merge sort is a divide and conquer technique [8]. In merge sort first divide the elements till we get single elements and then merge the elements and finally get a sorted list of items [7]. Merge sort preserves the order of duplicate keys or we can say that it is a stable sorting and it is not an adaptive sorting.
Parallel merge sort consists of three phases. In the first phase, we split the input data into 'p' equally sized blocks. In the second phase, all 'p' blocks are sorted using 'p' thread blocks. In the final phase, sorted blocks are merged into the final sequence. Let's understand the concept of parallel merge sort with the help of an example. In a first phase assign each thread to a number in the unsorted array example of parallel merge sort is shown in figure 1, we have used the two blocks and 4 threads per block.
Now we will see the CUDA function of merge sort:
The function sortBlocks() is used to sort the blocks. To do this each block is first compared with the adjacent element and the

elements are sorted after doing this. So the group is made of the four elements and the third process continues till we have got the sorted elements in the block.
The function mergeBlocks() is used to merge the blocks. We merge the blocks to make a larger size block, but arranged in such way that the elements in the resultant array are sorted. As the size of block doubles so this function is called until we are left with the single block.

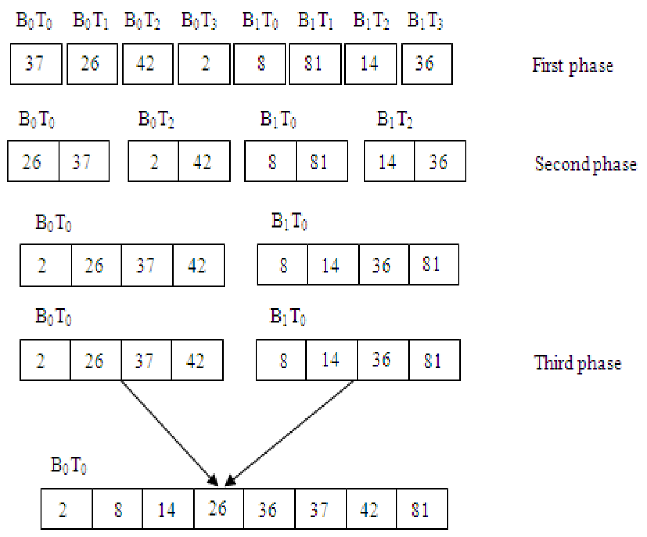

Fig.1. Example of parallel merge sort

Parallel Quick Sort using GPU computing with CUDA
Quick sort is a divide and conquer technique [10]. Quick sort is not stable sorting and not an adaptive sorting algorithm. The quick sort can be applied for highly parallel multi-core graphics processors [11]. Previously quick sort was not an efficient sorting solution for graphics processors, but we show that using CUDA with C on the NVIDIA's programming platform GPU-Quick sort [12] performs better than the fastest known sorting implementations for graphics processors. Parallel partition of quick sort is as follows; we use the deterministic pivot selection in our approach and used the different pivot selection scheme in two phases. During the first phase, value of pivot is calculated based on the average of minimum and maximum value of the sequence. In the next phase, the choice of pivot element is based on the median of the first, middle and last element [12].

Phase I
- Threads to be assigned to the several blocks.
- All the thread blocks will be working on the different parts of the same sequence of the elements to be sorted.
- After that we have to synchronize all the thread blocks.
- Different subsequences are formed by merging results of the different blocks.
- Still, we need to have a thread block barrier function between the partition iterations because blocks might be executed sequentially and we have no idea to know that in which order threads will be run.
- So, there is only one way to synchronize thread blocks are to wait until all blocks have finished executing. So user can assign new sequence to them.

Phase II
- In this phase, thread block is assigned its own subsequence of input data so, need of synchronization between thread and block will be eliminated.
- This means the second phase can run entirely on the GPU.
- Finally, we will get sorted list of items.

Parallel Time Complexity of Merge and Quick Sort
Merge Sort
Let p be the number of processes and p<n. Initially, each process is assigned a block of n/p elements which it sort internally in Θ ((n/p) log (n/p)) time. During each phase Θ (n) comparisons are performed and time Θ (n) is spent in communication [9]. So the formal representation of parallel run time is shown in equation (1).

$$T_p = \theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \theta(n) + \theta(n) \qquad (1)$$

Quick Sort
The parallel time depends on the split and merges time, and the quality of the pivot. For optimized results the primary focus is on the choice of pivot element. The algorithm executes in four steps: (i) choose the pivot and broadcast (ii) Rearrange the array and locally assigned to each process (iii) Rearrange the array globally and determine the locations in that array for the local elements will go to and (iv) perform the global rearrangement. Quick sort takes time Θ (log *p*) to choose the pivot, it will take Θ (*n/p*) in the second step, the third step takes time Θ (log *p*), and the fourth step takes time Θ (*n/p*). So the formal representation of parallel run time of quick sort is Θ (*n/p*) + Θ (log *p*). The algorithm will work until the lists are sorted locally for p lists. Therefore, the overall parallel runtime time of the parallel quick sort is shown in equation (2).

$$T_p = \theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \theta\left(\frac{n}{p}\log p\right) + \theta\left(\log^2 p\right) \qquad (2)$$

Experimental Evaluation
Hardware
We ran the algorithms on Window 7 32-bit operating system Intel® core™ i3 processor 530@ 2.93 GHz machine. System having the GeForce GTX 460 graphic processor with (7

multiprocessors X (48) CUDA cores\MP) = 336 CUDA cores. There are maximum 1536 threads per multiprocessor and 1024 threads per block. System having the CUDA driver version is 5.0 and CUDA runtime version is 4.2. The total amount of global memory present in the system is 768 Mbytes and the total amount of constant memory is 65536 bytes. The total amount of shared memory per block is 49152 bytes. System having the total number of registers available per block is 32768 and warp size is 32. Maximum sizes of each dimension of a block are 1024 x 1024 x 64 and maximum size of each dimension of a grid is 65535 x 65535 x 65535.

Algorithms Used

We compared the GPU quick and merge sort with CPU quick and merge sort. We have tested the merge and quick sort on a dataset [T10I4D100K (.gz)] [13]. The dataset contains the 1010228 items. The testing consists based on the dataset four cases as follows:

i. Random with repeated data (Random data)
ii. Reverse sorted with repeated Data (Reverse sorted data)
iii. Sorted with repeated data (Sorted data)
iv. Nearly sorted with repeated data (Nearly sorted data)

TABLE.1. Sequential and parallel execution time in seconds of merge sort using the four cases of the dataset.

Sorting Algorithms	Random	Nearly Sorted	Sorted	Reverse Sorted
Sequential Merge Sort	0.172	0.125	0.124	0.125
Parallel Merge Sort	0.0300016	0.02000154	0.01000151	0.02000167

TABLE.2. Sequential and parallel execution time in seconds of quick sort using the four cases of the dataset.

Sorting Algorithms	Random	Nearly Sorted	Sorted	Reverse Sorted
Sequential Quick Sort	1.043904	1.219802	1.26322	72.089548
Parallel Quick Sort	0.080012	0.085014	0.085013	0.085014

By analysing both the table 1 and 2 we can see that parallel merge and quick sort performs better results in comparison to the sequential merge and quick sort. We can see this effect with the help of graphs. In the figure 2 and 3, the X-axis represents the type of dataset and the Y-axis represents the execution time in seconds. The sequential quick sort having the performance gap for reverse sorted data versus other datasets. It is because of the depth of recursion, but it is not in the parallel case because in the parallel case we are taking the median value as a pivot. The performance gap of sequential quick sort can be overcome by using the median as a pivot.

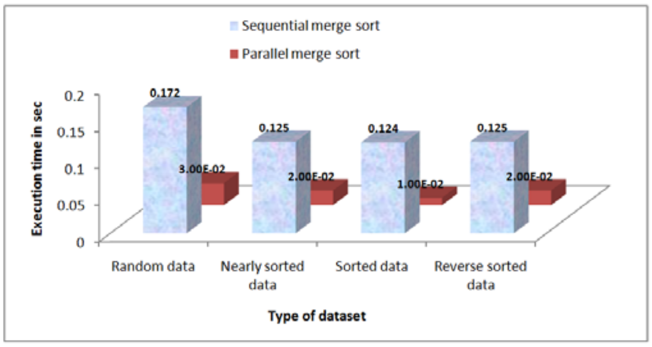

Fig.2. Graph shows the execution time comparison between sequential and parallel merge sort using four cases of the dataset

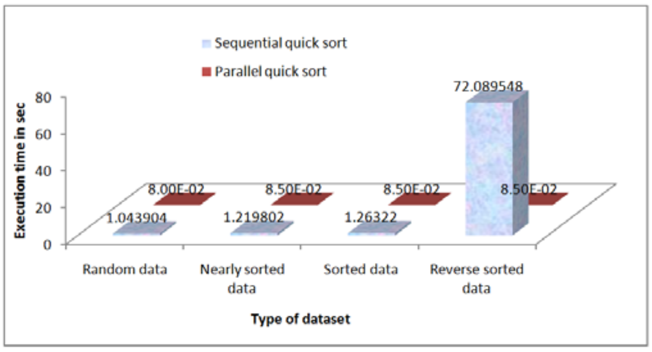

Fig.3. Graph shows the execution time comparison between sequential and parallel quick sort using four cases of the dataset

Space Complexity based Testing of Merge and Quick Sort using Dataset

Space complexity is not only limited to auxiliary space. It is the total space taken by the program which includes the following [6].

1. Primary memory required to store input data (M_{ip}).
2. Secondary memory required to store input data (M_{is})
3. Primary memory required to store output data (M_{op}).
4. Secondary memory required to store output data (M_{os})
5. Memory required to hold the code (M_c)
6. Memory required to working space (temporary memory) variables + stack (M_w)

We have calculated the space complexity for the every case of a dataset of merge and quick sort algorithm. In the figure 4 and 5 X-axis represents the type of dataset and the Y-axis represents the memory in bytes.

Merge Sort

Space complexity of sequential merge sort is 18023234 bytes. It will be a replica of the dataset having four cases. Space complexity of parallel merge sort is 18159366 bytes. It is also a replica of the dataset having four cases. It is shown in table 3.

TABLE.3. Sequential and parallel memory in bytes of merge sort using the random dataset.

Sorting Algorithms	M_{ip}	M_{is}	M_{op}	M_{os}	M_c	M_w	Total(bytes)
Sequential Merge Sort	4040924	4932283	4	4932283	76800	4040940	18023234
Parallel Merge Sort	4040924	4932283	4040924	4932283	110,592	102360	18159366

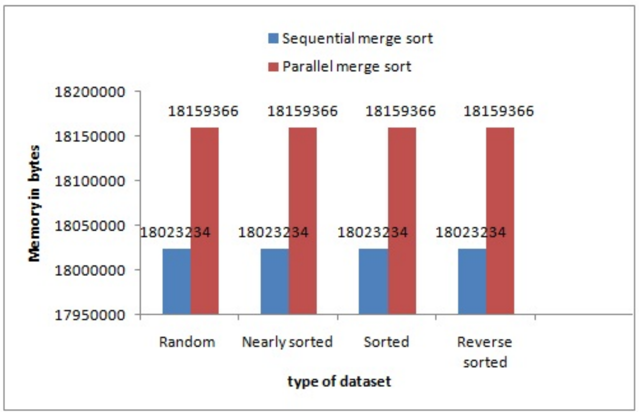

Fig. 4. Graph shows memory comparison between sequential and parallel merge sort using the four cases of the dataset.

Quick Sort

Space complexity of quick sort is shown in table 4 using all the four cases of the dataset

Table 4. Sequential and parallel memory in bytes of quick sort using all the four cases of the dataset.

Data Set	Sorting Algorithms	M_{ip}	M_{is}	M_{op}	M_{os}	M_c	M_w	Total(bytes)
Random Data	Sequential Quick Sort	4040924	4932283	4	4932283	76288	97756	14079538
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675,840	1422912	20045166
Nearly Sorted Data	Sequential Quick Sort	4040924	4932283	4	4932283	76288	97468	14079250
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675,840	1590880	20213134
Sorted Data	Sequential Quick Sort	4040924	4932283	4	4932283	76288	97756	14079538
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675,840	1590880	20213134
Reverse Sorted Data	Sequential Quick Sort	4040924	4932283	4	4932283	76288	99366	14081148
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675,840	1590880	20213134

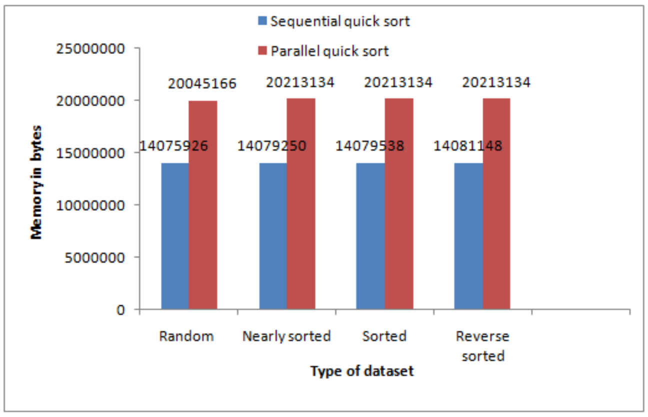

Fig. 5. Graph shows memory comparison between sequential and parallel quick sort using the four cases of the dataset.

By analysing the figure 4, 5 and table 3, 4 we found that the memory occupied by the sequential merge and quick sort is less in comparison to the parallel merge and quick sort. It is because we need more space to make parallel copies in parallel algorithms, but in sequential algorithms we do the sorting directly on the array.

Conclusion

In this paper, we present CUDA merge and quick sort and compare with the sequential merge and quick sort using dataset. We have used the four cases of the dataset which are random data, reverse sorted data, sorted data and reverse sorted data. All the four cases of dataset contain the 1010228 items. Sequential merge and quick sort is implemented in C-language. The designing of the program is done at Borland C++ 5.02 compiler and executed on Window 7 32 bit operating system Intel® core™ i5-3230 2.93 GHz machine, and the programs running at 2.2 GHz clock speed. GPU merge and quick sort implemented in CUDA using C at GeForce GTX 460 graphic processor. Experimental results of execution time have shown that GPU merge and quick sort outperforms the sequential merge and quick sort in all the four cases of the dataset. Space complexity analysis shows that sequential merge and quick sort outperforms the parallel merge and quick sort.

References

[1] Ghorpade, Jayshree, et al. "Gpgpu processing in cuda architecture", arXiv preprint arXiv, Vol.3, No.1, January 2012.

[2] Sanders, Jason, and Edward Kandrot. "CUDA by example: an introduction to general-purpose GPU programming", Addison-Wesley Professional 2010.

[3] Fatica, Massimiliano, et al. "High performance computing with CUDA", SC08 2008.

[4] Matloff, Norm. "Programming on parallel machines", University of California, Davis 2011.

[5] Pirjan, Alexandru. "Improving software performance in the Compute Unified Device Architecture", Informatica Economica, Vol.14, No.4, 2010.

[6] Neetu Faujdar, Satya Prakash Ghrera. "Analysis and Testing of Sorting Algorithms on a Standard Dataset", Paper Presented at the IEEE Fifth International Conference on Communication System and Network Technologies, 962-967, 2015.

[7] Żurek, Dominik, et al. "The comparison of parallel sorting algorithms implemented on different hardware platforms", Computer Science, Vol.14, No.4, 2013.

[8] Mišić, Marko J., and Milo V.Tomašević. "Data sorting using graphics processing units", Telfor Journal, Vol.4, No.4, 2012.

[9] Manwade, K. B. "Analysis of Parallel Merge Sort Algorithm", International Journal of Computer Applications, Vol. 12, No.19, pp. 70-73, 2010.

[10] Rajput, Ishwari S., Bhawnesh Kumar, and Tinku Singh. "Performance comparison of sequential quick sort and parallel quick sort algorithms", International Journal of Computer Applications, Vol. 57, No.9, pp. 14-22, 2012.

[11] Sintorn, Erik, and Ulf Assarsson. "Fast parallel GPU-sorting using a hybrid algorithm", Journal of Parallel and Distributed Computing, vol. 68, no. 10, pp. 1381-1388, 2008.

[12] Cederman, Daniel, and Philippas Tsigas. "Gpu-quicksort: A practical quicksort algorithm for graphics processors", Journal of Experimental Algorithmics (JEA), Vol. 14, No. 4, pp.1-22, 2009.

[13] Zubair Khan, Neetu Faujdar, et al. "Modified BitApriori Algorithm: An Intelligent Approach for Mining Frequent Item-Set", Proc. Of Int. Conf.on Advance in Signal Processing and Communication, pp. 813-819, 2013.