Performance Improvement for Multi-Key Quick Sort using Kepler GPUs

Bharath Shamasundar, Amoolya Shetty
Department of Computer Science and Engineering
NMAM Institute of Technology
Udupi, India
{bharathbs294, amushetty93}@gmail.com

Ananya Rao, Supreetha Shetty, Neelima Bayyapu
Department of Computer Science and Engineering
NMAM Institute of Technology
Udupi, India
{ananyaraoj, supreethashetty8}@gmail.com;
reddy_neelima@yahoo.com

Abstract— **This paper presents the performance improvement obtained by multi key quick sort on Kepler NVIDIA GPUs. Since Multi key quick sort is a recursion based algorithm many of the researchers have found it laborious to parallelize the algorithm on the multi and many core architectures. A survey of imperative string sorting algorithm and a robust perceptivity of the Kepler GPU architecture gave rise to an intriguing rumination of matching the template of Multi key quick sort with the dynamic parallelism feature offered by the Kepler based GPU's. The CPU parallel implementation of parallel multi key quick sort gives 33 to 50 percentage of improvement and 62 to 75 percentage of improvement when compared to 8-bit and 16-bit parallel MSD radix sort respectively. The GPU implementation of multi key quick sort gives 6X to 18X speed up compared to the CPU parallel implementation of parallel multi key quick sort. The naïve GPU implementation of multi key quick sort achieves 1.5X to 7 X speed up when compared with the GPU implementation of string sorting algorithm using singleton elements by Deshpande and Narayanan [1].**

Keywords- Multikey quick sort; Graphic processing unit; Kepler NVIDIA GPU;

I. INTRODUCTION

In this ever changing world, and the expanding horizon of research ideas, the need of the hour is to accelerate time bound applications with minimal effort. The use of GPUs has evolved from being a graphic rendering module to a component that can be used to handle efficient computations in parallel. The use of GPUs for general purpose computations is called General-Purpose computing on Graphics Processing Units (GPGPU). With the advent of GPGPU's it is now possible to exploit the massive computing capability of the GPU's by the use of APIs that shrouds the GPU hardware working from the programmers making it easily programmable. One such programming API that has been introduced by NVidia for parallel computing using GPUs is the Compute Unified Device Architecture (CUDA) [2-3].

With every new version of the GPU architecture, there is an increased possibility of applications that can take advantage of this improvement. A sorting algorithm, based on recursion that was previously considered incapable of being parallelized in terms of GPGPU is now ported onto the latest GPU architecture that supports the framework of the sorting algorithm is considered in this paper.

Some research studies have provided a plausible solution of using merge-sort to partition the input [4] as there are memory issues when string sorting is implemented on the GPU. This method might not work when there is a large amount of data since merge sort requires a lot of shared memory. Most Significant Digit (MSD) radix sort performs the best on GPUs.

The major contributions of this paper are as follows:

- Analyzing various string sorting algorithms that have been implemented on multi core and many core architectures
- Proposes porting multi key quick sort onto latest GPU cards that enables parallelization of recursion based algorithm
- Evaluating the implementation by comparing the performance with other high performing sorting techniques on both multi core and many core architectures.

II. BACKGROUND DETAILS

This section gives details about the CUDA programming model.

A. CUDA Programming Model

CUDA is programming interface to use the GPU for general purpose computing. It can be coded as an extension to the C language. The CPU sees a CUDA device as a multi-core co-processor. The CUDA design inherently does not have memory restrictions of GPGPU. One can access all memory available on the device using CUDA with no limitation on its representation though the access times is changing for different types of memory. Figure 1 shows the outline of CUDA programming model.

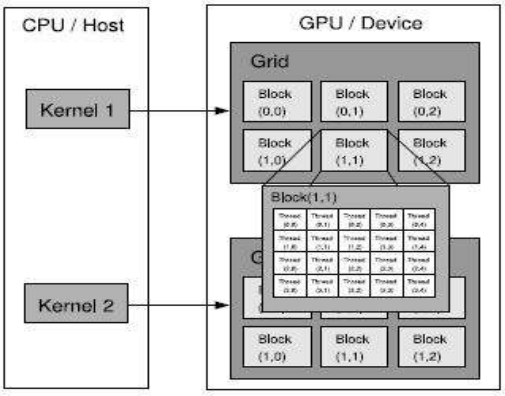

Figure 1: CUDA Programming model.

III. Related Work

This section gives review of string sorting algorithms on CPUs and GPUs.

A. CPU String Sorting Algorithms

The most popular and efficient string sorting algorithms are: Multi key quick sort [5], Most Significant Digit (MSD) radix sort [6], and Burst sort [7] [8]. Burst sort sorts the strings by using burst-trie data structure combined with some standard sorting algorithm [5] [9]. The strings are organized into small buckets and inserted into a burst-trie. These buckets are then sorted within the CPU cache memory. The buckets are already in lexicographic order and there is no need to sort them. The sorted buckets are the final desired sorted output. The limitations of burst sort include limiting the number of strings in the bucket in order to reduce the number of blocks present in the cache.

Further, Karkkainen and Rantala [6] have engineered a string radix sort using most significant digit. The authors have used counting technique to pre-compute the bucket size and the authors have also developed a dynamic approach of resizing and generating new buckets which use insertion sort when the bucket size is minimum.

Bentley and Sedgewick in [5] have implemented multi key quick sort which is a hybrid algorithm involving quick sort and MSD radix sort. The algorithm works similar to integer quick sort, in the sense that the first character and a pivot element is chosen and the remaining elements are partitioned into less than, greater than and equal to sets. These sets are in turn recursively sorted. The introduction of the equal to subset improves the efficiency as it avoids redundant string comparisons.

The implementation given in this paper is based on multi key quick sort with ternary search tree. The use of ternary search tree reduces the amount of string comparison of elements that match the pivot element under consideration.

B. GPU String Sorting Algorithms

Cederman and Tsigas [10] have developed parallel version of quick sort for which their implementation has a time complexity of O (n log (n)).Their implementation shows a speedup of 3x on high-end GPUs. Sathish, Harris and Garland [4] have developed radix sort that runs on GPU. Their implementation divides the input sequence into blocks that are processed by individual processors. This makes efficient use of memory bandwidth. They have also developed a fast merge-sort which merges small blocks present on the on-chip shared memory. Merill and Grimshaw [11] developed radix sort that uses fast scan primitives. Their algorithm optimizes radix sort by using it in a digit size manner in order to fit it on the GPU. Furthermore, Davidson et al. [12] have developed an efficient string sorting algorithm that makes use of merge sort in order to process strings with variable key lengths. Recently, Banerjee et al. [13] have made use of a hybrid CPU+GPU platform and implemented a faster merge sort algorithm than Davidson et al.Their merge sort is 20% faster for fixed length keys and 24% faster for variable length keys when compared to Davidson et al.

Deshpande and Narayan [1] have proposed an algorithm which places the string with their key values. Strings with unique prefixes will be singleton and will be placed in the final output, the rest of the strings are assigned segment id and sorting continues. The algorithm gains a speedup

of more than 10x over the best GPU string sorting algorithms.

IV. Parallel Multikey Quick sort

This section gives an overview of working of multi key quick sort and its implementation details on CPU and GPU.

A. Multikey Quick sort

Multi key quick sort is an adaptation of integer quick sort [14, 15]. Similar to integer quick sort, a pivot character is chosen. All the strings *s* with a common prefix *l* are chosen. *S* is then partitioned into $s_<$, $s_=$, $s_>$ depending on the pivot element chosen. These sub sets are in turn sorted. This process is carried out recursively. Figure 2 shows the ternary partition of input based on the pivot element in the multi key quick sort algorithm. A variant of multi key quick sort has been implemented by Rantala [16]. They have used caching of characters and it is the fastest sequential implementation of multi key quick sort till date. Bentley and Sedgewick [5] proposed multikey quick sort that uses 'split-end' partition technique to sort strings. This technique involves lots of operations to swap equal elements among the partitions. An improvement of the algorithm for many equal elements has been provided by Kim and Park [17]. They have used a 'collect-center' partitioning technique. In this method they have partitioned the equal elements directly to the middle instead of swapping them back and forth from the middle and end partitions. The proposed work in this paper uses this fast technique of partitioning in the implementation of multi key quick sort.

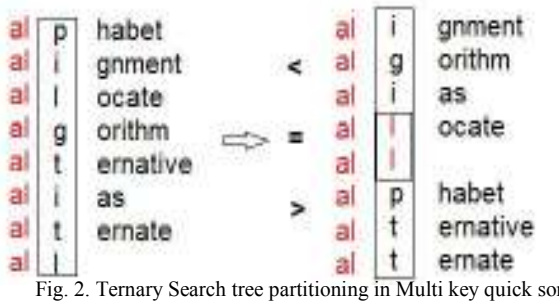

Fig. 2. Ternary Search tree partitioning in Multi key quick sort

B. CPU Parallel Multi key Quick sort

Bingmann and Sanders [18] have implemented the CPU parallel multi key quick sort. In order to parallelize multi key quick sort, the authors have extended a well-known blocking schema. When all items in a block are classified as <, = or >, then the block is added to the corresponding output set, namely S<, S=, or S>. This continues as long as un-partitioned blocks are available. If no more input blocks are available, an extra empty memory block is allocated and a second phase starts. The second partitioning phase ends with fully classified blocks that might be only partially filled. For each partitioning step there can be at most 3p partially filled blocks. The output sets S<, S=, and S> are processed recursively with threads divided as evenly among them as possible. The cached characters are updated only for the S= set.

A naïve approach to parallelize the recursive algorithm is using dynamic parallelism provided by the latest Kepler architectures. The technique of task parallelism for every recursive branch is used. By using this technique, the number of threads acting on a recursive function is doubled for every level of recursion. In the first iteration, the input string is partitioned into three sub-partition arrays of less than, greater than and equal to partitions. This iteration is done using a single thread launched by the kernel. In the next iteration the three threads are launched to handle the recursive sub-partitioning of the array. Therefore, the number of threads to be launched increases linearly by a factor of three for every iteration. The Pseudo code for GPU multi-key quick sort in shown in Figure 3.

```
GPUmultikeysort(s,n,depth)
Create CUDA Streams
If(n==1) then
  Return
Choose a pivot element v
Partition s around v on character depth to form
Sequence s<,s>,and s= of size n<,n> and n=
Launch CUDA Kernel1 for Less than partition
   multikeysort(s<,n<,depth)
If pivot element v is not NULL Character
  Launch CUDA Kernel2 for equal to partition
  Multikeysort(s=,n=,depth+1)
Launch CUDA Kernel3 for Greater than partition
Multikeysort(s>,n>,depth)
```

Fig. 3. Psuedo-Code for GPU Multikey quick sort

V. Experimental Results and Performance Analysis

This section gives implementation of efficient and popular string sorting algorithms that have been tested on different test cases and we use these algorithm's performance measures to compare multi key quick sort's performance. The sequential algorithms that have been implemented are Sinha burst sort [19], Rantala MSD radix sort [6] and Bingman and Sanders multikey quick sort

[18]. The parallel CPU algorithms include Bingman MSD radix sort and Bingman parallel multi key quick sort. The GPU implementations considered here for comparison is string sorting after removing singleton elements by Deshpande and Narayanan [1].

A. Experimental Setup

All the sequential implementation has been written in C++. All the programs have been compiled using gcc 4.6.3 and on a Linux operating system on Intel i7 processor model running at 3.4 GHz and 4 GB DDR3 1333 Zion RAM. All the details are measured using Linux clock function.

The GPU environment consists of an NVidia Tesla K40C GPU that belongs to the Kepler architecture (Compute v3.5) and CUDA software version 6.0. The performance of parallel string sorting algorithm is tested on different datasets as shown in Table I. The runtimes measured for GPU algorithms do not include the file I/O time.The input data sets have been taken from the work by Sinha [19].

B. Sequential String Sorting on CPU

The graph in Figure 4 shows the performance of the serial algorithms on various data sets. The radix sort implementation by Rantala [6] shows better performance than the other algorithms for some data sets. Radix sort performs the best among the other sequential algorithms. Multi key quick sort and burst sort perform poorly when compared to radix sort.

Datasets	**Size**	**Distinct Words (x10^5)**	**Word occurrences (x10^5)**
URL's	304Mb	12.898	100
Genome	302Mb	2.620	316.230
NoDup	382Mb	316.230	316.230
Large Artificial	169.4Mb	---	---

TABLE 1: DATASETS USED

C. Parallel String Sorting on CPU

This section analyzes the performance of the Parallel implementation of MSD radix sort that processes 16bit data, parallel MSD radix sort that processes 8bit data and parallel multi key quick sort. All The algorithms have been implemented by Bingman [18]. For the benchmark data set named Large Data Artificial, parallel multi key quick sort is 33 and 67 percent better than 8-bit and 16-bit parallel MSD radix sort respectively. For the other benchmark data set named URL, parallel multi key quick sort is 50 and 75 percent better than 8-bit and 16-bit parallel MSD radix sort respectively. Similarly, for the benchmark data set named Genome, parallel multi key quick sort is 50 and 62 percent better than 8-bit and 16-bit parallel MSD radix sort respectively.

Multi key quick sort doesn't perform as expected for NoDup data set as shown in Figure 5. Multi key quick sort performs the best when the input data set contains duplicate elements.

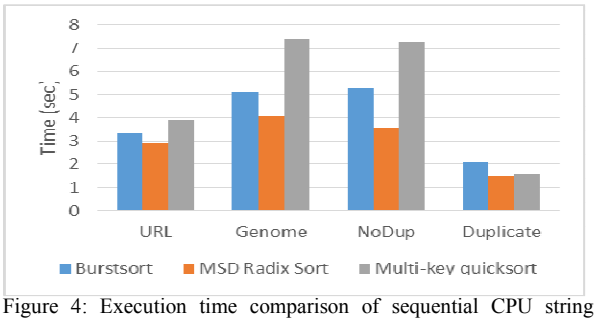

Figure 4: Execution time comparison of sequential CPU string sorting algorithms..

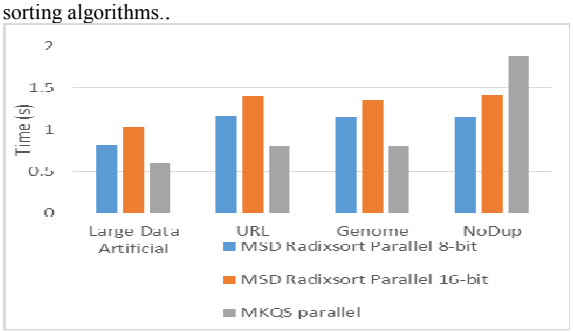

Fig. 5: Execution time comparison of parallel CPU string sorting algorithms

D. Parallel String Sorting on GPU

The proposed work in this paper utilizes the dynamic parallelism feature of Kepler Architecture. Figure 6 shows the normalized sorting time on an NVidia Tesla K40 machine. It can be seen that the GPU parallel implementation of multikey quick sort outperforms the CPU parallel multikey quick sort [18]. The GPU implementation of multi key quick sort achieves 6X to 18 X speed up in performance compared to CPU parallel implementation of multi key quick sort for the best case scenario. We also compared our work with the most efficient and highly

optimized string sorting algorithm by Deshpande and Narayanan [1].

The performance results are distributed into two graphs as shown in Figure 7 and Figure 8 to highlight the differences in each implementation and separated based on the scale of y-axis. It can be seen from Figure 8 and Figure 9 that the GPU parallel multi key quick sort outperforms the string sorting algorithm on all datasets. GPU based parallel multi key quick sort achieved 1.5X to 7X speed up on string sorting with singleton elements removed from [1].

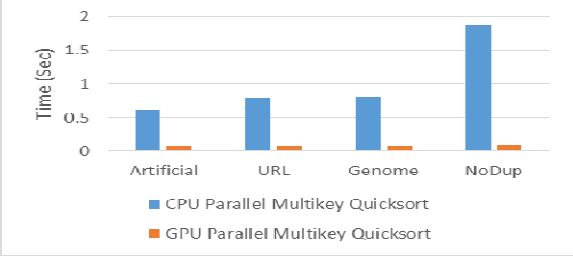

Fig. 6: Execution time comparison of CPU parallel Multikey quick sort vs GPU parallel Multi key quick sort.

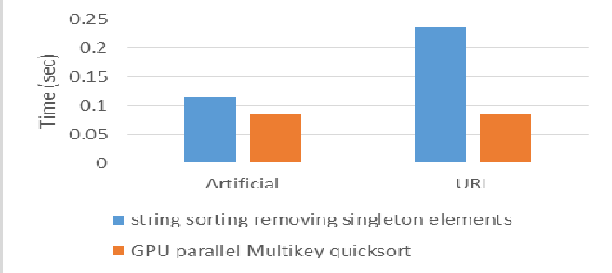

Fig. 7: Execution time comparison of GPU parallel Multikey quick sort vs GPU string sorting removing singleton elements [1] for Artificial and URL datasets.

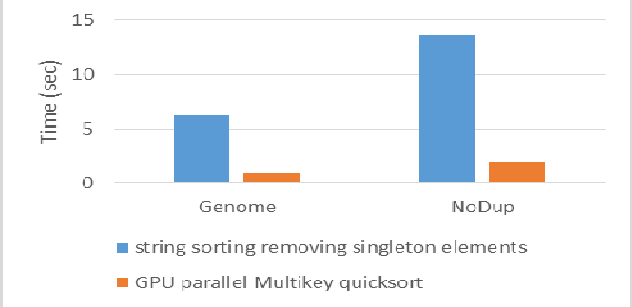

Fig. 8: Execution time comparison of GPU parallel Multikey quick sort vs GPU string sorting removing singleton elements [1] for Genome and NoDup datasets

VI. Conclusion and Future Work

The introduction of Kepler architecture opens up an untouched field of parallelizing recursive algorithms on GPUs. This work highlights the fact that a naïve implementation of multi key quick sort was able to yield high performance when compared to some of the most efficient string sorting algorithms on the Kepler architecture. As a future work, it is planned to implement the proposed algorithm on multi GPUs.

References

[1] A Deshpande, P .J. Narayan. Can GPUs sort strings efficiently?. In high performance computing (HiPC), 2013 International Conference on, pages 305-313, 2013.

[2] Neelima B., and Prakash S. Raghavendra. Recent Trends in Softwareand Hardware for GPGPU Computing: A Comprehensive Survey. In Fifth International Conference on Industrial and Information Systems (ICIIS).pp.319-324.,August 2010.

[3] http://www.nvidia.com/object/cuda_home_new.html CUDA Zone.

[4] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In Parallel Distributed Processing, 2009.IPDPS 2009. IEEE International Symposium on, pages 1–10, 2009.

[5] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, SODA '97, pages 360-369, 1997.

[6] J. Karkkainen and T. Rantala. Engineering radix sort for strings. In Proceedings of the 15th International Symposium on String Processing and Information Retrieval, SPIRE '08, pages 3–14, 2009.

[7] R. Sinha, A. Wirth. Engineering burstsort: towards fast in-place string sorting. In C. McGeoch, editor, Experimental Algorithms, volume 5038 of lecture notes in computer science, pages 14-27. 2008.

[8] R. Sinha, J. Zobel and D Ring. Cache-efficient string sorting using copying. J. Exp. Algorithmics, 11 Feb. 2007.

[9] P. M. McIlroy, K. Bostic and M. D. McIlroy. Engineering radix sort.Computing Systems, 6:5–27, 1993.

[10] D. Cederman and P. Tsigas. Gpu-quick sort: A practical quick sort algorithm for graphics processors. J. Exp. Algorithmics, 14:4:1.4–4:1.24, Jan. 2010.

[11] Merrill, D. and Grimshaw, A. 2011. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters. 21, 02 (2011), 245-272.

[12] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient parallelmerge sort for fixed and variable length keys. In Innovative Parallel Computing, page 9,May 2012.

[13] D. S. Banerjee, P. Sakurikar, and K. Kothapalli. Fast, scalable parallel comparison sort on hybrid multicore architectures. In Workshop OnAccelerators for Hybrid Exascale Systems (AsHES), IPDPS'13, 2013.

[14] C.A.R. Hoare, Quick sort, Computer Journal 5 (1) (April 1962) 10–15.

[15] R. Sedgewick, Quick sort with equal keys, SIAM Journal on Computing 6 (1977) 240–267.

[16] Tommi Rantala. Library of String Sorting Algorithms in C++. http://github.com/rantala/string-sorting.Git repository accessed November 2012. 2007.

[17] Kim, E., Park, K.: Improving multikey quick sort for sorting strings with many equal elements. Inf. Process. Lett. 109(9), 454–459 (2009).

[18] T. Bingmann and P. Sanders. Parallel string sample sort. In 21st European Symposium on Algorithme (ESA), volume 8125 of LNCS, pages 169–180. Springer, 2013.

[19] Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. J. Exp. Algorithmics 9 (December 2004) 1.5:1–31.