Implementation of GPU based Sorting Algorithms

Arvind Ramachandran
Department of Computer Engineering
National Institute of Technology Karnataka Surathkal
Email: arvind0705.ar@gmail.com

Aswanth P. P.
Department of Computer Engineering
National Institute of Technology Karnataka Surathkal
Email: ppaswanth3@gmail.com

***Abstract*—Parallel sorting algorithms are widely studied nowadays. The difficulty in improving sorting run-time when run serially has lead to a lot of research in this area. Efficient sorting is required for algorithms like binary search ,merging two lists etc. The main aim is to effectively reduce the running time of these algorithms specially after the introduction of parallel processors like the GPU and CUDA, OpenCL, etc. This paper presents a survey of some well known sorting algorithms that are GPU based.**

***Index Terms*—Merge Sort, Quick Sort, Radix Sort, Sample Sort, Bitonic Sort, Fix Sort, GPU, CUDA**

I. Introduction

Sequential algorithms were implemented on a central processing unit using C++, whereas parallel algorithms were implemented on a graphics processing unit using CUDA platform. Sorting on a Central Processing Unit(CPU) is slower than sorting on a GPU.
In this Project, we analyze five parallel sorting algorithms for a given range of inputs - Merge Sort, Quick Sort, Radix Sort, Sample Sort and Bitonic Sort. Their methods are described in brief and performance compared.

II. Objective

We are comparing sorting time for each parallel sorting algorithm for a dataset of array size 50000 to 500000 with an interval of 5000 .The performance of each algorithm will be compared in terms of running time and corresponding graphs will be plotted for each set of inputs.

III. Sorting Algorithms

We discuss a few parallel sorting algorithms here.

A. Merge Sort

Merge sort is based on the divide - and - conquer approach. It follows this approach by splitting the sequence into multiple subsequences, sorting them and then merging them into sorted sequence. If the algorithm for merging sorted sequences is stable, than the whole merge sort algorithm is also stable.

B. Quick Sort

In the Quick sort algorithm, a list of elements is taken and partitioned around a specific pivot element. The exact position of the pivot element in the sorted list is found. The lists are recursively partitioned until they become too small to partition. Quick sort is the fastest and most studied algorithm in CPU architecture.

C. Radix Sort

Radix sort is one of the fastest sorting algorithms for short keys and is the only sorting algorithm in this report which is not comparison based. Its sequential variation first splits the elements being sorted (numbers, words, dates, ...) into d r - bit digits. The elements are then sorted from least to most significant digit. For this task, the sorting algorithm has to be stable, in order to preserve the order of elements with duplicate digits.

D. Sample Sort

Sample sort is or has been the fastest sorting algorithm if the inter-process communication is high . It selects a subset of the input. This subset is referred to as splitters. Splitters are sorted by some other procedure. The input sequence is divided into buckets using these splitters. Each bucket is sorted in parallel and the result is the concatenation of these buckets. However, performance of the Sample sort degrades if the number of elements per processor is low.

E. Bitonic Sort

Bitonic Sort is one of the most studied algorithms on GPU. It falls into the group of sorting networks , which means, that the sequence and direction of comparisons is determined in advance and is independent of input sequence. Bitonic sort is based on a bitonic sequence. Parallel implementation of bitonic sort is very efficient when sorting short sequences, but it becomes slower when sorting very long sequences, because shared memory size is limited and long bionic sequences cannot be saved into it. In order to merge long bionic sequences, global memory has to be used instead of shared memory. Furthermore, global memory has to be accessed for every step of bionic merge. In order to increase the speed of sort, multiple steps of bitonic merge have to be executed with a single kernel invocation. This can be achieved with multistep bitonic sort.

IV. Project Timeline

This is the proposed timeline which we hope to achieve :
27th September 2017 - Discussion of Project Proposal
2nd October 2017 - Submission of Proposal
23rd October 2017 - Mid Progress Evaluation
13th November 2017 - Final Demo
24th November 2017 - Report Submission

V. Work Distribution

This is how we plan to split the work and proceed with the Project. The work has been divided equally among the team.

A. Arvind Ramachandran (15CO111)

Implementation and Analysis of Merge Sort and Sample Sort.

B. Aswanth P P (15CO112)

Implementation and Analysis of Quick Sort and Radix Sort. Bitonic Sort has been done by both of us together.

VI. GPU Sorting Algorithms

Merge sort, Radix sort, Quick sort, Sample sort and Bitonic sort algorithms were successfully implemented on CUDA , running times calculated and results verified using functions from header "wb.h".

A. Merge Sort

Conceptually, a merge sort works as follows:
1) Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
2) Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Since this follows a divide and conquer approach, each sort and merge operation is performed individually (recursively as per the algorithm) which takes up time. There arises a need to parallelize this algorithm in order to improve its efficiency. Merge sort parallelizes well due to use of the divide-and-conquer method. It is very difficult to find a merging algorithm that can achieve a high level of parallelism and maximize utilization on the GPU due to the multi-level parallelism requirements of the architecture. In a sense, parallelizing merging algorithms is even more difficult due to the small amount of work done per each element in the input and output. The merge phase of the original Merge Path algorithm is not well-suited for the GPU as the merging stage is purely sequential for each core. Therefore, it is necessary to extend the algorithm to parallelize the merge stage in a way that still uses all the SPs on each SM once the partitioning stage is completed. For full utilization of the SMs in the system, the merge must be broken up into finer granularity to enable additional parallelism while still avoiding synchronization when possible.

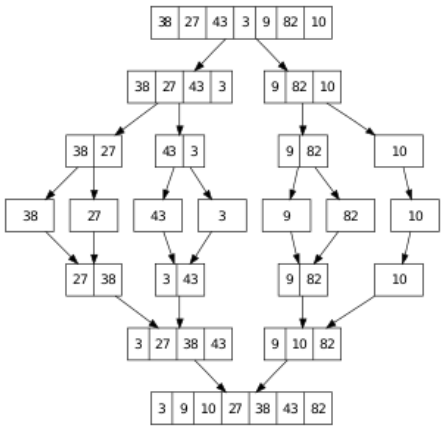

Fig. 1. An example of the execution of merge sort.

We have adopted the following strategy to parallelize merge sort :
1) Start with two lists: Your input array, and a temp array that's the same size.
2) Define a width, starting at 2. During each step, width gets multiplied by 2.
3) While width is less than 2N, sort each width-sized chunk of the list into the temp list. Then switch the pointers of the two lists. We thereby avoid allocating small arrays or copying temp back to input.

NOTE : This is the step that happens in parallel. Each thread gets a chunk of the list to sort. Two halves of each chunk are are sorted / merged against each other into the temp array.

4) We end up with one big chunk being sorted into the final list, and you switch input and temp one last time, returning temp.

B. Radix Sort

Radix Sort iterates over the keys bits from the least-significant to the most-significant digit, considering an implementation specific number of consecutive bits at a time. With each sorting pass, a stable counting sort is used to partition the keys into buckets according to the bits being considered with the current pass. The stable counting sort computes each keys offset by counting the number of keys with a smaller digit value and, as it needs to be stable, the keys with the same digit value preceding the key in the input sequence. Sorting relies on the reinterpretation of a k-bit key as a sequence of d-bit digits, which are considered one at a time. The basic idea is, that splitting the k bits of the keys into smaller d-bit digits results in a small enough radix r = 2 d , such that the keys can efficiently be partitioned into r distinct buckets. As sorting on each digit can be done with an effort that is linear in the number of keys n , the whole sorting can be achieved with a total complexity of O (

dk/de*n). Iterating over the keysâĂŹ digits can be performed in two fundamentally different ways. Either by proceeding from the most-significant to the least-significant digit (MSD radix sort), or vice versa (LSD radix sort). Radix sort is an out-of-place sort and we need to ping-pong values between the input and output buffers provided. We need to do a copy at the end.

The basic idea is to construct a histogram on each pass of how many of each âĂİdigitâĂİ there are. Then we scan this histogram so that we know where to put the output of each digit. For example, the first 1 must come after all the 0s so we have to know how many 0s there are to be able to start moving 1s into the correct position.
1) Histogram of the number of occurrences of each digit
2) Exclusive Prefix Sum of Histogram
3) Determine relative offset of each digit For example [0 0 1 1 0 0 1] = [0 1 0 1 2 3 2]
4) Combine the results of steps 2 3 to determine the final output location for each element and move it there LSB.

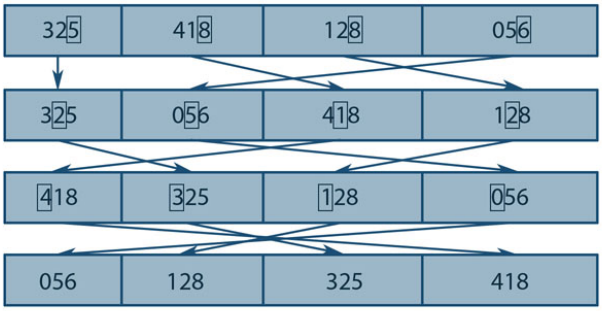

Fig. 2. Parallel Radix Sort on the GPU

Implementation of Radix on CUDA algorithm:
1) Get the predicate of your list (bit in common, starting from the LSB)
2) Scan the predicate, and record the sum of the predicate in the process Implement Scan on the GPU Note that your predicate will be of arbitrary size
3) Flip bits of the predicate, and scan that
4) Move the values in your array with the following rule: For the i^{th} element in the array: if the i^{th} predicate is TRUE, move the i^{th} value to the index in the i^{th} element of the predicate scan else, move the i^{th} value to the index in the i^{th} element of the !Predicate scan plus the sum of the Predicate
5) Move to the next significant bit (NSB)

Verification of the radix sort algorithm is done by using "wb.h" library where the result was compared with existing sorted elements file.which gave the following result for the execution of 50000 elements within range of 5000. A sample of the working is as follows :

Generic 0.030998016 Importing data to host

GPU 0.000162048 Allocating GPU memory.

GPU 0.000172800 Copying input memory to the GPU.

Compute 0.039742976 Performing CUDA computation

Copy 0.000344832 Copying output memory to the CPU

GPU 0.000151040 Freeing GPU Memory

Solution is correct.

C. Quick Sort

In the Quick sort algorithm, a list of elements is taken and partitioned around a specific pivot element. The exact position of the pivot element in the sorted list is found. The lists are recursively partitioned until they become too small to partition. Quick sort is the fastest and most studied algorithm in CPU architecture.

The algorithm is suitable for segmented scan primitive due to communications between elements (threads) inside a single segment. A pivot element is chosen in each segment (the first element of the segment). The pivot element is distributed across the segments. The input element is compared to the pivot.Greater-than or greater-than-or-equal are compared accordingly in alternating passes of the algorithm. A segmented vector containing true and false is produced by the comparison operation. This segmented vector is used to split and segment the input. As a result, smaller elements are placed at the head of the vector and larger elements are placed at the end of the vector.

A sequence to be partitioned is divided logically into sections. Each section is processed by a thread block. Each thread in the thread block keeps track of the number of elements it has seen larger and smaller than the pivot. Each thread stores this information in two arrays in shared local memory. A cumulative sum is calculated to find out the index of each element. Threads will write their assigned elements in new position in the auxiliary buffer.When the number of sub-sequences is large enough that each thread block can be assigned one, the algorithm enters in the second phase. There is no need for inter-thread block synchronization. When the sub-sequences become small enough to be sorted entirely in the fast local memory, the authors suggest using a different sorting algorithm which performs well when the size of the list approaches the number of threads.

One kernel is assigned to the group having elements smaller than the pivot and the other is assigned to the group having elements larger than the pivot. The kernel at the top knows the index and the size of the two groups; it will have

the information on whether to launch a kernel and how many threads to use. The parent kernel is able to launch its child kernels immediately after partitioning the list. The program progresses in an asynchronous manner. The kernel launches its two children in a separate stream. CUDA streams are executed simultaneously, which means the two sub-sorts will run in parallel.

Input Size : 50000
Execution Time : 0. 51
Input Size/Execution Time : 98039

D. Sample Sort

Sample sort is or has been the fastest sorting algorithm if the inter-process communication is high. It selects a subset of the input. This subset is referred to as splitters. Splitters are sorted by some other procedure. The input sequence is divided into buckets using these splitters. Each bucket is sorted in parallel and the result is the concatenation of these buckets. However, performance of the Sample sort degrades if the number of elements per processor is low.

The idea behind sample sort is simple. A sample of size s is selected from the n-element sequence, and the range of the buckets is determined by sorting the sample and choosing m - 1 elements from the result. These elements (called splitters) divide the sample into m equal-sized buckets. After defining the buckets, the algorithm proceeds in the same way as bucket sort. The performance of sample sort depends on the sample size s and the way it is selected from the n-element sequence.

Consider a splitter selection scheme that guarantees that the number of elements ending up in each bucket is roughly the same for all buckets. Let n be the number of elements to be sorted and m be the number of buckets. The scheme works as follows. It divides the n elements into m blocks of size n/m each, and sorts each block by using quicksort. From each sorted block it chooses m - 1 evenly spaced elements. The m(m - 1) elements selected from all the blocks represent the sample used to determine the buckets. This scheme guarantees that the number of elements ending up in each bucket is less than 2n/m.

How can we parallelize the splitter selection scheme? Let p be the number of processes. As in bucket sort, set m = p; thus, at the end of the algorithm, each process contains only the elements belonging to a single bucket. Each process is assigned a block of n/p elements, which it sorts sequentially. It then chooses p - 1 evenly spaced elements from the sorted block. Each process sends its p - 1 sample elements to one process - say P0. Process P0 then sequentially sorts the p(p - 1) sample elements and selects the p - 1 splitters. Finally, process P0 broadcasts the p - 1 splitters to all the other processes.

Since the algorithm spends a significant amount of time on sorting buckets (for example approximately 55 percentage for sorting 16 million randomly distributed 32-bit integers) one should not underestimate influence of this step on the overall run-time of the algorithm. We delay sorting of buckets until the whole input is partitioned into buckets of size at most M. Since the number of buckets grows with the input size, it is larger than the number of processors in most of the cases. Therefore, we can use a single thread block per bucket without sacrificing exploitable parallelism. To improve load-balancing we schedule buckets for sorting ordered by size.

In order to sort buckets efficiently we split them into chunks that fit into shared memory, which then can be sorted without expensive accesses to the global memory. Initially we employed a sample sort based algorithm for this purpose. But we found that it performed slightly worse than our quick sort adaptation by Cederman and Tsigas, which we therefore use in our final implementation. We also extended it to support key-value pairs inputs. Quick sort does not cause any serialization of work, except for pivot selection and stack operations. Additionally, its consumption of registers and shared memory is modest.

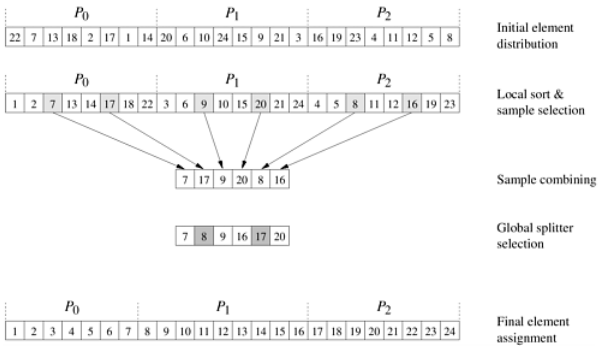

Fig. 3. An example of the execution of sample sort on an array with 24 elements on three processes.

By choosing t = 256 threads per block and = 8 elements per thread, we achieve a compromise between the parallelism exposed by the algorithm, the amount of data (nÂůk)/(tÂů) written in the second phase and memory latency in the fourth phase.

Input Size : 50000
Execution Time : 0. 09
Input Size/Execution Time : 555555

E. Bitonic Sort

The bitonic sorting scheme has been extensively used in parallel computing (mainly to construct sorting network). Bitonic sort with the property that sequence of comparisons is data-independent. Therefore it is one of the fastest parallel

sorting algorithms. There are two steps in bitonic sorting algorithm. Firstly, it makes the arbitrary sequence that is called bitonic sequence. Secondly, it can be converted to the first one by cyclically shifting. Bitonic sequence consists of two monotonic sequences. The monotonic sequence is increase or decrease from left to right.

For all k < n; If $a_k < a_{k+1}$; a_1, a_2, $a_3 ... a_n$ is a monotonic sequence. Bitonic sequence monotonically increases (decreases), reaches the maximum (minimum), then monotonically decreases (increases). For example, suppose 3 5 8 9 7 4 2 1 bitonic sequence; it increases from 3 to 9 then it decreases. If x_0, x_1, $x_2 ... x_n$ is a bitonic sequence, an index i is between 0<i<n-1. When we apply this rule to sequence, it increases from x_0 to x_i , then decreases from x_i to x_n . Bitonic split is applied to sequence to get a bitonic sequence. The sequence length must be 2n. Then, n steps are needed to sort the entire array. Each indices are compared using (i, n/2-1). If $a_i > a_{i+n/2}$, the two elements are exchanged, 1<i<n . When this swapping operation is applied to indices between i=(0, n/2-1), two bitonic sequences are produced. The first split elements are smaller than second split elements. Then, in the second step the same rule will be applied to each subsequence. Each subsequence will produce new subsequence. When n step is applied, it produces the subsequence of length 1. Finally, all the elements in the bitonic sequence will be sorted.

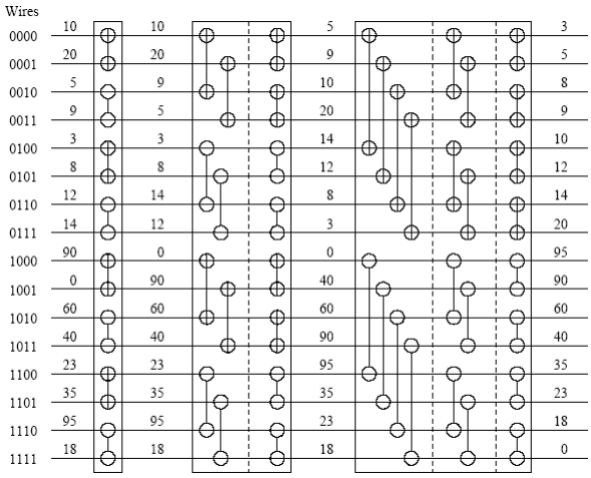

Fig. 4. Bitonic Sorting Network

Bitonic sort consists of O(n(logn)) 2 comparison operation (compare/exchange) in each n/2 step. Total number of steps are k=log(n). When this algorithm is parallelized, it would take n processor to sort it with O(logn) 2 complexity. All steps in bitonic sort are creating a bitonic sequence and the sorting are (k(k+1))/2. For example the sorting of an array that consist 16 elements (2 4) would take 10 steps. Firstly, pair wise exchange is made to make it easier to find the bitonic sequence. Then, it is divided two sub-arrays. Each element in sub-array is compared using (i,n/2-1) indices. If $a_i > a_{i+n/2}$, swap operation is applied. After all elements are compared, bitonic sequence is implemented. The first split of the array is in increasing order; the second split is in decreasing order. It is reached to bitonic sequence in 3 steps. Next step is the bitonic sorting. These 2 bitonic splits are divided again until each split that will have 1 element. Compare and swap approach is applied to all splits. Finally, bitonic sorted array is implemented.

In parallel implementation of this sorting algorithm, each split and compare operation is applied on different cores. Each core works on different part of an array. These splitted arrays results don't affect the other one's result. Therefore, each CUDA core works on array separately.

VII. Performance Analysis

We have compared the running times of these sorting algorithms over an input size ranging from 10000 to 50000. We have plotted a graph of as Rung time against Input Size nnifollows:

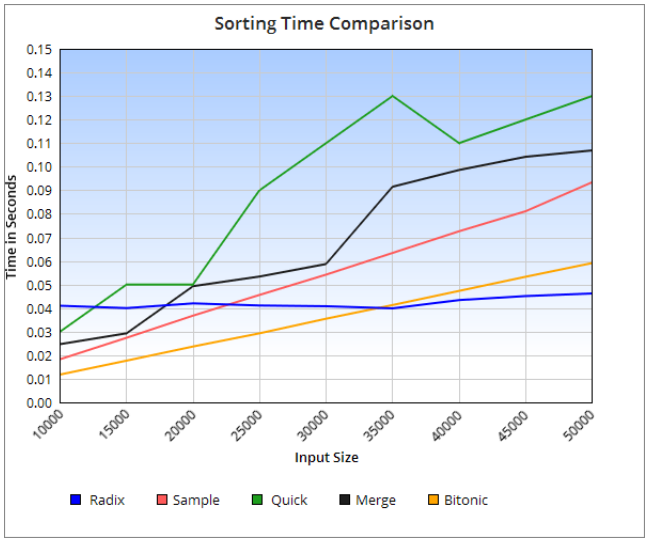

Fig. 5. Sorting Time Comparison

VIII. Conclusion

This paper presented a survey of GPU based sorting algorithms. We have selected five sorting algorithms for this purpose - Merge sort, Radix sort, Sample sort, Quick sort and Bitonic sort. Working and performances of these algorithms have been calculated and compared. Our analysis showed that all these performed better than their serial counterparts. Due to the highly optimal complexity of Radix sort, it is a difficult to outrival algorithm on GPU. In the future, we will try to extend our algorithms and try other sorting algorithms to compare these algorithmsâĂŹs performance. Different data ranges and distribution can be applied to these algorithms in both academic and industrial fields based on the use of GPU. Finally, we will compare the results for more GPU compute

capabilities and propose an algorithm to automatically choose the best one for each input parameters.

REFERENCES

[1] Sam White, Niels Verosky and Tia Newhall, "A CUDA-MPI Hybrid Bitonic Sorting Algorithm for GPU Clusters", in *41st International Conference on Parallel Processing Workshops*, 2012.

[2] Zehra Yildiz, Musa Aydin and Guray Yilmaz, "Parallelization of bitonic sort and radix sort algorithms on many core GPUs".

[3] Bakulev Aleksander Valerievich, Bakuleva Marina Alekseevna, Pyurova Tatiana Anatolievna and Skvortsov Sergei Vladimirovich, "The Implementation on CUDA Platform Parallel Algorithms Sort the Data", in *6th Mediterranean Conference on Embedded Computing*, 2017.

[4] Nikolaj Leischner, Vitaly Osipov and Peter Sanders, "GPU Sample Sort", in *Parallel & Distributed Processing (IPDPS), IEEE International Symposium*, 2010.

[5] Rafael Schmid, Edson Borin, Edson Caceres and Flavia Pisani, "An Evaluation of Segmented Sorting Strategies on GPUs", in *14th IEEE International Conference on Smart City*; *IEEE 2nd International Conference on Data Science and Systems*; *18th IEEE International Conference on High Performance Computing anf Communications*, 2016.

[6] Dhirendra Pratap Singh, Ishan Joshi and Jaytrilok Choudhary, "Survey of GPU Based Sorting Algorithms", 2017.

[7] http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf