Implementation of GPU based Sorting Algorithms

Arvind Ramachandran(15CO111) and Aswanth PP(15CO112)
Department of Computer Engineering, NITK Surathkal
Email: arvind0705.ar@gmail.com, ppaswanth3@gmail.com

***Abstract*—Parallel sorting algorithms are widely studied nowadays. The difficulty in improving sorting runtime 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 analyse 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 incr ease 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. Work Distribution

This is how we plan to split the work and proceed with the Project.

A. Arvind Ramachandran (15CO111)

Implementation and Analysis of Merge Sort and Bitonic Sort.

B. Aswanth P P (15CO112)

Implementation and Analysis of Quick Sort, Radix Sort and Sample Sort.

V. Progress - I

Merge sort and Radix 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 parallelise 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 theGPU 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.

We have adopted the following strategy to parallelise merge sort :
1) Start with two lists: Your input array, and a temp array thats 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 complex- ity of O (d k/d e n). Iterating over the keys 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).

The basic idea is to construct a histogram on each pass of how many of each digit 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 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.
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 ith element in the array: if the ith predicate is TRUE, move the ith value to the index in the ith element of the predicate scan else, move the ith value to the index in the ith 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 compared the result with existing sorted elements file.which gave the following result for the execution of 50000 elements within range of 5000

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.

VI. 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

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.