PERFORMANCE ENHANCEMENT OF SORTING ALGORITHMS USING GPU COMPUTING WITH CUDA

Thesis submitted in fulfillment for the requirement of the Degree of

DOCTOR OF PHILOSOPHY

By

NEETU FAUJDAR

Enrollment No. 136211

Under the Supervision of

Prof. Dr. Satya Prakash Ghrera

Department of Computer Science and Engineering

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT, SOLAN 173234, H.P., INDIA

September, 2016

©Copyright JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT
ALL RIGHTS RESERVED September, 2016

DECLARATION BY THE SCHOLAR

I hereby declare that the work reported in the Ph.D. thesis entitled, **"PERFORMANCE ENHANCEMENT OF SORTING ALGORITHMS USING GPU COMPUTING WITH CUDA"** submitted at **Jaypee University of Information Technology, Waknaghat, India**, is an authentic record of my work carried out under the supervision of **Prof. Dr. Satya Prakash Ghrera**. I have not submitted this work elsewhere for any other degree or diploma.

Neetu Faujdar
Enrollment Number-136211
Department of Computer Science
Jaypee University of Information Technology, Waknaghat, India
Date: 24th September 2016

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY
(Established by H.P. State Legislative vide Act No. 14 of 2002)
P.O. Waknaghat, Teh. Kandaghat, Distt. Solan - 173234 (H.P.) INDIA
Website : www.juit.ac.in
Phone No. +91-01792-257999 (30 Lines).
Fax: +91-01792-245362

CERTIFICATE

This is to certify that the thesis entitled, **"PERFORMANCE ENHANCEMENT OF SORTING ALGORITHMS USING GPU COMPUTING WITH CUDA"** which is being submitted by **Neetu Faujdar** in fulfillment for the award of degree of **Doctor of Philosophy** in **Computer Science** by the **Jaypee University of Information Technology, Waknaghat, India** is the record of candidate's own work carried by her under our supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Prof. Dr. Satya Prakash Ghrera
Head of Department
Department of Computer Science
JUIT, Waknaghat
PIN - 173234 (H.P.) INDIA

Acknowledgements

This thesis arose in part out of years of research that has been done at Jaypee University of IT Waknaghat, HP, India. For this research I have worked with a great number of people whose contributions in assorted ways to the research the making of the thesis deserved special mention. It is a pleasure to convey my gratitude to them all in my humble acknowledgment.

First and foremost I express my deep sense of gratitude to my supervisor, **Dr. Satya Prakash Ghrera**, Professor, Department of CSE, JUIT, Waknaghat, for his guidance, encouragement and invaluable suggestions that made this research possible.

I would also like to convey thanks to **Dr. Vivek Sehgal**, Ph.D Coordinator and, **Dr. Pardeep Kumar**, Faculty, Jaypee University of Engineering & Technology and other DPMC (Doctoral Program Monitoring Committee) members for providing me assistance, moral support, valuable suggestions and necessary facilities during the course of my research work.

I also wish to convey my gratitude to Director & Academic Head **Prof. Samir Dev Gupta**, Vice Chancellor **Prof. Vinod Kumar**, for their encouragement and help during the course of my research tenure at JUIT. I would like to thank the authorities of Jaypee University of Information Technology, Waknaghat for providing the financial support during my research work.

Many colleagues at JUIT collaborated with me for the success of the research. I would like to acknowledge the contribution of **Mr Akash Punhani**, **Mr Jabir Ali** and **Ms Sukhnandan Kaur**, research scholars. For the preparation of the thesis I have been assisted by the laboratory staff of the department of CSE and IT. I greatly acknowledge their assistance. I would also like to thank all faculty members of JUIT for their support in numerous ways. I would like to thank the entire Administration and Management of JUIT for supporting me for this research work.

In the end, I dedicate this thesis to my parents **Jawahar Singh** and **Malti Devi**, who have always been the source of inspirations for continued higher learning and achievements. I would like to express my earnest gratitude to my husband **Sudhir Kushwah** and my brother-in-law **Sohan Singh** for their support and encouragement.

(Neetu Faujdar)

Table of Contents

Abstract

The main goal of the thesis is to show the novel use of the Graphics Processing Unit (GPU) computing with Compute Unified Device Architecture (CUDA) hardware for enhancing the performance of sorting algorithms. The properties of sorting algorithms make them special in that they can use the parallel processing in order to have good time and space complexity. Many authors have implemented the some sorting algorithms using GPU computing with CUDA. A comprehensive study of various sorting algorithms and developed some new algorithms using GPU computing on CUDA hardware is the objective of the work.

The dataset and sorting benchmark has been considered for testing the various sorting algorithms. The dataset consists the four types of datasets. The four cases of the dataset are random data, reverse sorted data, sorted data, and nearly sorted data. The sorting benchmark contains the six types of test cases which are uniform, gaussian, zero, bucket, staggered and sorted.

In the beginning, Count sort, Merge Sort, Quick Sort and Bubble Sort have been tested on standard dataset and sorting benchmark using GPU computing and compared with the sequential version of same. The outcome shows that more speedup is gained by parallel sorting algorithms.

Next, the problems of odd-even transposition sorting network (OETSN) has been solved. Odd-even transposition sorting is designed for networks. In networks compare-exchange operation is used to compare the elements. We have find that the time taken for sorting by OETSN is same for all test cases such as uniform, sorted, zero, gaussian, staggered and bucket. The sequential and parallel time complexity is $O(n^2)$ and $O(n)$ respectively, of OETSN using any kind of test cases. In our approach, we reduced the time complexity $O(n)$ to $O(1)$ over two types of test case which are sorted and zero. We have motivated from the bubble sort technique. If the data is sorted and unique, bubble sort requires only one pass and terminate the program. In our approach we have also used this technique.

In this way, we have reduced the number of levels in the network and the time complexity for sorted and zero test cases. OETSN has been tested using GPU computing with CUDA hardware on the six types of test cases.

Next, we have solved the problems of library sort algorithm. Library sort is also called gapped insertion sort. It is a sorting algorithm that uses insertion sort with gaps. Time taken by insertion sort is $O(n^2)$ because each insertion takes $O(n)$ time; and library sort has insertion time $O(\log n)$ with high probability. Total running time of library sort is $O(n \log n)$ time with high probability. Library sort has better run time than insertion sort, but the library sort also has some issues. The first issue is the value of gap which is denoted by 'ε', the range of gap is given, but it is yet to be implemented to check that given range is satisfying the concept of library sort algorithm. The second issue is re-balancing which accounts the cost and time of library sort algorithm. The third issue is that, only a theoretical concept of library sort is given, but the concept is not implemented. The library sort algorithm is designed and the gap value has evaluated.

The library sort using non-uniform gap distribution algorithm (LNGD) is also proposed in the thesis. The final results have shown that, execution time is decreased when the gap value increases.The proposed algorithm is tested using the four types of test cases. The experimental result of proposed algorithm is compared to the library sort algorithm with uniform gap distribution (LUGD) and LNGD proves to provide better results in all the aspects of execution time like re-balancing and insertion. We have achieved an improvement that ranges from 8% to 90%. The improvements of 90% has been found in the cases where the LUGD is performing poorer.

Finally, we have proposed the efficient bucket sort algorithm. The bucket sort has two issues 1) The first issue is that the bucket sort has the dynamic nature and the memory for each bucket is allocated at the run time. 2) The second issue is based on the data distribution over the buckets. The data are distributed to the designed buckets. If the data is equally distributed to the buckets, then there is no issue comes to the algorithm. But the problem occurs

in the algorithm if elements belonging to the same bucket are in large number rather than having element being classified equally into different buckets. We have solved this problem using the threshold (τ) for the size of buckets. The threshold is calculated for each bucket and different size of data sets. It will helpful to decide the nature of data and to reduce the memory consumption.

The application of the proposed algorithms is mainly in the area of Commercial computing, Search for information, Operations research, Event-driven simulation, Numerical computations, Combinatorial search, Prim's algorithm, Dijkstra's algorithm, Kruskal's algorithm, Huffman compression, String processing, Searching, Frequency distribution, Selection and Convex hulls. The results are discussed in accordance to the sorting algorithms implemented. Experimental results have shown that proposed sorting algorithms can powerfully enhance the performance in all aspects of sorting. Finally, we concluded and give suggestions for future research work.

Keywords

Sorting, Searching, GPU Computing, CUDA, Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort, Heap Sort, Shell Sort, Counting Sort, Radix Sort, Library Sort, Bucket Sort, Stability, Time Complexity, Space Complexity, Adaptivity, Sorting Benchmark, Standard Dataset, Parallel Sorting.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation behind Sorting

Sorting is a fundamental operation in computing. Hardly a month goes by without any research reported on the problem. While it is important to thus understand the basic principles behind classic sorting algorithms, the questions in this experiment let you think of current problems. Another important aspect which can be observed from this experiment is to think of large data sets. How classic computation such as sorting should be designed for those large data sets.

1.2 Introduction of Sorting

Sorting [1] is defined as arranging an unordered collection of data into particular order. Order can be monotonically increasing or decreasing. Suppose M = (p_1, p_2, p_3....p_n) be a sequence of *'n'* elements in unsorted manner, sorting transforms '*M*' into a monotonically increasing sequence S' = (p'_1, p'_2, p'_3....,p'_n). Sorting has two categories, internal sorting and external sorting [2]. Sorting algorithm can be sort the data as comparison-based and non-comparison-based. In comparison-based sorting algorithm [3], sort the unordered data by comparing the pairs of data repeatedly, and if the data are out of order then exchange them to each other. This exchanging operation of this sorting is called compare-exchange. Non-comparison-based [4] sorting algorithms sort the data by using certain well known properties of the data, such as the binary representation or data distribution. Sorting algorithms have four types of performance measures which are stability, adaptivity, time complexity, space complexity [5].

Definition 1 (Stability) A sorting algorithm is stable [6] if it preserves the order of duplicate keys, or stability means that equivalent elements retain their relative positions, after sorting.

Definition 2 (Adaptivity) A sorting algorithm is adaptive [7] if it sorts the sequences that are close to sorted faster than random sequences. A sorting algorithm [8] is denoted adaptive if the time complexity is a function depending on the size as well as the pre-sortedness of the input sequence.

Definition 3 (Time complexity) Time complexity [9] of an algorithm signifies the total time required by the program to run to completion. Algorithms have different cases of complexity [10] which are best case, average case, and worst case. The time complexity of an algorithm is represented using the asymptotic notations [11]. Asymptotic notations provide the lower bound and upper bound of an algorithm.

Definition 4 (Space complexity) Space complexity [12] of any algorithm is also important, and it is the number of memory cells which an algorithm needs. Space complexity calculated by both auxiliary space and space used by the input.

1.3 Sorting Algorithms

In this section, the working of some traditional and popular sorting algorithms have been explained with the help of algorithms and examples.

1.3.1 Insertion Sort

Insertion sort [11] algorithm works efficiently for a small number of elements. In insertion sort [13] algorithm we sort the one element at a time. We can use the

Algorithm 1 Insertion Sort Algorithm

INPUT: Unsorted list of n items

OUTPUT: Sorted list of n items

```
for m = 0 to length[A] do
    lock = A[m]
    Arrange A[m] in sorted order
end for
i = m - 1
while (i > 0 and A[i] > lock do) do
    A[ i + 1] = A[i]
    i = i - 1
    A[i + 1] = lock
end while
```

insertion sort algorithm, when we sort a deck of cards. In this algorithm we pick an element from the list and place it in the correct location in the list. Process will be repeated till there is no more unsorted items remained. It is an adaptive sorting algorithm, it takes $O(n)$ time when data is nearly sorted. Insertion sort algorithm is stable, and also it is an online sorting. The insertion sort algorithm can be depicted as follows; for the value m = 2, 3,..., n. Where n = Length [A].

Time Complexity of Insertion Sort

• **Best Case:** In the above algorithm line 1 to 7 is the outer loop and line 5 to 7 is the inner loop. Insertion sort have best case when the data is already sorted or nearly sorted, and for the best case the inner loop never executed in the algorithm. So the comparison will be like that, we need 1-comparison to compare first element and 2-comparison to compare second element and n-comparison to compare n-elements.

$$1 + 2 + 3 + ... + n \tag{1.3.1}$$

$$T(n) = \Omega(n) \tag{1.3.2}$$

• **Average Case:** If we compare the average case with worst case, then we find that, the average case comes to be same as the worst case. Suppose if there are 'n' numbers and the numbers chosen randomly and apply an insertion sort. Then how much time algorithm will take to determine the sub array $A[1...m\text{-}1]$ to insert element $A[m]$. In average case, we divide the elements in two halves, in one half elements are in [1... m-1] and these elements are less than $A[m]$, and in another half the elements are greater. In average case, we also check one half of the sub array $A[1...m\text{-}1]$ and so t_m is become $m/2$. Then we find the resulting average case execution time to be a quadratic function of the input size 'n', which is same as the worst case execution time. i. e.

$$T(n) = \Theta(n^2) \tag{1.3.3}$$

• **Worst Case:** Insertion sort worst case occurs if the array is re-versed sorted that is in decreasing order. In the worst case, inner loop is executed exactly m-1 times for every iteration of the outer loop. Calculation of number of comparison of an array 'n' elements in worst case will be: to insert the first element comparison is not necessary, and to insert the second element one comparison is needed and so on, and to insert the last element (n-1) comparisons is required at most

$$Total : 1 + 2 + 3 + ... + (n - 1) = O(n^2) \tag{1.3.4}$$

Space Complexity of Insertion Sort

Auxiliary space complexity of insertion sort is $O(1)$, i.e. insertion sort having a constant space complexity.

1.3.2 Selection Sort

In the selection sort [11] algorithm, smallest item will be selected and swapped by the item which is the filled in the next position. Selection sort working is that: we search the smallest element through the entire array, once we find it, swap item with the smallest element in the position of the first element of the array. After that we search for the second smallest element in the remaining array and exchange it with the second element and so on. Selection sort is not stable sorting, and it is also not adaptive sorting. Selection sort comes in the categories of comparison based sort. Selection sort algorithm can be depicted as follows, in the algorithm length $[B] = n$, where 'n' is the input data which is used for sorting in the algorithm.

Algorithm 2 Selection Sort Algorithm

INPUT: Unsorted list of n items

OUTPUT: Sorted list of n items

```
n ← = length[B]
for m ← 1 to n - 1 do
    smallest ← m
    for i ← m + 1 to n do
        if B[i] < B[smallest] then
            smallest ← i
            exchange B[m] ↔ B[smallest]
        end if
    end for
end for
```

Time Complexity of Selection Sort

- **Best Case, Average Case, Worst Case** There is difficult to analyzing the time complexity in selection sort algorithm [10], because there are no loops depend on the item in the given array n-1 comparison will be taken to selecting the lowest element. And to select the lowest element, we require scanning all 'n' elements and then lowest element swapped to the first position. After that we again scan the remaining n-1 elements to find the next lowest element, and also further scan the elements till there are no more items to swap, so the time complexity of selection sort will be

$$T(n) = (n-1) + (n-2) + ... + 2 + 1 = n(\frac{n-1}{2}) = \Theta(n^2) \tag{1.3.5}$$

Comparisons. In any cases of selection sort (worse case, best case or average case) the number of comparisons between elements is the same. So in all the three cases selection sort have the time complexity:

$$T(n) = \Theta(n^2) \tag{1.3.6}$$

Space Complexity of Selection Sort

Auxiliary space complexity of selection sort is $O(1)$ i.e. selection sort having a constant space complexity.

1.3.3 Bubble Sort

The basic idea underlying the bubble sort [4] is to pass through the list of elements sequentially several times. In each pass, we compare each element in the array and interchanging the two elements if they are not in proper order. Bubble sort is a stable sorting algorithm. It is an adaptive sorting algorithm, it takes $O(n)$ time when data type is nearly sorted, it is also a comparison based sorting algorithm. Bubble sort algorithm can be depicted as follows: In the algorithm length [Array] $= n$, where 'n' is the input data which is used for sorting in the algorithm.

Algorithm 3 Bubble Sort Algorithm

```
INPUT: Unsorted list of n items
OUTPUT: Sorted list of n items
  for i ← 1 to length[array] - 1 do
      for m ← to length[array] - i do
          if array[m] > array[m + 1] then
              exchange array[m] ↔ array[m + 1]
          end if
      end for
  end for
```

Time Complexity of Bubble Sort

• **Best Case:** Bubble sort have best case when the data in the array is already sorted or nearly sorted. In the best case, where the array is already sorted algorithm will terminate after the first iteration and no swap will made and the one iteration will take n-1 comparison, so the

$$T(n) = \Omega(n) \tag{1.3.7}$$

• **Average Case:** Bubble sort average case is $\Theta(n^2)$, which is same as worst case of bubble sort.

• **Worst Case:** In the worst case of bubble sort elements compares till n-1 iterations with the following comparisons.

$$T(n) = (n-1) + (n-2) + ... + 2 + 1 = n(\frac{n-1}{2}) = O(n^2) \tag{1.3.8}$$

Space Complexity of Bubble Sort

Auxiliary space complexity of bubble sort is $O(1)$, i.e. bubble sort having a constant space complexity.

1.3.4 Heap Sort

Heap sort algorithm is sorted the items based on the data structure heaps. Heaps are two types: 1. A max heap 2. A min heap. Heap sort should satisfy the heap property. Heap property: [15]

• All nodes are either greater than or equal to or less than or equal to of each of its children, according to a comparison the node will define for the heap.

• Heaps with a mathematical ($\geq$) comparison predicate are called max-heaps. Figure 1.1 shows the max-heap.

• Heaps with a mathematical ($\leq$) comparison predicate are called min-heaps. Figure 1.2 shows the min-heap.

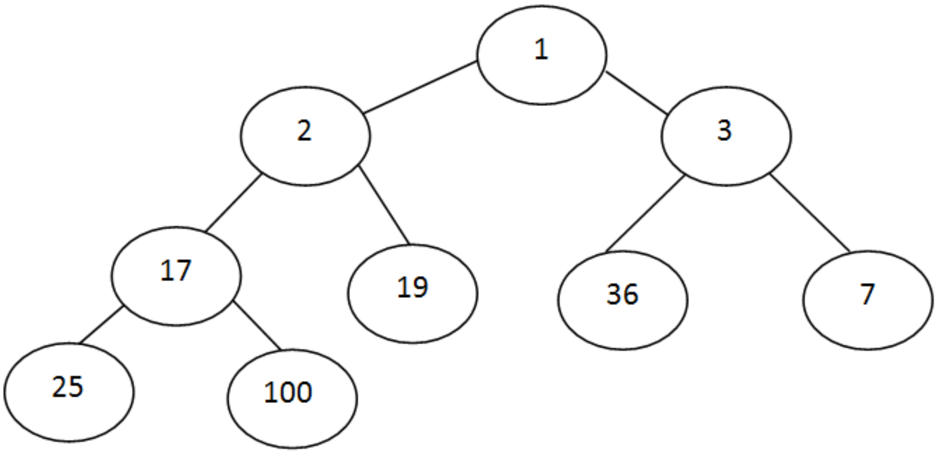


Figure 1.2: Min-heap

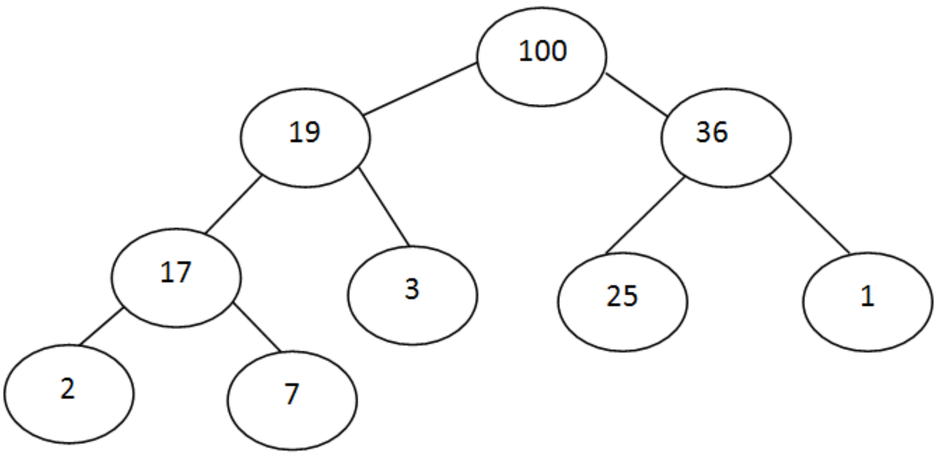


Figure 1.1: Max-heap

Heap sort works as: [9]

- Construct a heap,
- Add each item to its (maintaining the heap property),
- After adding the all items, we remove them one by one (restoring the heap property as each one is removed).

Heap sort is not a stable sorting algorithm. Heap sort is not an adaptive sorting algorithm. A heap sort algorithm can be depicted as follows:

Algorithm 4 Heap Sort Algorithm

INPUT: Unsorted list of n items

OUTPUT: Sorted list of n items

```
BuildHeap(B)
for i ← length(B) down to 2 do
    exchange B[1] ↔ B[i]
    heap-size[B] ← heap-size[B] - 1
    Heapify(B, 1)
end for
```

Algorithm 5 Heapify (B, i)

INPUT: Unsorted list of n items

OUTPUT: Sorted list of n items

```
l← left[i]
r ← right[i]
if l ≤ heap-size[B] and B[l] > B[i] then
    largest ← l
else
    largest ← i
end if
if r ≤ heapsize[B] and B[i] > B[largest] then
    largest ← r
end if
if largest ≠ i then
    exchange B[i] ↔ B[largest]
    Heapify(B, largest)
end if
```

Time Complexity of Heap Sort

- **Best Case, Average Case, Worst Case**

1. $\Theta(n)$ trips through the loop.

2. $T(n)_{Maxheapify} = \Theta(\log\ n)$.

3. $T(n)_{Heapsort} = \Theta(n) \times \Theta(\log\ n) = \Theta(n\ \log\ n)$

Space Complexity of Heap Sort

The auxiliary space complexity of heap sort is $O(1)$. i.e. it is having constant space complexity.

1.3.5 Shell Sort

Shell sort [3] is the diminishing increment sort. Shell sort is a generalization of bubble sort or insertion sort. The shell sort makes many passes through the data in to array, and each time sort the numbers that are equally distanced. The shell sort algorithm is similar to bubble sort in the sense that, it also moves elements by the exchanges. It begins by comparing elements that are at a distance'd'. In each pass of shell sort the value of 'd' is reflected to half i.e. in each pass, we compare the each element that is located 'd' position away from it, after comparing the element exchanges are made if required. In the next iteration the value of 'd' will get change, the algorithm terminates when d=1.

The example of shell sort is explained in Figure 1.3 to Figure 1.5.

12, 9, -10, 22, 2, 35, 40

$d = n/2 = 7/2 = 3$

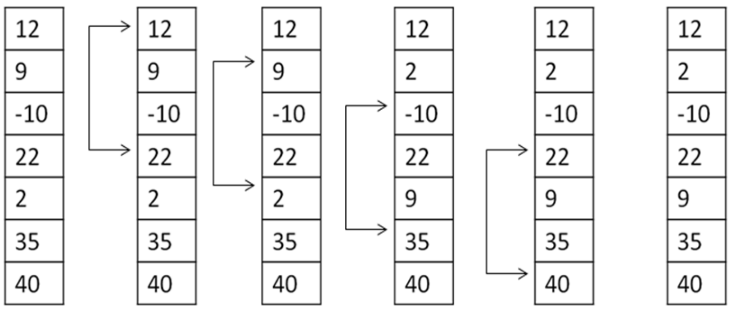


Figure 1.3: Example of shell sort

Pass 1 is completed in Figure 1.3. Now the value of $d = 2$

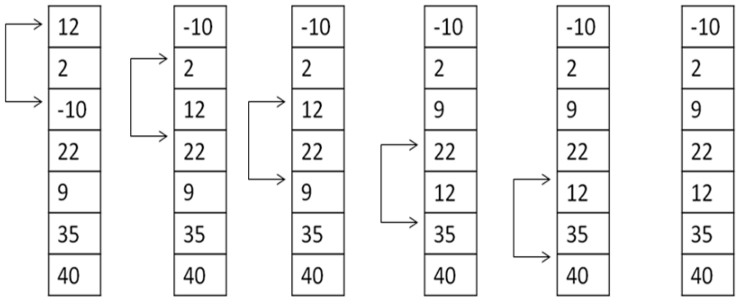


Figure 1.4: Example of shell sort

Pass 2 is completed in Figure 1.4. Now the value of $d = 1$.

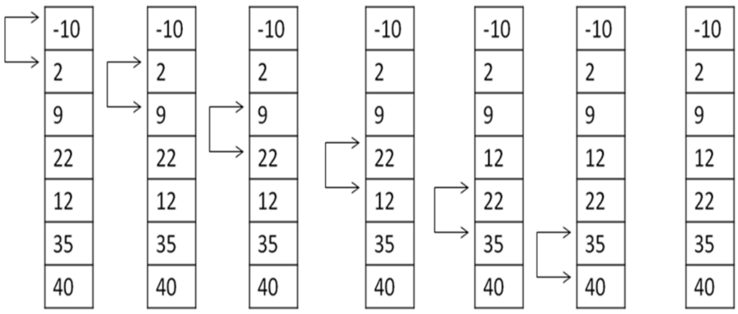


Figure 1.5: Example of shell sort

Pass 3 is completed, and the algorithm got terminated as $d = 1$. Finally, we got the sorted list -10, 2, 9, 12, 22, 35, and 40 Shell sort is no preserved the stability. It is an adaptive algorithm: it takes $O(n\log n)$ time when data is nearly sorted. The shell sort algorithm can be depicted as follows:

Algorithm 6 Shell Sort Algorithm

INPUT: Unsorted list of n items

OUTPUT: Sorted list of n items

```
for each(gap in gaps) do
    for m = gap; m < n; m + = 1 do
        temp = a[m]
    end for
end for
for q= m; q > = gap and a[q - gap] > temp; q - = gap do
    a[q] = a[q - gap]
    a[q] = temp
end for
```

Time Complexity of Shell Sort

• **Best Case:** Shell sort have best case, when the data is already sorted, because the number of passes will be less in this case. Passes = n, for 1 sorts with item 1 apart (last step) $3 \times n/3$, for 3 sorts with items 3 apart (next-to-last step. $7 \times n/7$, for 7 sorts with items 7 apart. $15 \times n/15$, for 15 sorts with items 15 apart +.....Each term is 'n'. The question arise how many terms are there. The value of 'k' such that $2k$ - l $< n$. So $k < \log\,(n + 1)$, meaning that the sorting time in the best case is less than

$$n \times log(n+1) = \Omega(nlogn) \tag{1.3.9}$$

• **Average Case:** The average case time complexity of shell sort is depends on the gap sequences between the elements.
• **Worst Case:** The worst case is similar as average, but overall computation is differ. Passes $\leq n^2$, for 1 sort with item 1 apart (last step). $3 \times (n/3)^2$, for 3 sorts with items 3 apart (next-to-last step). $7 \times (n/7)^2$, for 7 sorts with items 7 apart. $15 \times (n/15)^2$, for 15 sorts with items 15 apart +.... So the number of passes is bounded by

$$n^2 \times \left(1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + ...\right) < n^2 \times \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...\right) = n^2 \times 2 = O(n^2) \tag{1.3.10}$$

Space Complexity of Shell Sort

The auxiliary space complexity of shell sort is $O(1)$. i.e. it is having constant space complexity.

1.3.6 Counting Sort

Counting sort [16] is a linear time sorting, counting sort is an algorithm that sorts the items according to keys. Counting sort is used to sort those items, when they belong to a fixed and finite set. Example of items belongs to a fixed interval are $m1$ to $m2$, and example of items belongs to finite set will be no limit. The algorithm makes two passes one for 'A' and one for 'B'. If the size of items belongs to fixed interval range 'm' will be less than size of input 'n', then time complexity will be $O(n)$. Counting Sort preserves the stability. The efficiency of counting sort is better for the number of objects, when its range is less than input data. The algorithm of counting sort is as follows:

Algorithm 7 Counting Sort Algorithm

INPUT: Unsorted list of n items
OUTPUT: Sorted list of n items

```
for i = 1 to m do
    X[i] = 0
end for
for j = 1 to length[A] do
    X[A[j]] = X[A[j]] + 1
    X[i] here elements equal to i contains by X[i]
end for
for i = 2 to m do
    X[i] = X[i] + X[i-1]
    X[i] here number of elements less than or equal to i contains by X[i]
    B[C[A[j]]] = A[j]
    X[A[j]] = X[A[j]] - 1
end for
```

Time Complexity of Counting Sort

• **Best Case:** If the size of items belongs to fixed interval range 'm' will be less than size of input 'n', then time complexity will be $T(n) = \Omega(n)$.

• **Average Case:**

$$\Theta(n) + \Theta(k) = \Theta(n + k) \tag{1.3.11}$$

• **Worst Case:**

$$O(n) + O(k) = O(n + k) \tag{1.3.12}$$

Space Complexity of Counting Sort

The auxiliary space complexity of counting sort is $O(n+k)$, because the counting sort needs $O(n)$ auxiliary space for the array and 'k' is the number of key elements.

1.3.7 Quick Sort

Quick sort [11] use the divide and conquer technique[38]. Quick sort first divides the given array of data into two smaller sub-arrays: the low elements and the high elements, sub-arrays are recursively sorted. The steps of quick sort are[39]:
1. Quick sort first chooses the pivot from the list.
2. Reorder the array, and in the array the values which are less than the pivot come before the pivot and the values which are greater than the pivot come after the pivot and the equal values can go either way.
3. The above two steps are recursively applied to the sub-array of data with smaller data and separately to the sub-array of data with greater values.

Quick sort is not a stable sorting algorithm. Quick sort is not an adaptive sorting algorithm; it is a comparison based sorting. The quick sort algorithm can be depicted as follows. Partitioning the array: The main thing in the algorithm is the partition procedure, which rearranges the sub array $A[p....r]$.

Algorithm 8 Quick Sort Algorithm

INPUT: Unsorted list of n items
OUTPUT: Sorted list of n items

```
if p < r then
    q = Partition(A, p, r)
    Quick sort(A, p, q - 1)
    Quick sort(A, q + 1, r)
end if
Partition(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1 do
    if A[j] ≤ x then
        i = i + 1
        Exchange A[i] with A[j]
        Exchange A[i+ 1] with A[r]
        return i + 1
    end if
end for
```

Time Complexity of Quick Sort[17]

• **Best Case:** Best case of quick sort occurs when the sub arrays are completely balanced every time. Each sub array has $\leq n/2$ elements. Get the recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Omega(n) \tag{1.3.13}$$

Where $\Omega(n)$ time is the partitioning cost . By case 2 of master theorem [11] the equation (1.2.1) has the solution:

$$T(n) = \Omega(nlogn) \tag{1.3.14}$$

• **Average Case:** In average case, we produce the good and bad splits by partition the sub array.

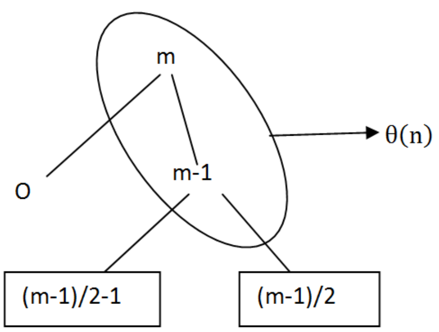


Figure 1.6: Split of n on two consecutive levels

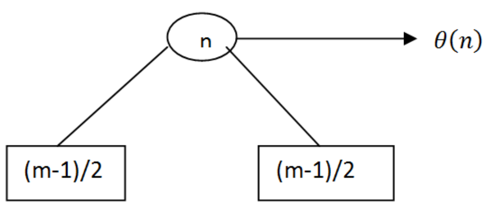


Figure 1.7: Split of n on two consecutive levels

Figure 1.6 shows the division of recursion tree at two consecutive levels. The partitioning cost shown by root of tree 'n', and the size of sub array produced m -1 and '*0*', which is the worst case of quick sort. And next level have , the sub array of size m-1 undergoes best case partitioning into a sub array of size (m-1)/2-1 and (m-1)/2. So if we combine the bad split and the good split then it produces three sub array and the size of sub array is 0, (m-1)/2-1, and (m-1)/2 at a total partitioning cost of:

$$\Theta(n) + \Theta(n-1) = \Theta(n) \tag{1.3.15}$$

Figure 1.7 shows a single level partitioning that produces two sub arrays of size (m-1)/2 at a cost of $\Theta(n)$ Both figures result in $\Theta(n\log n)$ time, though the constant for the figure on the left is higher than that of the figure on the right. So

$$T(n) = \Theta(nlogn) \tag{1.3.16}$$

• **Worst Case:** Unbalanced sub-array produces the worst case of quick sort. It has '0' element in one sub-array and n-1 element in the other sub-array. Then the recurrence:

$$T(n) = T(n-1) + T(0) + O(n) \tag{1.3.17}$$

$$T(n) = T(n-1) + O(n) \tag{1.3.18}$$

$$T(n) = O(n^2) \tag{1.3.19}$$

Where $O(n)$ is the partitioning cost, which is same as insertion sort. Quick sort have worst case, when the input is in sorted manner, but insertion sort runs in $O(n)$ time in this case. Worst case of quick sort also occurs when the array is in reverse sorted order and also when all elements are same.

Space Complexity of Quick Sort

Quick sort uses the constant additional space with unstable partitioning before making any recursive call. Constant amount of information is stored by quick sort for every nested recursive call. As the best case of quick sort takes at most $O(\log n)$ many nested recursive calls, it uses $O(\log n)$ space. However, if we limit the recursive calls, without Sedgwick's trick then the worst case of quick sort could make $O(n)$ nested recursive calls and need $O(n)$ auxiliary space.

1.3.8 Radix Sort

There are two types of radix sorting [19]:

1. MSD radix sort starts sorting from the beginning of strings (Most Signicant Digit).

2. LSD radix sort starts sorting from the end of strings (Least Signicant Digit).

Example of radix sort explained in Figure 1.8 to Figure 1.12.

Ex. 0712, 21171, 00120, 43589, 73641, 31975, 60433

i. Padding used to make all numbers 5-digits.

ii. Start from the last most digit.

iii. We can use the buckets.

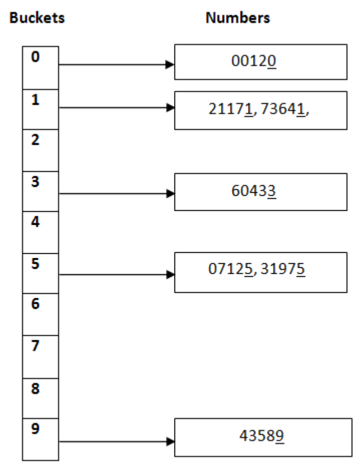


Figure 1.8: Example of Radix Sort

Now reassemble the list according to the buckets of Figure 1.8.

00120, 21171, 73641, 60433, 07125, 31975, 43589

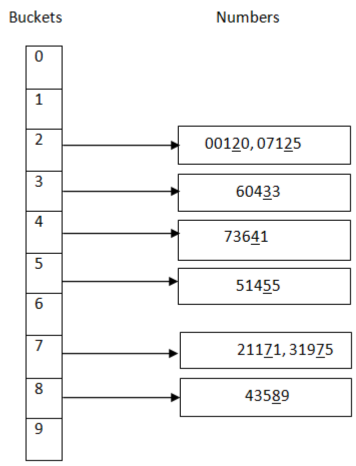


Figure 1.9: Example of Radix Sort

Now reassemble the list again according to the buckets of Figure 1.9. We will get 00120, 07125, 60433, 73641, 51455, 21171, 31975, 43589.

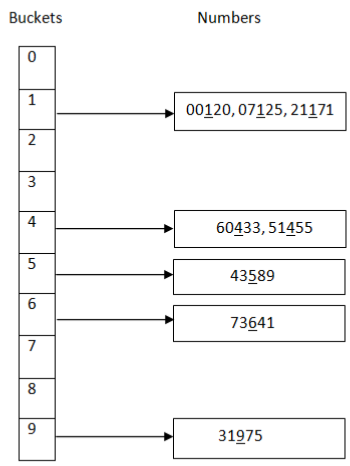


Figure 1.10: Example of Radix Sort

Now reassemble the list again according to the buckets of Figure 1.10, we will get 00120, 07125, 21171, 60433, 51455, 43589, 73641, 31975.

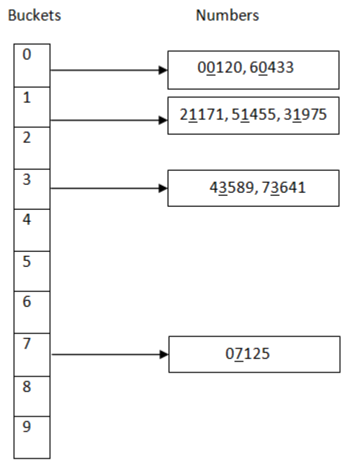


Figure 1.11: Example of Radix Sort

Now reassemble the list again according to the buckets of Figure 1.11, we will get 00120, 60433, 21171, 51455, 31975, 43589, 73641, 07125.

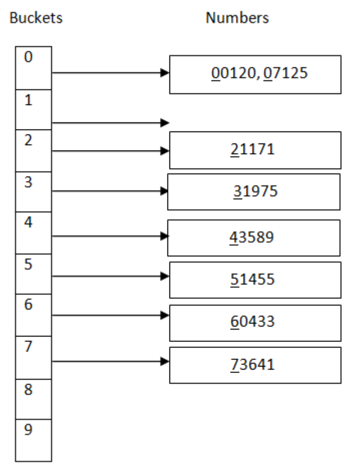


Figure 1.12: Example of Radix Sort

Now see the 5th digit from the right. Rearrange the list again according to the buckets; we will get 00120, 07125, 21171, 31975, 43589, 51455, 60433, and 73641. This is the final sorted list.

Radix sort preserves the stability. Radix sort [20] is not an adaptive sort. The radix sort algorithm can be depicted as follows:

Algorithm 9 Radix Sort Algorithm

INPUT: Unsorted list of n items

OUTPUT: Sorted list of n items

RADIX SORT(X, n)
for $i = 1$ to n **do**
  Any stable sort algorithm is used to sort array X one digit i.
end for

Time Complexity of Radix Sort

• **Best Case:** Radix sort algorithm requires 'k' to pass over the list of 'n' numbers. So the radix sort complexity = $\Omega(nk)$, where 'k' is the input number of elements which is used for sorting in the algorithm, and 'k' is the number of digits in the longer input number. We don't know that how 'k' big can be, sometimes 'k' can be large and it can be small. When the 'k' is small then $\Omega(nk)$ = $\Omega(n)$.

• **Average Case:** The average case time complexity of radix sort is $\Theta(kn)$, which is same as the worst case of radix sort.

• **Worst Case:** We don't know whether 'k' is large or 'n' is large, so we keep them both. So radix sort worst case complexity = $O(kn)$.

Space Complexity of Radix Sort

Auxiliary Space Complexity of radix sort is $O(k + n)$, where 'k' is the number of buckets and 'n' is the input elements.

1.3.9 Bucket Sort

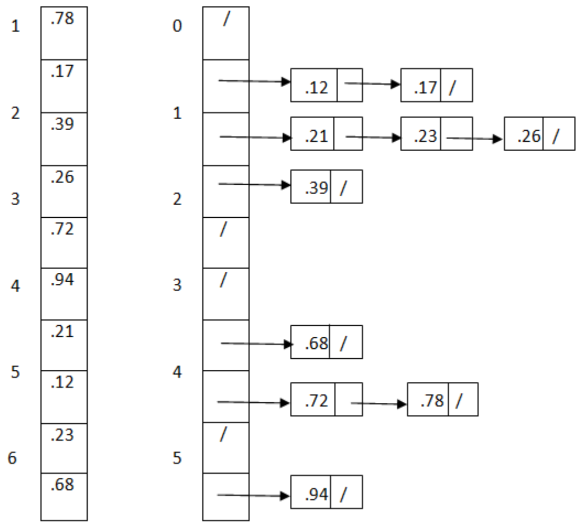


Figure 1.13: Example of Bucket Sort

A Bucket sort [20] is not really a sorting algorithm because it's not really sorts anything. It partitions an array of elements into a number of buckets, and then buckets are individually sorted, either recursively or we use some other algorithm. Bucket sort preserves the stability. Bucket sort is not an adaptive sorting algorithm. It is non-comparison based sorting algorithm. The bucket sort algorithm can be depicted as follows: In the algorithm length$[B] = n$, where 'n' is the input number of elements which is used for sorting in the algorithm.

Algorithm 10 Bucket Sort Algorithm

INPUT: Unsorted list of n items

OUTPUT: Sorted list of n items

$n \leftarrow$ length$[B]$

for $i = 1$ to n **do**

Insert $B[i]$ into list $A[B[i]/b]$, where b is the bucket size.

end for

for $i = 0$ to n-1 **do**

Sort list A with Insertion sort

Concatenate the lists $A[0]$, $A[1]$, ...$A[n$-1$]$ together in order.

end for

Time Complexity of Bucket Sort

• **Best Case:** We have a list of 'n' elements. Going through the list and put the elements in the correct bucket = $\Omega(n)$. Merging the buckets = $\Omega(k)$, where 'k' is the number of buckets.

$$\Omega(n) + \Omega(k) = \Omega(n + k) \tag{1.3.20}$$

• **Average Case:**

$$\Theta(n) + \Theta(k) = \Theta(n + k) \tag{1.3.21}$$

• **Worst Case:** If every element belongs to the same buckets, then what will happen? In the worst case, this would imply that we would have $O(n^2)$ performance, because if all the element belongs to the same bucket, then use insertion sort on 'n' elements which is $O(n^2)$.

Space Complexity of Radix Sort

The auxiliary space complexity of bucket sort is $O(nk)$, because the bucket sort needs $O(n)$ auxiliary space for the array and 'k' is the number of buckets.

1.4 Introduction to GPU

GPU stands for Graphics Processing Unit. In the 1999-2000 computer scientist started using the GPU to extend the range of scientific domain [21]. The term GPU was familiarize in 1999 by NVIDIA. The world first GPU was a Geforce 256. To do the GPU programming, we require the use of graphics APIs such as OpenGL and WebGL [22]. In 2002 James Fung developed OpenVIDIA. It is used for parallel GPU computer vision. The projects of OpenVIDIA implement computer observation algorithms run on graphics hardware such as OpenGL, Cg and CUDA-C [23]. In November 2006 NVIDIA launched CUDA (Compute Unified Device Architecture) [24]. It is an API (Application Programming Interface) that allows coding the algorithms for execution on Geforce GPUs using C as a high level programming language [25]. CUDA can use with other languages also see with the help of a diagram in Figure 1.14 [26].

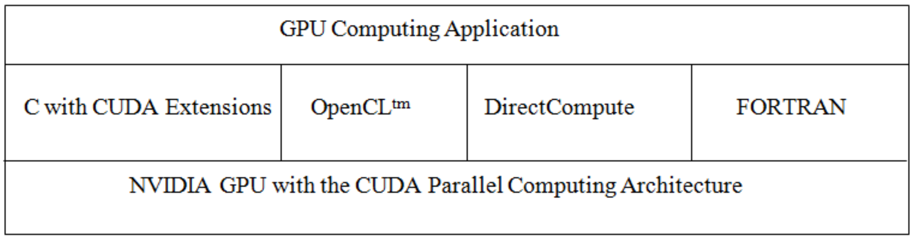


Figure 1.14: CUDA support the various languages

The parallel execution of sorting algorithms using graphics processing unit (GPU) is allowed by general purpose computing, on graphics processing unit (GPGPU) [32]. Parallel sorting algorithms having highly code are handled by GPU as a co-processor. The framework of NVIDIAs compute unified device architecture (CUDA) release that provides free programmability of GPUs [23]. The number of stream processors are used for floating point calculations with CPUs. Parallelism is limited in a stream processor like ALUs [33]. 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 [34] compiler, functions executed on the device only supports CUDA C. So we are going to develop the parallel version of merge and quick sort using GPU in the framework of NVIDIA's CUDA C [35].

The parallel computing with CUDA organizes concepts of Grid, Block and Thread which can be defined as:

- Grid: this is the group of blocks. There is no synchronization between the blocks.

• Block: This is the group of threads.

• Thread: This is the execution of the kernel.

Nowadays GPU is a big domain in parallel computing [27]. GPU works in many spheres of our daily lives. The architecture of GPU is shown in Figure 1.15.

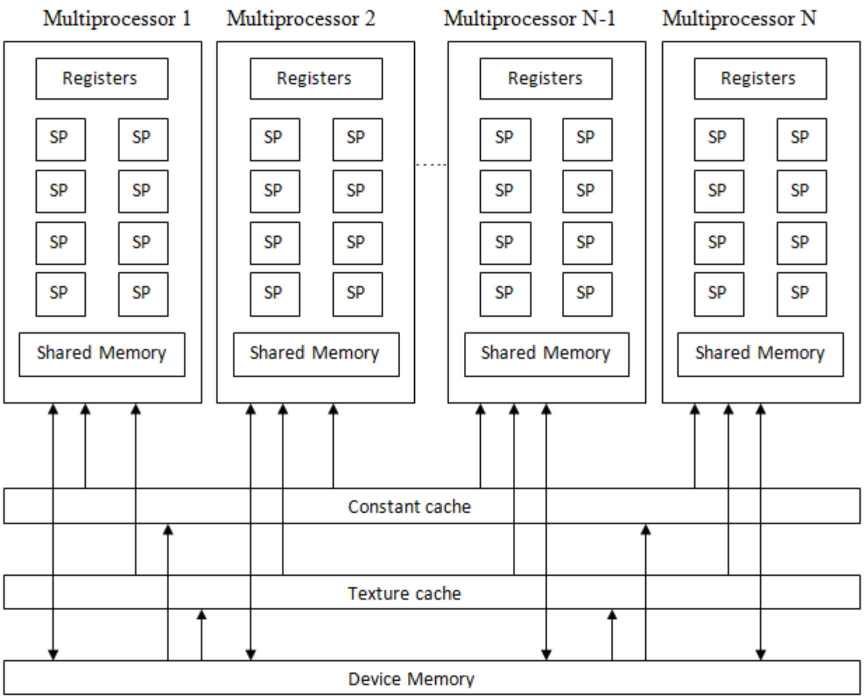


Figure 1.15: GPU Architecture

1.4.1 Scalable Programming Model

The scalable programming model [28] allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions. A scalable programming model is shown in Figure 1.16.

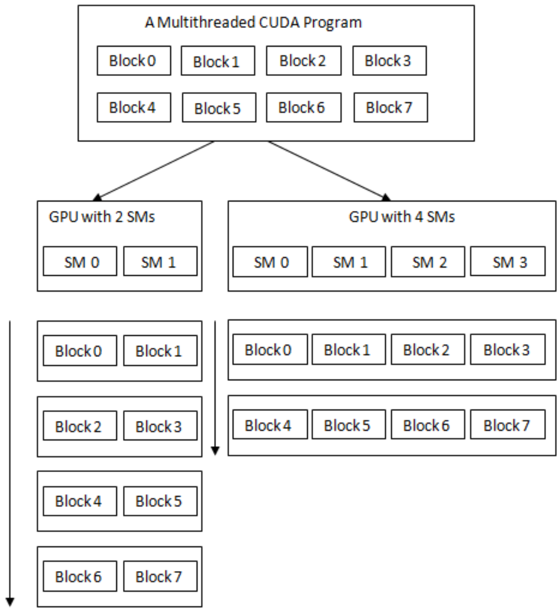


Figure 1.16: Scalable Programming Model

1.5 Motivation to our approaches

Nowadays GPU is in big demand in parallel computing. Numerous researchers are working on the GPU. The programmability of the GPU is rising in the world. The rising GPU has enabled the threshold. The following point makes the user to work on the GPU rather than the CPU.

- GPUs are faster than the CPU.
- GPUs are commercially successful.
- GPUs have a disruptive innovation path.
- The GPU programming model emerging.
- GPU is massively parallel.
- It has hundreds of cores.
- It has thousands of threads.
- It is cheap.
- It is highly available.

1.6 Sorting Benchmark and Standard Dataset

In this thesis, sorting benchmark and standard dataset are used to test both the versions (sequential and parallel) of sorting algorithms. Six types of test cases are used by sorting benchmark which are Uniform, Sorted, Zero, Bucket, Gaussian, and Staggered [24][29][30]. The size of input data is varied from 100 to 10000000 and the thread in the multiple of 2 from 1 to 1024.

1. Uniform test case: Input values are picked randomly from 0 to 2^{32}.

2. Gaussian test case: This test case consists the distribution of data created by taking the average of four randomly values picked from the uniform distribution.

3. Zero test case: A constant value is used as input by this test case.

4. Bucket test case: For $p \in n$, the input of size n is split into p blocks, such that the first n/p^2 elements in each of them are random numbers in [0, $2^{31}/p$-1], the second n/p^2 elements in [$2^{31}/p$, $2^{31}/p$ - 1], and so forth.

5. Staggered test case: For $p \in n$, the input of size n is split into p blocks such that if the block index is $i \leq p/2$ all its n/p elements are set to a random number in [(2i-1)$2^{31}/p$, (2i)($2^{31}/p$ - 1)].

6. Sorted test case: Sorted uniformly distributed value has been taken as input.

Some Sorting algorithms are evaluated on four cases of standard dataset [31]. The dataset contains the 1010228 items. The four cases of dataset are:

1. Random with repeated data (Random data). **2.** Reverse sorted with repeated Data (Reverse sorted data). **3.** Sorted with repeated data (Sorted data). **4.** Nearly sorted with repeated data (Nearly sorted data)

1.7 Hardware

In order to run proposed and designed algorithms Window 7 32-bit operating system Intel core i3 processor 530@ 2.93 GHz machine is used. System build with GeForce GTX 460 graphic processor with (7 multiprocessors $\times$ (48) CUDA cores\MP) = 336 CUDA cores. System body consists 1536 threads per multiprocessor and 1024 threads per block. CUDA runtime version of the system is 6.0. 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. Total number of registers available per block is 32768 and warp size is 32. Maximum sizes of each dimension of a block are 1024 $\times$ 1024 $\times$ 64 and maximum size of each dimension of a grid is 65535 $\times$ 65535 $\times$ 65535.

1.8 Thesis road map

The main contribution of the thesis is explained via the following chapters:

1. Performance Enhancement of Existing Sorting Algorithms using GPU Computing.

2. Performance enhancement of parallel OETSN using GPU computing.

3. Performance Enhancement of Library Sort Algorithm with Uniform Gap Distribution.

4. Performance Enhancement of Library Sort Algorithm with Non-Uniform Gap Distribution.

5. Performance Enhancement of Bucket Sort using Hybrid Algorithm.

6. Performance Enhancement of Bubble Sort using GPU Computing.

Chapter 2

Performance Enhancement of Existing Sorting Algorithms using GPU Computing

2.1 GPU Count Sort using CUDA

At this time multi core CPUs [56] are available in the market. The multi core CPUs are not satisfactory to solve the high data computation task. So, recently GPU [57-58] introduced to solve these problems. The GPU is having the multi core processors thousands of threads running concurrently [59]. To program a GPU the basic need is the parallel platform like NVIDIAs CUDA. The prime difference between OpenCL and CUDA is that: 1) the cuda is specifically for Nvdia hardware, but opencl is run on different hardware which conforms to its standard [60-61]. There are GPU and CPU available, but for to achieve high performance, primarily focuses on the GPUs. Count sort is a non-comparison based sorting algorithm [62-63]. The contribution of the paper is as follows.

• The main content of this section is based on count sort. The problem with count sort is that, it is not recommended for larger sets of data because it depends on the range of key elements.

- The drawback of count sort has been taken as research aspect.
- Sorting benchmark is used to test the parallel and sequential count sort.
- The speedup achieved by parallel count sort is also measured in this chapter.

Bajpai *et al* presented the modified version of counting sort called E-Counting sort. In E-Counting sort some efficiency has been improved by author and execution time with original one [66].
Svenningsson *et al* investigated two sorting algorithms which are counting sort and a variation occurrence sort. The suggested algorithms are used to remove duplicate elements and examine their suitability running on the GPU [67].
Sun *et al* depicted the design issue of data parallel implementation of count sort using GPU with CUDA. The parallel version is more efficient than sequential [68].

Objective

Sorting is considered a very important application in many areas of computer science. Nowadays parallelization of sorting algorithms using GPU computing, on CUDA hardware is increasing rapidly. The objective behind using GPU computing is that the users can get, the more speedup of the algorithms.

Methods

In this chapter, we have focused on count sort. It is very efficient sort with time complexity $O(n)$. The problem with count sort is that, it is not recommended for larger sets of data because it depends on the range of key elements. In this chapter this drawback has been taken for the research concern and we parallelized the count sort using GPU computing with CUDA.

Findings

We have measured the speedup achieved by the parallel count sort over sequential count sort. The sorting benchmark has been used to test and measure the performance of both the versions of count sort (parallel and sequential). The sorting benchmark has six types of test cases which are uniform, bucket, Gaussian, sorted, staggered and zero. In this chapter, our finding is that we have tested the parallel and sequential count sort on a larger sets of data which vary from n=1000 to n=10000000.

2.1.1 Implementation of Sequential Count Sort

In this section the implementation results of the sequential count sort has been shown. We have implemented the algorithm on the sorting benchmark using six types of test cases. We have calculated execution time in milliseconds of the algorithm which is shown in Table 2.1. In Table 2.1 we have shown that the algorithm recommended for the large data sets as the data size has been varied from 100 to 10000000. By analyzing the Table 2.1. We can see that zero test case is more efficient compare to other test cases.

Table 2.1: Execution time in milliseconds of sequential count sort

n/Test case	Uniform	Sorted	Zero	Bucket	Gaussian	Staggered
100	1418	1248	0.001	1529	1336	1581
1000	1472	1527	0.002	1539	1368	1599
10000	1691	1679	1	1541	1461	1641
100000	1765	1868	2	1642	1763	1689
500000	1773	1968	11	1734	1861	1742
1000000	1831	1971	19	1883	1896	1795
2500000	1993	1975	41	1959	1917	1863
5000000	2342	1995	97	1994	1974	1888
7500000	2379	2096	109	1997	1991	1959
10000000	2427	2159	129	2177	2059	1999

2.1.2 Implementation of Parallel Count Sort

In the Tables 2.2 to 2.7, we have shown the parallel execution time using six types of test cases with varying data and thread size.

Table 2.2: Execution time in milliseconds of parallel count sort using uniform test case

n/T	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
100	0.044	0.040	0.040	0.039	0.036	0.036	0.035	0.034	0.033	0.031
1000	0.100	0.066	0.054	0.051	0.049	0.049	0.048	0.046	0.045	0.044
10000	0.678	0.368	0.231	0.203	0.192	0.176	0.173	0.172	0.169	0.167
100000	8.293	3.497	2.117	1.784	1.639	1.491	1.450	1.416	1.395	1.387
500000	37.467	20.145	11.708	8.796	8.007	7.816	6.997	6.923	6.814	6.711
1000000	74.799	40.351	23.544	19.053	15.985	14.724	14.358	14.188	13.149	13.362
2500000	184.719	100.631	58.557	47.843	43.137	35.742	34.434	33.035	32.907	31.596
5000000	367.033	199.474	117.197	94.633	83.796	71.566	68.537	66.966	65.874	64.917
7500000	549.743	297.629	174.571	144.056	126.228	106.157	102.674	99.820	98.722	97.298
10000000	732.190	396.518	232.582	189.154	166.405	140.392	137.161	134.435	133.611	132.594

The thread size has been varied from $T = 1$ to 1024 but we have drawn the graph of execution time using $T = 1024$ as it is not possible to show all the graphs using all the possible value of thread given in the table. In all the Figures 2.1 to 2.6, X-axis shows the execution time in milliseconds and the Y-axis shows the increasing data size. We have calculated the execution time using varying sizes of data and threads, but in the graphs, we have only shown the execution time comparison between parallel and sequential count sort using the thread value 1024. The remaining graph can be drawn in the similar manner using the possible values of threads listed in the tables.

The Table 2.3 shows the execution time in milliseconds of the parallel count sort using sorted test case. The parallel version of sorted test case is more efficient than sequential. We can see this effect in Table 2.3 and in Figure 2.2.

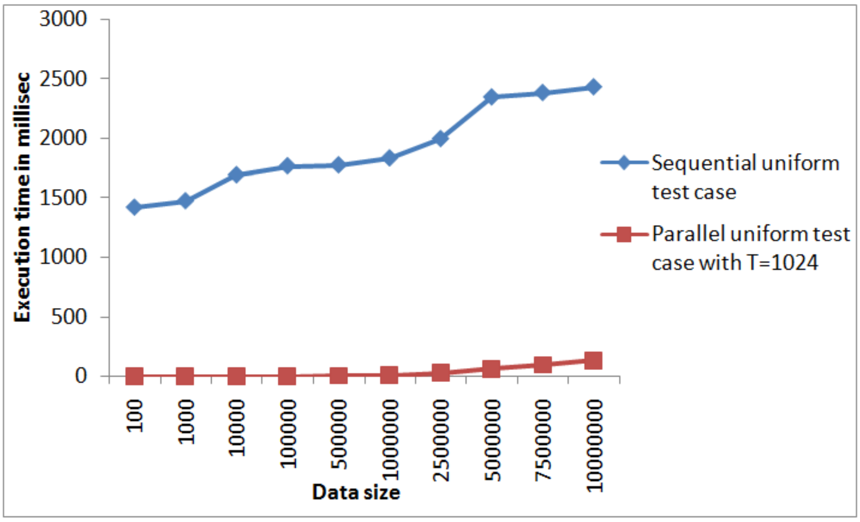


Figure 2.1: Execution time comparison between parallel and sequential count sort using uniform test case

Table 2.3: Execution time in milliseconds of parallel count sort using sorted test case

n/T	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
100	0.032	0.034	0.030	0.030	0.030	0.030	0.029	0.029	0.029	0.027
1000	0.101	0.069	0.052	0.041	0.035	0.035	0.034	0.034	0.034	0.034
10000	0.653	0.391	0.376	0.293	0.195	0.177	0.140	0.119	0.119	0.105
100000	8.206	5.695	5.493	5.424	5.298	4.634	4.481	4.354	4.223	3.950
500000	37.570	20.002	18.936	17.401	16.488	15.458	15.069	14.444	13.945	13.756
1000000	75.269	51.859	43.502	39.172	34.869	33.946	33.162	33.056	32.979	32.887
2500000	188.816	150.414	121.414	101.414	91.414	87.414	82.746	82.338	81.712	81.283
5000000	379.675	267.187	228.859	209.285	199.285	181.872	174.953	163.719	163.148	162.836
7500000	569.112	406.415	478.981	493.749	380.549	345.386	315.310	245.196	244.701	243.381
10000000	754.410	671.365	611.044	521.194	416.709	453.242	497.458	380.816	326.293	323.284

The Table 2.4 shows the execution time in milliseconds of the parallel count sort using zero test case. The parallel version of zero test case is not efficient than the sequential version of zero test case. It is because the zero means one unique number and to sort this, the sequential count sort take one count only as it is already sorted and unique. It is not in the case of the parallel count sort because in parallel, we always divide the number into a number of blocks and threads, whether the data are unique or sorted. In the Table 2.4 and Figure 2.3 we can see that sequential count is more efficient than parallel when the test case is zero.

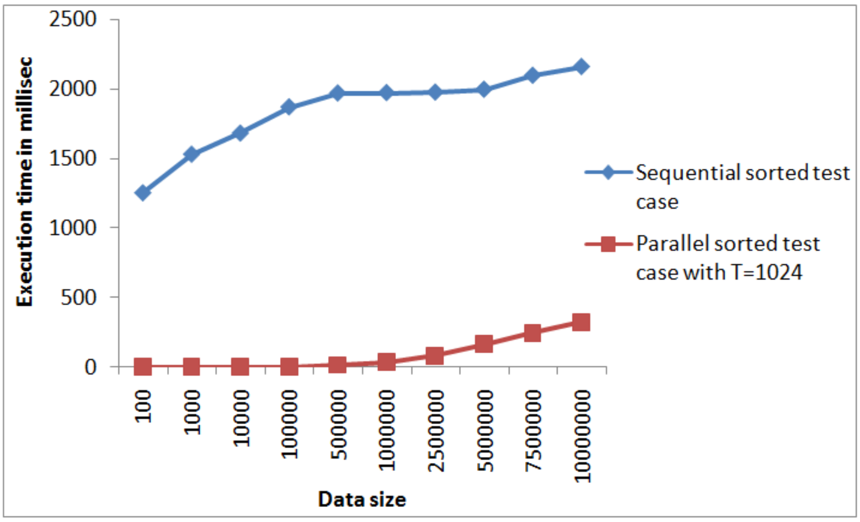


Figure 2.2: Execution time comparison between parallel and sequential count sort using sorted test case

Table 2.4: Execution time in milliseconds of parallel count sort using zero test case

n/T	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
100	0.025	0.024	0.024	0.023	0.022	0.022	0.020	0.020	0.020	0.020
1000	0.081	0.055	0.053	0.051	0.050	0.050	0.046	0.044	0.043	0.041
10000	0.631	0.361	0.336	0.336	0.334	0.324	0.319	0.300	0.300	0.299
100000	4.780	4.713	3.270	3.244	3.240	3.226	3.208	3.104	3.015	3.010
500000	38.029	21.117	18.222	17.526	16.580	15.519	14.255	14.229	13.434	13.187
1000000	75.887	42.284	36.134	34.456	32.989	31.032	30.364	30.261	30.129	30.105
2500000	188.004	105.031	90.246	86.356	85.136	83.355	82.661	81.714	80.822	80.503
5000000	373.451	207.937	180.223	171.998	167.729	166.847	163.299	162.962	162.520	161.926
7500000	559.198	311.249	270.726	259.670	251.825	247.898	244.596	243.982	241.184	240.215
10000000	745.075	413.768	360.978	343.993	334.753	330.347	324.874	323.811	322.980	321.848

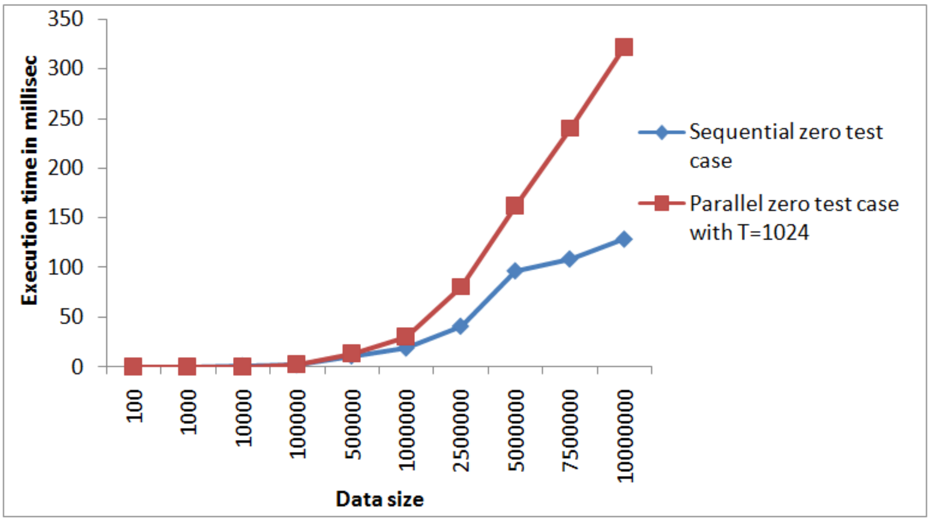


Figure 2.3: Execution time comparison between parallel and sequential count sort using zero test case

Table 2.5: Execution time in milliseconds of parallel count sort using bucket test case

n/T	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
100	0.041	0.036	0.034	0.033	0.033	0.031	0.030	0.030	0.030	0.029
1000	0.099	0.065	0.051	0.049	0.048	0.047	0.045	0.044	0.043	0.043
10000	0.677	0.368	0.232	0.200	0.188	0.169	0.169	0.168	0.161	0.160
100000	8.340	3.510	2.123	1.785	1.701	1.400	1.371	1.357	1.354	1.342
500000	37.902	21.378	10.715	8.783	7.979	6.882	6.694	6.627	6.620	6.605
1000000	75.584	42.820	21.642	18.170	15.945	14.748	14.465	14.173	14.000	13.476
2500000	187.064	107.055	100.112	95.294	85.750	35.635	34.978	33.510	31.774	30.618
5000000	371.482	211.861	107.769	89.266	79.266	69.266	68.388	66.388	61.807	60.807
7500000	556.547	316.744	160.419	137.144	117.144	106.133	102.549	101.275	98.349	97.349
10000000	740.812	421.667	213.901	180.264	140.374	137.022	136.350	135.485	126.532	125.519

The Table 2.5 shows the execution time in milliseconds of the parallel count sort using bucket test case. The parallel version of the bucket test case is more efficient than sequential. We can see this effect in Table 2.5 and in the Figure 2.4. The Figure 2.4 tells us that parallel bucket test case is having the very much less execution time in comparison to the sequential bucket test case. So in this way speedup is also increased.

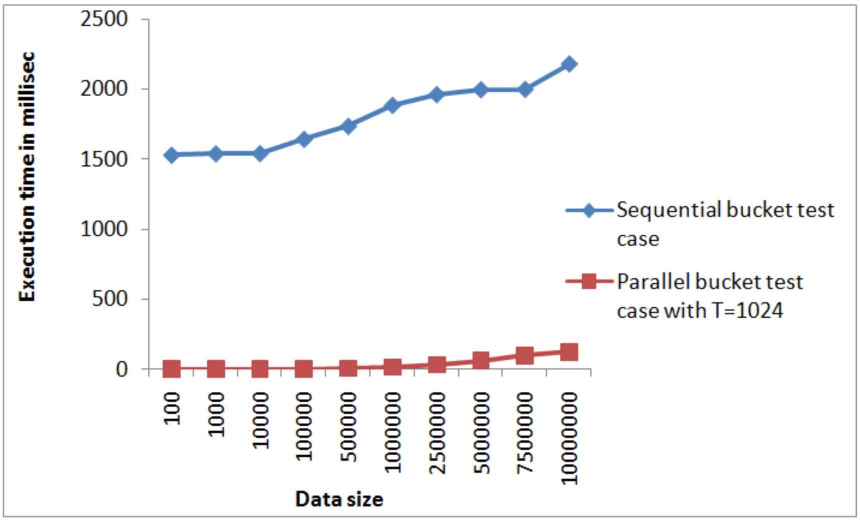


Figure 2.4: Execution time comparison between parallel and sequential count sort using bucket test case

Table 2.6: Execution time in milliseconds of parallel count sort using Gaussian test case

n/T	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
100	0.069	0.066	0.061	0.060	0.049	0.038	0.034	0.030	0.030	0.029
1000	0.099	0.066	0.050	0.042	0.036	0.033	0.032	0.030	0.030	0.026
10000	0.678	0.373	0.225	0.142	0.095	0.073	0.063	0.061	0.059	0.053
100000	8.334	3.565	2.060	1.192	0.712	0.461	0.358	0.323	0.322	0.316
500000	37.792	20.576	11.450	5.907	3.475	2.204	1.677	1.503	1.404	1.220
1000000	75.471	41.073	22.872	12.979	6.902	4.398	3.321	3.986	2.056	1.607
2500000	186.491	102.587	57.103	32.703	20.177	13.192	8.304	7.945	7.582	6.068
5000000	370.635	202.987	114.191	64.945	37.993	25.003	18.469	15.911	14.150	13.442
7500000	555.043	303.311	169.633	97.741	57.609	35.837	26.336	24.296	22.644	20.694
10000000	738.959	403.503	226.569	129.902	75.493	47.312	36.312	32.531	31.158	30.824

The Table 2.6 shows the execution time in milliseconds of the parallel count sort using Gaussian test case. The parallel version of the Gaussian test case is more efficient than sequential. We can see this effect in Table 2.6 and in Figure 2.5. The Figure 2.5 tells us that parallel Gaussian test case is having the very much less execution time in comparison to the sequential Gaussian test case.

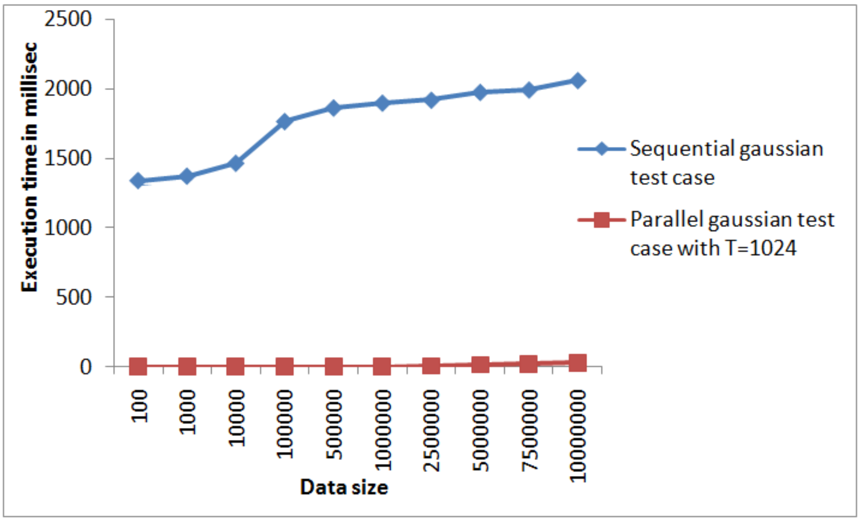


Figure 2.5: Execution time comparison between parallel and sequential count sort using gaussian test case

Table 2.7: Execution time in milliseconds of parallel count sort using staggered test case

n/T	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
100	0.061	0.054	0.051	0.051	0.050	0.050	0.040	0.040	0.030	0.031
1000	0.099	0.066	0.052	0.045	0.044	0.044	0.043	0.042	0.041	0.040
10000	0.649	0.572	0.454	0.430	0.429	0.401	0.398	0.395	0.380	0.321
100000	7.753	5.684	4.302	3.965	3.864	3.570	3.389	3.291	3.266	3.226
500000	35.234	19.543	9.162	8.583	7.532	6.983	6.845	6.731	6.431	6.231
1000000	73.652	40.752	19.654	17.875	16.986	15.877	14.865	14.542	14.362	14.123
2500000	183.755	101.766	95.864	88.885	81.777	32.876	31.886	30.766	29.654	29.123
5000000	365.754	208.676	105.665	85.754	79.765	65.888	64.886	63.999	62.665	61.664
7500000	551.886	303.768	156.776	134.776	114.976	101.765	97.544	95.765	94.765	94.123
10000000	735.766	417.655	208.654	175.433	132.876	128.654	125.876	121.765	115.764	104.654

The Table 2.7 shows the execution time in milliseconds of the parallel count sort using Gaussian test case. The parallel version of the staggered test case is more efficient than sequential. We can see this effect in Table 2.7 and in Figure 2.6.

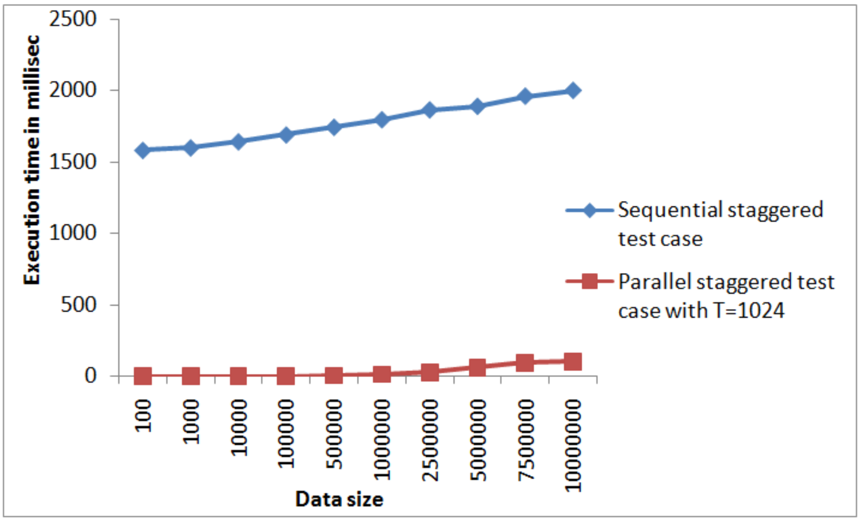


Figure 2.6: Execution time comparison between parallel and sequential count sort using staggered test case

2.1.3 Measurement of Speedup

Now we will show the speedup of parallel count sort in comparison to the sequential. As the speedup measures performance gain achieved by parallelizing a given application over sequential application [69]. We have implemented the count sort using the varying data size and number of threads. Here we have only shown the speedup achieved by parallel count sort with $n = 10000000$, $n = 7500000$, $n = 5000000$, $n = 2500000$ and $n = 1000000$ data size, for the remaining values of 'n' we can find out speedup in the similar manner.

Table 2.8: Speedup achieved by parallel count sort using different types of test cases with n=7500000

Test case	**T=1**	**T=2**	**T=4**	**T=8**	**T=16**	**T=32**	**T=64**	**T=128**	**T=512**	**T=1024**
Sorted	3.689	4.158	4.376	4.586	5.508	6.069	6.648	8.549	8.666	8.786
Gaussian	3.587	6.564	11.737	20.371	34.561	55.556	75.599	81.947	87.926	96.213
Uniform	4.327	7.993	13.628	16.514	18.847	22.411	23.171	23.833	24.098	24.451
Bucket	3.588	6.305	12.449	14.561	17.047	18.816	19.474	19.719	20.305	20.514
Staggered	3.549	6.449	12.496	14.535	17.038	19.251	20.083	20.456	20.672	20.899

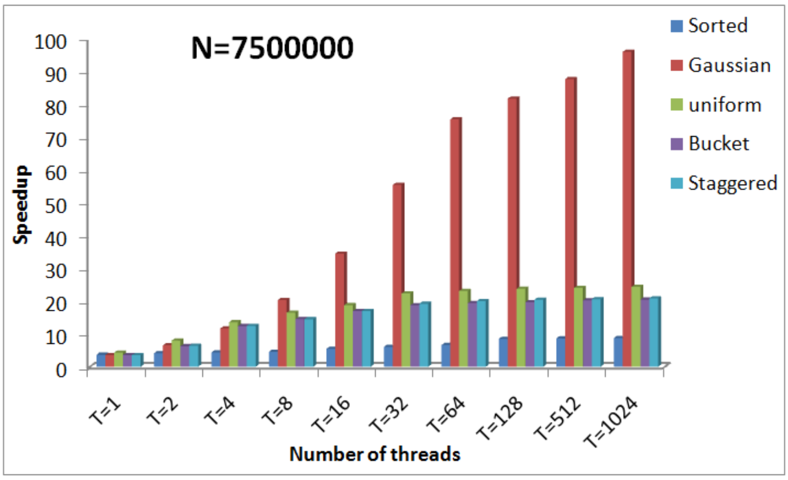


Figure 2.7: Speedup achieved by parallel count sort using different types of test cases with n=7500000

In the Tables 2.8, 2.9, 2.10, 2.11 and 2.12, we have measured the speedup achieved by the parallel count sort using the different types of test cases. In all the Tables, we can see that zero test case is not taken to measure the speedup. It is because the parallel zero test case is less efficient than sequential. The reason is explained earlier. The Figures 2.7, 2.8, 2.9, 2.10 and 2.11 have been drawn using the Tables 2.8-2.12. In all the Figures X-axis represents the speedup achieved by the algorithm and Y-axis represents the number of threads. By analyzing all the Figures, we can see that if we increase the number of threads the speedup is also increases. And in all the Figures Gaussian test case has achieved more speedup compared to other test cases.

Table 2.9: Speedup achieved by parallel count sort using different types of test cases with n=10000000

Test case	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
Sorted	2.862	3.216	3.534	4.152	5.182	5.764	5.892	5.998	6.617	6.679
Gaussian	2.787	5.103	9.088	15.851	27.274	43.519	56.703	63.293	66.081	66.798
Uniform	3.315	6.121	10.435	12.831	14.585	17.287	17.695	18.053	18.165	18.304
Bucket	2.939	5.163	10.178	12.077	15.509	15.888	15.967	16.068	17.201	17.344
Staggered	2.717	4.786	9.581	11.395	15.044	15.538	15.881	16.417	17.268	18.123

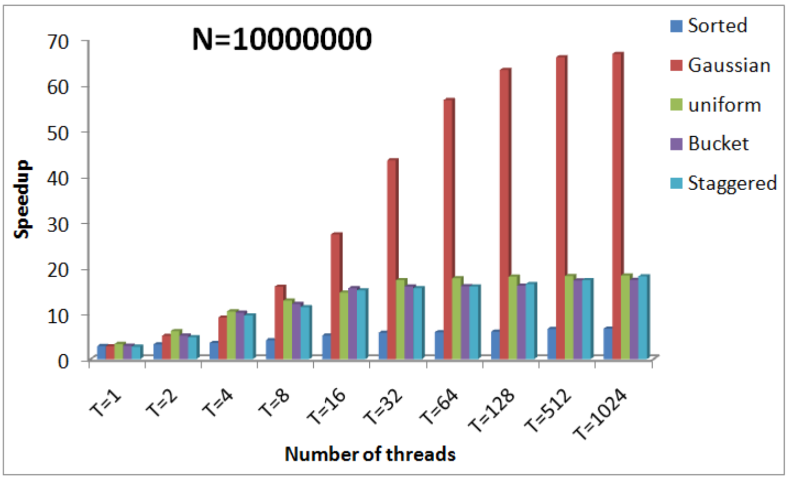


Figure 2.8: Speedup achieved by parallel count sort using different types of test cases with n=10000000

Table 2.10: Speedup achieved by parallel count sort using different types of test cases with n=5000000

Test case	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
Sorted	5.254497	7.466667	8.717176	9.532443	10.01077	10.96927	11.40304	12.18551	12.22819	12.25157
Gaussian	5.325988	9.724768	17.28682	30.39504	51.95757	78.9505	106.8822	124.0665	139.5039	146.8582
Uniform	6.38089	11.74089	19.98341	24.74813	27.94895	32.72502	34.17155	34.97321	35.55292	36.07679
Bucket	5.367693	9.411845	18.50262	22.3378	25.15589	28.78768	29.15716	30.03555	32.2617	32.79226
Staggered	5.161934	9.047536	17.86779	22.01647	23.66953	28.65469	29.09719	29.50055	30.12846	30.61754

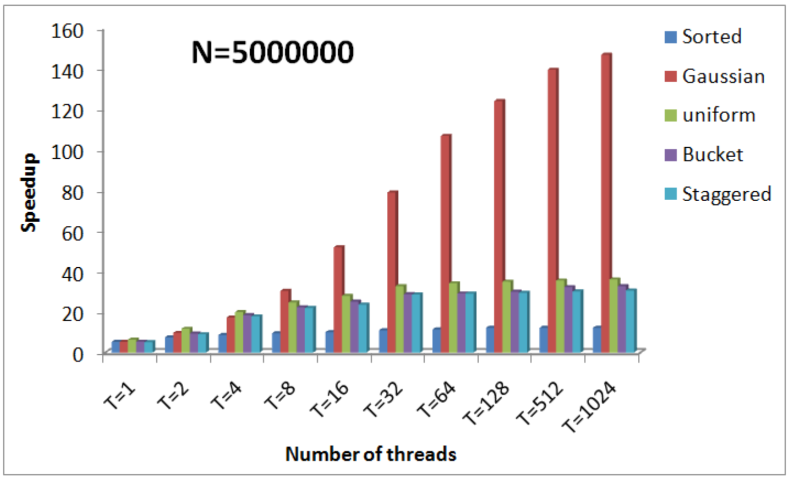


Figure 2.9: Speedup achieved by parallel count sort using different types of test cases with n=5000000

Table 2.11: Speedup achieved by parallel count sort using different types of test cases with n=2500000

Test case	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
Sorted	10.45991	13.1304	16.26662	19.47458	21.60494	22.59357	23.86822	23.98663	24.17038	24.2977
Gaussian	10.2793	18.68656	33.57112	58.61777	95.00785	145.3139	230.841	241.2967	252.8213	315.9162
Uniform	10.78939	19.80505	34.03518	41.65739	46.20117	55.76106	57.87844	60.32947	60.5649	63.07805
Bucket	10.47238	18.29901	19.56813	20.55748	22.8456	54.97427	56.00727	58.45935	61.65334	63.98287
Staggered	10.13851	18.30676	19.43378	20.95957	22.78159	56.66748	58.4269	60.55465	62.82458	63.97006

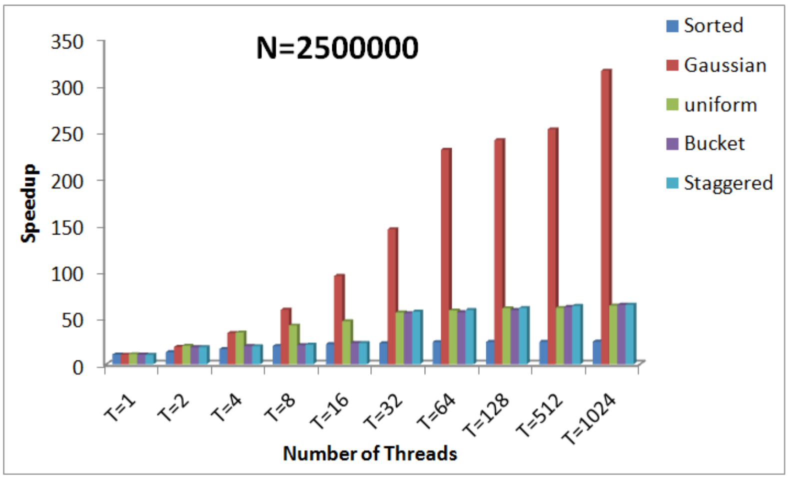


Figure 2.10: Speedup achieved by parallel count sort using different types of test cases with n=2500000

Table 2.12: Speedup achieved by parallel count sort using different types of test cases with n=1000000

Test case	T=1	T=2	T=4	T=8	T=16	T=32	T=64	T=128	T=512	T=1024
Sorted	26.18623	38.00683	45.30774	50.317	56.52545	58.063	59.43467	59.62546	59.765	59.93167
Gaussian	25.12211	46.16132	82.89624	146.0799	274.7189	431.0753	570.8532	475.6324	922.222	1179.738
Uniform	24.47887	45.37677	77.77023	96.09797	114.5477	124.3513	127.526	129.0489	139.2475	137.0302
Bucket	24.91273	43.97504	87.00669	103.633	118.0935	127.6792	130.1749	132.8559	134.5037	139.7292
Staggered	24.37133	44.04681	91.33187	100.4179	105.6728	113.06	120.7534	123.4356	124.9826	127.0976

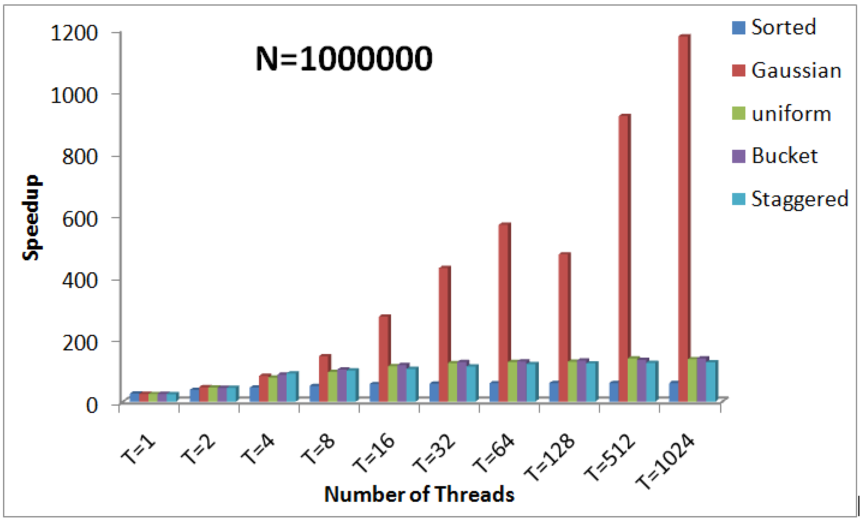


Figure 2.11: Speedup achieved by parallel count sort using different types of test cases with n=1000000

The main conclusion of this section is that parallel count sort has better experimental results over sequential using five types of test case which has explained earlier. We have implemented our code of the sequential count sort algorithm in C language. And the parallel count sort algorithm has done using GPU computing with CUDA hardware.

2.2 GPU Merge Sort using CUDA

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 2.12, 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.

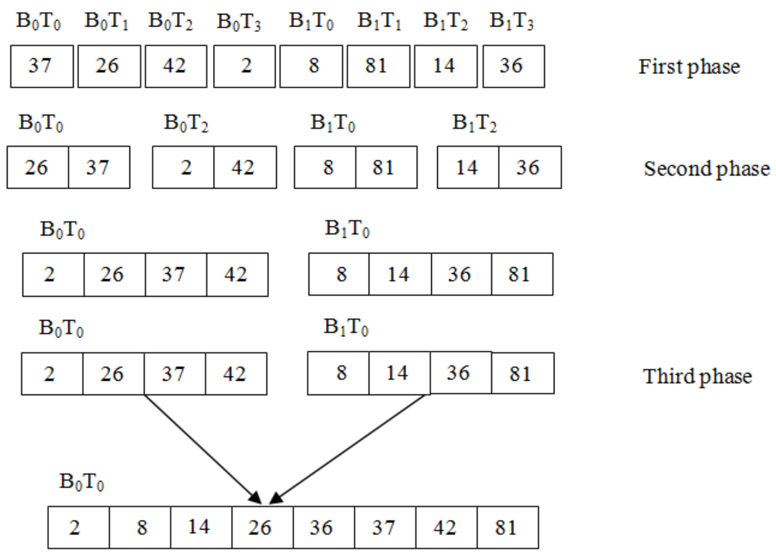


Figure 2.12: Example of parallel merge sort

2.3 GPU Quick Sort using CUDA

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 [29] 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 [29].

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.

2.3.1 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 $O((n/p)\log(n/p))$ time. During each phase $O(n)$ comparisons are performed and time $O(n)$ is spent in communication [40]. So the formal representation of parallel run time is shown in equation (2.3.1).

$$T_p = O\left(\frac{n}{p}log\frac{n}{p}\right) + O(n) + O(n) \tag{2.3.1}$$

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.
(iv) Perform the global rearrangement.
Quick sort takes time $O(\log p)$ to choose the pivot, it will take $O(n/p)$ in the second step, the third step takes time $O(\log p)$, and the fourth step takes time $O(n/p)$. So the formal representation of parallel run time of quick sort is $O(n/p)$ + $O(\log p)$. 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 (2.3.2).

$$T_p = O\left(\frac{n}{p}log\frac{n}{p}\right) + O\left(\frac{n}{p}logp\right) + O(log^2p) \tag{2.3.2}$$

2.3.2 Algorithms Used

We have 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)] [31].

Table 2.13: Sequential and parallel execution time in seconds of merge and quick sort using the four cases of the dataset.

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
Sequential Quick Sort	1.043904	1.219802	1.26322	72.089548
Parallel Quick Sort	0.080012	0.085014	0.085013	0.085014

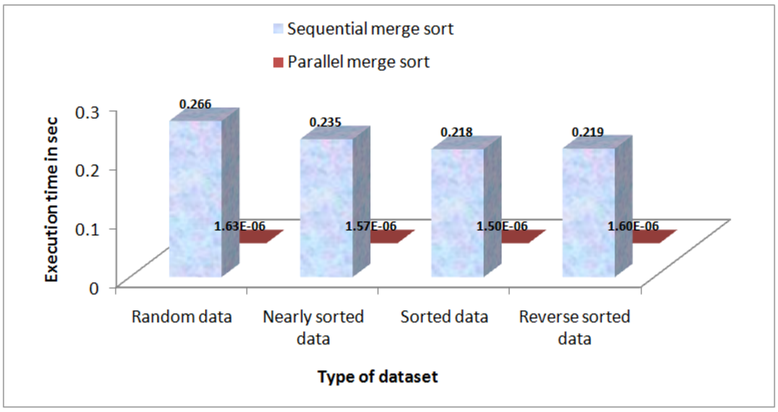


Figure 2.13: Execution time comparison between sequential and parallel merge sort

By analysing the Table 2.13, 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 2.13 and 2.14, 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.

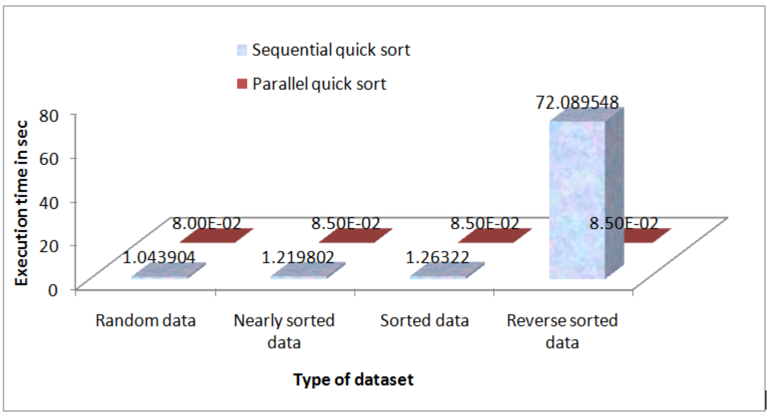


Figure 2.14: Execution time comparison between sequential and parallel quick sort

2.3.3 Memory Occupied by Merge and Quick Sort

We have calculated the space complexity for the every case of a dataset of merge and quick sort algorithm. In the Figure 2.15 and 2.16, 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 2.14.

Table 2.14: Sequential and parallel memory in bytes of merge sort using the random dataset.

Algorithms	M_{ip}	M_{is}	M_{op}	M_{os}	M_c	M_w	Total
Sequential Merge Sort	4040924	4932283	4	4932283	76800	4040940	18023234
Parallel Merge Sort	4040924	4932283	4040924	4932283	110592	102360	18159366

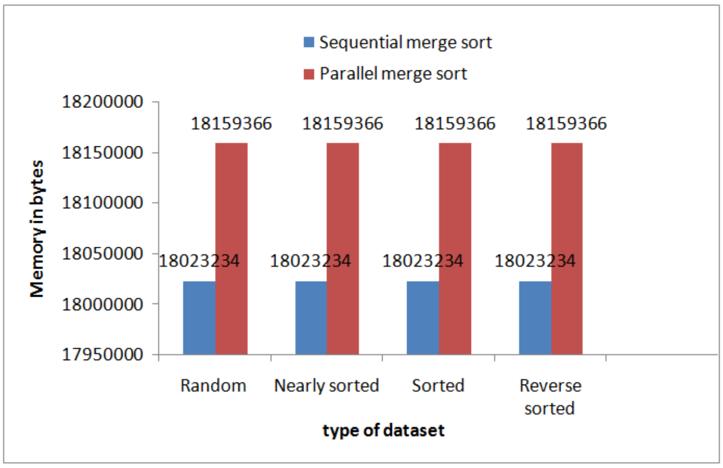


Figure 2.15: Memory comparison between sequential and parallel merge sort

Quick Sort

Space complexity of quick sort is shown in Table 2.15 using all the four cases of the dataset .

Table 2.15: Sequential and parallel memory in bytes of quick sort

Data Set	**Sorting Algorithms**	M_{ip}	M_{os}	M_{op}	M_{os}	M_c	M_w	**Total**
Random	Sequential Quick Sort	4040924	4932283	4	4932283	76288	97756	14079538
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675840	1422912	20045166
Nearly Sorted	Sequential Quick Sort	4040924	4932283	4	4932283	76288	97468	14079250
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675840	1590880	20213134
Sorted	Sequential Quick Sort	4040924	4932283	4	4932283	76288	97756	14079538
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675840	1590880	20213134
Reverse Sorted	Sequential Quick Sort	4040924	4932283	4	4932283	76288	99366	14081148
	Parallel Quick Sort	4040924	4932283	4040924	4932283	675840	1590880	20213134

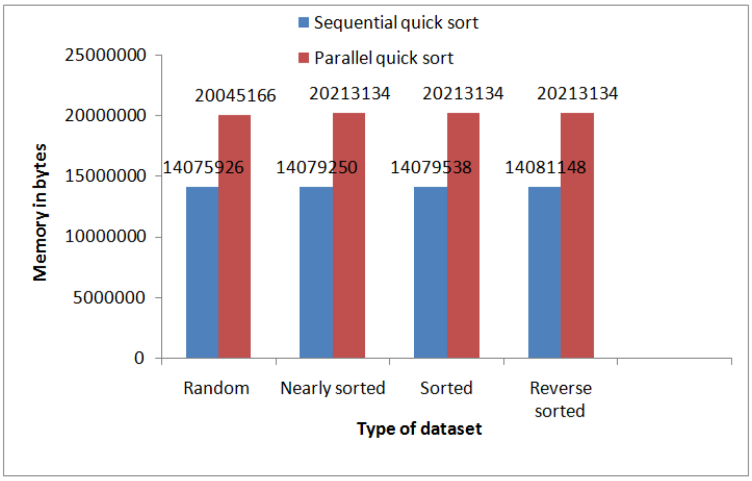


Figure 2.16: Memory comparison between sequential and parallel quick sort

By analysing the Figure 2.15 and 2.16 and Table 2.14, 2.15, 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.

2.4 Existing Sorting Algorithms on a Standard Dataset

The goal of this section is to test the various existing sorting algorithms and to evaluate the total space complexity of various sorting algorithms on a standard dataset. Sorting algorithms are evaluated on four cases of standard dataset [T10I4D100K(.gz)][31].

2.4.1 Execution time testing of various sorting algorithms

The execution time in seconds of various sorting algorithms using the standard dataset is represented in Table 2.16.

Table 2.16: Execution Time of various sorting algorithm in seconds

Algorithms	**Random**	**Reverse**	**Sorted**	**Nearly Sorted**
Insertion sort	228.136	663.768	0.005	1.196
Selection sort	479.999	460.756	415.082	422.247
Bubble sort	1457.88	2561.09	0.002	3.96
Heap sort	0.374	0.26	0.223	0.235
Shell sort	0.514	0.314	0.119	0.275
Count sort	0.114	0.109	0.088	0.104
Quick sort	2.403	72.52	1.32	2.444
Merge sort	0.327	0.161	0.14	0.151
Radix sort	0.197	0.156	0.151	0.385

Insertion Sort

We have plotted the Figure 2.17 by using the Table 2.16. By examining this figure, we can see that insertion sort takes less time when data is in sorted and nearly sorted order. And the insertion sort takes more time when the data is reverse sorted.

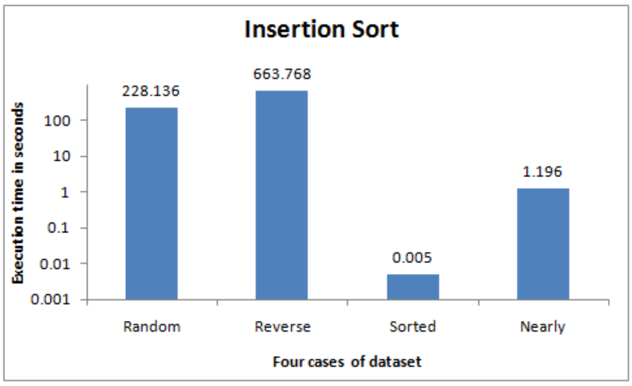


Figure 2.17: Execution time of insertion sort

Selection Sort

We have plotted the Figure 2.18 by using the Table 2.16. By examining this figure, we can see that selection sort takes less time when data is sorted and takes more time when data is in random order.

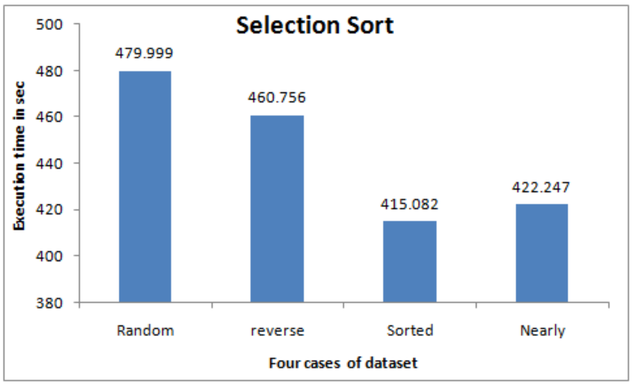


Figure 2.18: Execution time of selection sort

Bubble Sort

We have plotted the Figure 2.19 by using the Table 2.16. By examining this figure, we can see that bubble sort takes less time when data is in sorted order, and it takes more time when data is in random order.

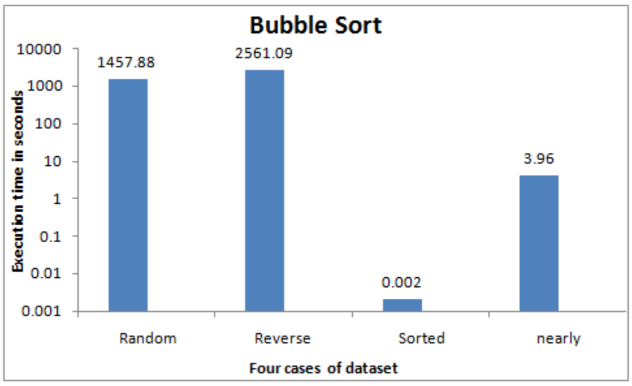


Figure 2.19: Execution time of bubble sort

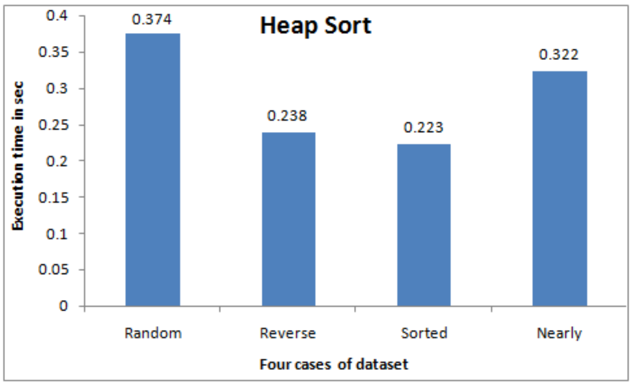


Figure 2.20: Execution time of heap sort

Heap Sort

We have plotted the Figure 2.20 by using the Table 2.16. By examining this figure, we can see that heap sort takes less time when data is in nearly sorted order, and it takes more time when data is in a random order.

Shell Sort

We have plotted the Figure 2.21 by using the Table 2.16. By examining this figure, we can see that shell sort takes less time when data is in sorted order, and it takes more time when data is in random order.

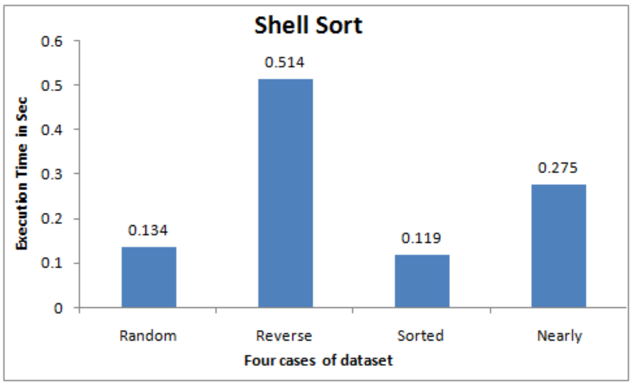


Figure 2.21: Execution time of shell sort

Count Sort

We have plotted the Figure 2.22 by using the Table 2.16. By examining this figure, we can see that count sort takes less time when data is in random order, and it takes more time when data is in sorted order.

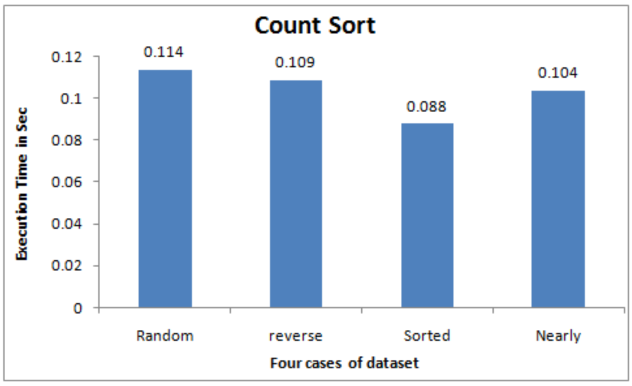


Figure 2.22: Execution time of count sort

Quick Sort

We have plotted the Figure 2.23 by using the Table 2.16. By examining this figure, we can see that quick sort takes less time when data is in sorted order, and it takes more time when data is in reverse sorted order i.e. the worst case of quick sort occur when data is in reverse sorted order.

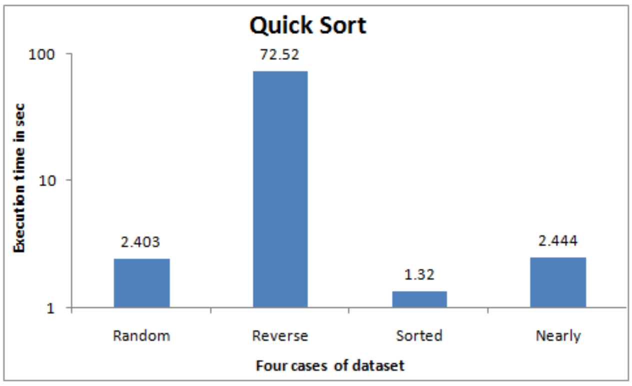


Figure 2.23: Execution time of quick sort

Merge Sort

We have plotted the Figure 2.24 by using the Table 2.16. By examining this figure, we can see that merge sort takes less time when data is in nearly sorted order, and it takes more time when data is in random order.

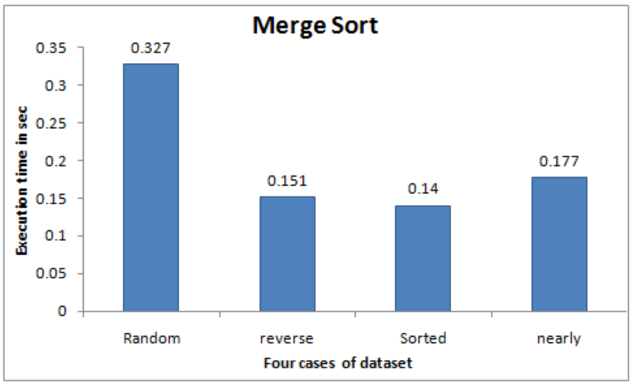


Figure 2.24: Execution time of merge sort

Radix Sort

We have plotted the Figure 2.25 by using the Table 2.16. By examining this figure, we can see that radix sort takes less time when data is in sorted order, and it takes more time when data is in reverse sorted order. In the entire Figure 2.17 to Figure 2.25, the X-axis represented the four cases of dataset in which we have tested

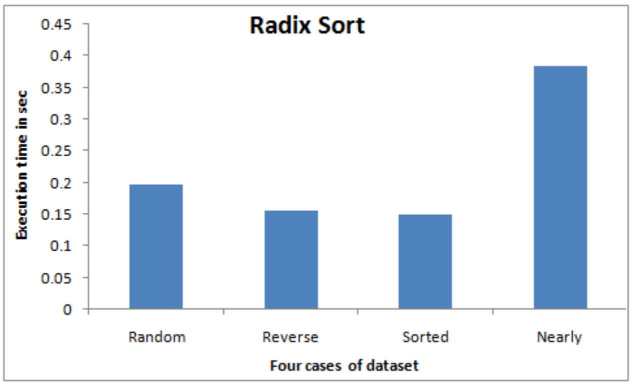


Figure 2.25: Execution time of radix sort

the various sorting algorithms, and the Y-axis represented the execution time in seconds of various sorting algorithms. All the above discussed sorting algorithms implemented in C-language. The programs is designed at Borland C++ 5.02 compiler and executed on Intel I5 processor, and the programs running at 2.2 GHz clock speed.

2.4.2 Memory testing of various sorting algorithms

In Table 2.17 we have summarized the auxiliary space complexity of the various sorting algorithms. On the basis of the results obtained after the execution of the sorting algorithms we have concluded the stability and adaptivity of the various sorting algorithms. In Table 2.17, we have shown the auxiliary space complexity taken by the various sorting algorithms, but the space complexity is not only limited to auxiliary space. It is the total space taken by the program which includes the following.

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)

M_{ip}: For M_{ip}, we have to allocate memory of four bytes for each variable (element) as we are having total of 1010228 elements so it will consume 1010228 $\times$ 4 = 4040912 bytes, again to input these items in an array we have an index variable 'a' will of four bytes so it will be total of 4040912 bytes + 4 bytes of file pointer = 4040916 bytes , and 8 bytes are used for variable declared in the program so total space complexity taken by the M_{ip} = 4040924 bytes. And the M_{ip} will be same for all the discussed sorting algorithms in all four cases of dataset, because we are using the 1010228 elements for all the four cases of dataset and for all the discussed sorting algorithms.

M_{is}: We will get this input as storage file in secondary storage , but in file we store this data a stream of bytes in character for this it will have slightly larger memory in comparison to primary memory. And the M_{is} are same for all the discussed sorting algorithms in all four cases of dataset, because we are using the 1010228 elements for all the four cases of dataset and for all the discussed sorting algorithms.

M_{op}: As we get the result either in input variable or in temporary variable so it will not require the storage in primary memory, but as we have to write this data in to secondary storage so it will require file pointer of 4 bytes. And the M_{op} is same for all the discussed sorting algorithms in all four cases of dataset, because we are using the 1010228 elements for all the four cases of dataset and for all the discussed sorting algorithms.

M_{os}: As we get the result either in input variable or in temporary variable i.e. in borland C++ the output store in str file and the size of M_{os} will be the size of str file and it will same for all the discussed sorting algorithms in all four cases of dataset, because we are using the 1010228 elements for all the four cases of dataset and for all the discussed sorting algorithms.

M_c: To calculate this space, we have to find out the size of .exe files created in windows for the discussed sorting programs, as these program will be stored in main memory for their execution. The size of .exe file depends on the sorting algorithms.

M_w: The space complexity of M_w of an algorithm depends on the variable declared for the allocation, temporary variables and size taken by the stack.

Table 2.17: Total memory occupied by various sorting algorithms using dataset

Algorithms	M_{ip}	M_{is}	M_{op}	M_{os}	M_c	M_w	**Total**
Insertion sort	4040924	4932283	4	4932283	76288	8	13981790
Selection sort	4040924	4932283	4	4932283	76288	8	13981790
Bubble sort	4040924	4932283	4	4932283	76288	12	13981794
Heap sort	4040924	4932283	4	4932283	76800	16	13982310
Shell sort	4040924	4932283	4	4932283	75776	20	13981290
Counting sort	4040924	4932283	4	4932283	76288	4000	13985782
Quick sort	4040924	4932283	4	4932283	76288	97468	14079250
Merge sort	4040924	4932283	4	4932283	76800	4040940	18023234
Radix sort	4040924	4932283	4	4932283	76800	4040972	18023266

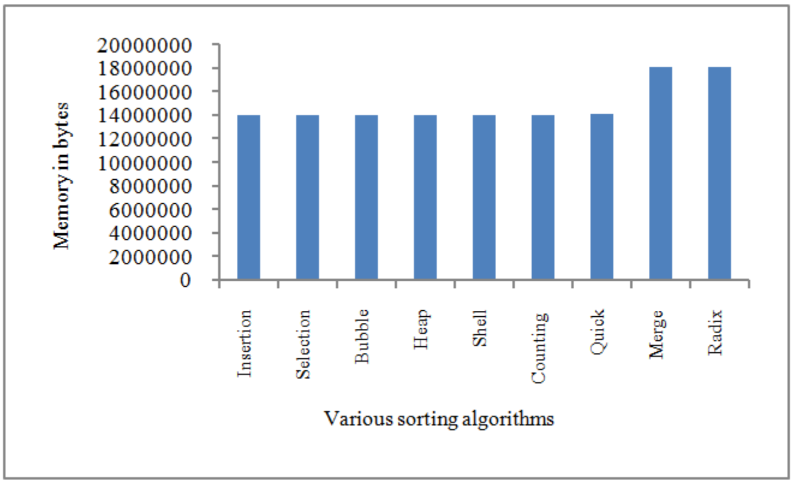


Figure 2.26: Memory occupied by various sorting algorithms

We have plotted the Figure 2.26 using Table 2.17 and it shows the total space complexity taken by the various discussed sorting algorithms in all four cases of dataset. In this figure the X-axis represents the various discussed sorting algorithms, and Y-axis represents the total memory in bytes.

By overall best case time complexity analysis, it is found that the count sort is comes out to be best sorting algorithm among other sorting algorithms in three cases of dataset, which are random, nearly sorted, reverse sorted. And the bubble sort is the best sorting algorithm when data is sorted.

And by overall worst case time complexity analysis, it is found that bubble sort is comes out to be worst sorting algorithm among other sorting algorithms when data is random, nearly sorted, reverse sorted. And the selection sort is the worst sorting algorithm when data is sorted.

By overall memory analysis, it is found that the shell sort is the best sorting algorithms in all the four cases of dataset, and radix sort is comes out to be worst sorting algorithm in all the four cases of dataset.

2.5 Conclusion

Final conclusion of this chapter is that, some existing sorting algorithms have been tested using GPU computing and compared with existing sequential sorting algorithms. The final outcome shows that more speedup is achieved by parallel sorting algorithms using GPU computing.

We have also tested the various sorting algorithms on a standard dataset. There are four cases of the dataset and every case of dataset contains the 1010228 items. We apply the various sorting algorithms in the four cases of dataset and compare the performance in each case. And also we have found out the total space complexity taken by all discussed sorting algorithms.

Chapter 3

Performance enhancement of odd-even transposition sorting network(OETSN) using GPU computing

In sorting networks comparators [70] are used to compare and exchange the data. Compare and Exchange operation is used in sorting networks [71]. There are two types of comparators available first is increasing (low to high) and the second is decreasing (high to low) comparator. Two types of comparator [72] are shown in the Figure 3.1. Odd-Even transposition sorting is an extension of bubble sort technique [73-74]. The algorithm is designed for network model and in network models comparators are used to rearrange the numbers. In odd-even transposition sorting network, increasing comparator is used to compare and exchange the data. The OETSN algorithm performs $n/2$ iteration and each iteration has two phases, first phase is the odd-even exchange and the second phase is even-odd exchange. We will understand the concept of OETSN with the help of an example.

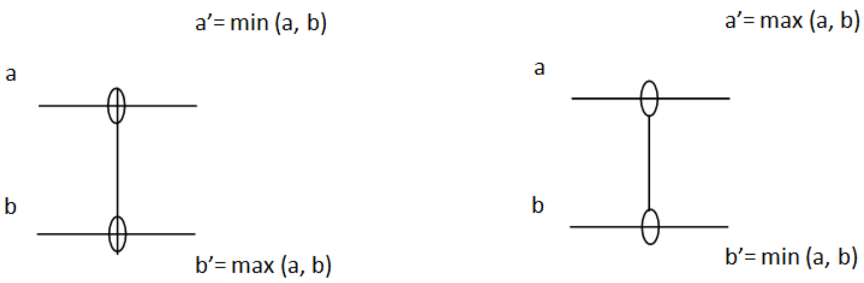


Figure 3.1: (a) Increasing Comparator (b) Decreasing Comparator.

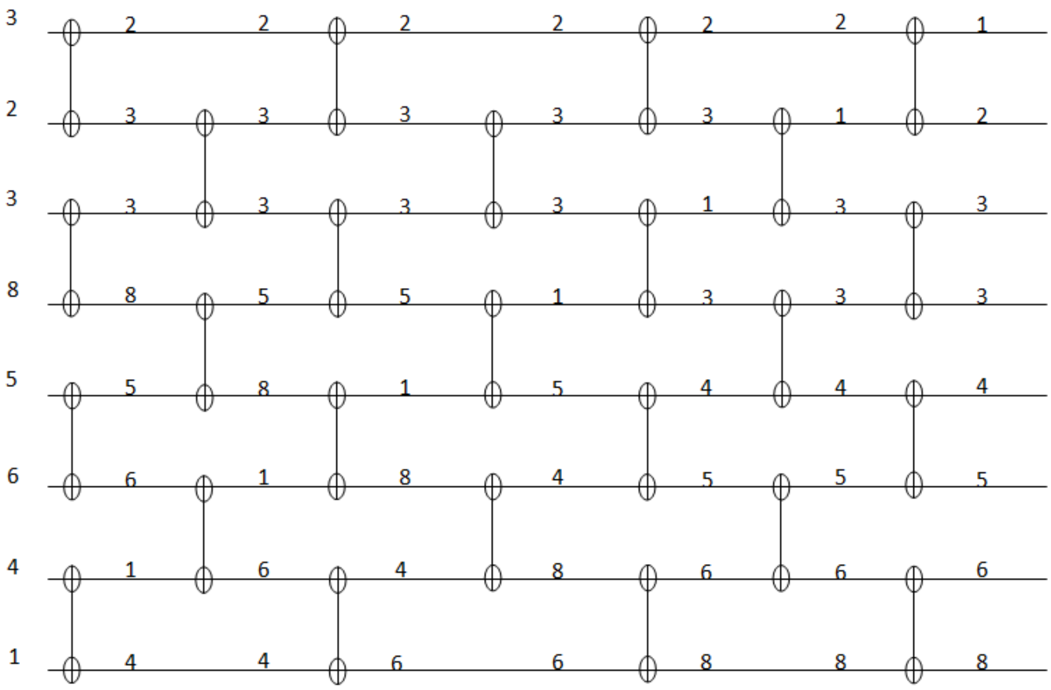


Figure 3.2: Example of OETS network

The example of OETSN is shown in Figure 3.2. After n phases of odd-even exchange, the sequence is sorted. Each phase of the algorithm either odd or even requires $O(n)$ comparisons, and there is a total of n phases; thus the sequential complexity of OETSN is $O(n^2)$.

3.1 Objective

Odd-even transposition sorting is designed for networks. In networks compare-exchange operation is used to compare the elements. We have found that the time taken for sorting by OETSN is same for all test cases such as uniform, sorted, zero, gaussian, staggered and bucket. The sequential and parallel time complexity is $O(n^2)$ and $O(n)$ respectively, of OETSN using any kind of test cases.

In our approach, we reduced the time complexity $O(n)$ to $O(1)$ over two types of test case which are sorted and zero. We have motivated from the bubble sort technique. If the data is sorted and unique, bubble sort requires only one pass and terminate the program. In our approach we have also used this technique. In this way, we have reduced the number of levels in the network and the time complexity for sorted and zero test cases.

3.2 Parallel OETSN Algorithm

It is easy to parallelize OETSN algorithm [24]. Compare-exchange operations performed simultaneously on each pair of elements. There can be two cases first case if $n = p$ where 'p' is the number of processing elements and 'n' is the number of elements to be sorted. In both the phases odd and in even phases compare-exchange operation will be performed on its right neighbour elements. This requires time $\Theta(1)$. A total of 'n' phases is performed. So the parallel run time of this formulation will be $\Theta(n)$. Second case if $p<n$ or $p>n$ then Initially, each process is assigned a block of n/p elements which it sort internally in $\Theta((n/p)(n/p))$ time. After this the processes execute 'p' phases ($p/2$ odd and $p/2$ even). During each phase $\Theta(n)$ comparisons are performed and time $\Theta(n)$ is spent in communication. We are not using any local sort before odd-even phase. Thus the parallel run time of this formulation is:

$$T_p = \Theta\left(\frac{n^2}{p^2}\right) + \Theta\left(n\right) + \Theta\left(n\right) \tag{3.2.1}$$

Since the sequential complexity of sorting is $\Theta(n^2)$, the speedup(S) and efficiency(E) of this formulation as follows:

$$S = \frac{\Theta\left(n^2\right)}{\Theta\left(\frac{p^2}{n^2}\right) + \Theta\left(n\right)} \tag{3.2.2}$$

$$E = \frac{\Theta\left(n^2\right)}{p[\Theta\left(\frac{p^2}{n^2}\right) + \Theta\left(n\right)]} \tag{3.2.3}$$

3.3 Proposed Modified Parallel OETSN Algorithm

Algorithm 11 Proposed Modified OETSN Algorithm

INPUT: Unsorted List A, Number of threads T.
OUTPUT: Sorted List A

```
for i= 1 to n/2 do
    Initialize the P array to zero for GPU
    OddPhase(A, P, n)
    EvenPhase(A, P, n)
end for
if (i==0 OR i== n/16 OR i== n/8 OR i== n/4) then
    Evalute(P)
    Read sum from GPU
end if
if sum == 0 then
    break
end if
```

The proposed sorting algorithm has been inspired from the traditional bubble sorting algorithm. In the traditional bubble sort algorithm, we compare the adjacent elements. If the elements are sorted, no swapping is done, otherwise the elements need to be swapped. Traditional bubble sort has taken 'n' passes to complete the sorting in the best case.

In the modified version of bubble sort, we have the flag variable to keep the track of swapping. If the variable highlights swapping, the next pass is executed. The same concept has been applied to the odd even transposition sorting algorithm using GPU. Here instead of using a single variable array we use two variable arrays, i.e. 'P' and 'T'. 'T' is equal to the number of threads and 'P' is the sum of total swapping performed in the proposed algorithm. Now the odd-even pass is executed. If there is no swapping then the sum of 'P' comes to be zero and we got the sorted array. This gives an added advantage for the sorted and unique test case need not to execute the code on a GPU unnecessary in the case when the data is sorted or unique. On the other hand, a slight increase in the execution time for the uniform, staggered, bucket, Gaussian test cases. This makes them unable to take the advantage of the above propose approach.

We have done same observations on $n/2$, $n/4$ and $n/8$ of the data. We have used the GPU NVDIA GeForce GTX 460 with compute capability 2.1 but the new version of GPU cards come up with the compute capability 3.0 which have got the unified memory for the GPU and CPU which can further enhance the performance of the suggested algorithm.

Future enhancements may possible to get a further speedup like we can use scan function to make the sum up faster. The functionality of the proposed algorithm is described through the flowchart shown in Figure 3.3. The green colored box shows the modules running on GPU. The proposed algorithm is more efficient in comparison with the existing techniques using two types of test case i.e. zero and sorted test cases.

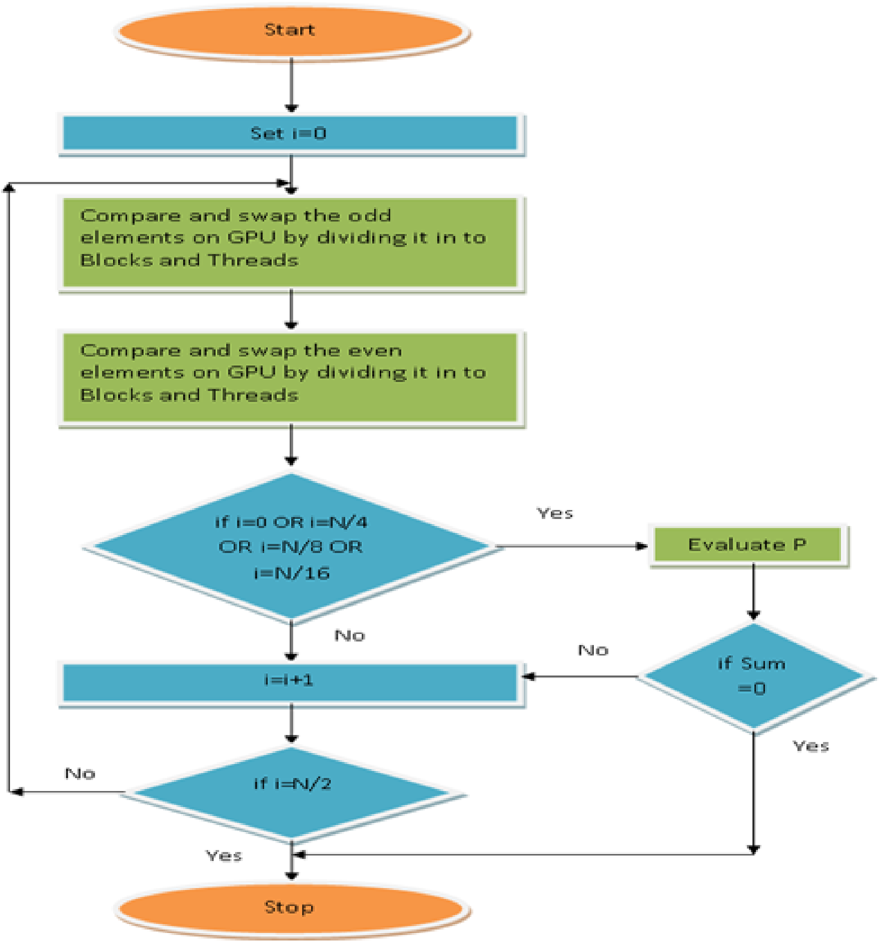


Figure 3.3: Flowchart for the proposed modified parallel OETSN

3.4 Experimental Results of Sequential and Parallel OETSN Algorithm

Sorting benchmark has been used for testing the algorithms. We have tested the sequential and parallel OETSN algorithms on six types of test cases using GPU computing having CUDA hardware. Table 3.1 shows the execution time in seconds of the sequential OETSN algorithm. The 'n' is the size of the data used for the particular cases here for the performance analysis of the algorithm. The value 'n' is varied from 1000 to 2500000. Table 3.2 shows the execution time in seconds of the parallel OETSN algorithm using different types of test cases. The size of the is denoted by 'n'. The number of threads is denoted by 'T'. The values of 'T' vary from 1 to maximum 1024. The threads increase in the power of 2.

The CUDA hardware version 2.1 has the total of 1024 threads per block so the maximum value of thread is selected as 1024. In Table 3.1, the sequential execution time shows for the six types of test cases. If we analyse the Table 3.1, zero test case has less execution time in comparison to others. It has less execution time for all the values of 'n'. After that sorted test case has less execution time in comparison to the bucket, staggered, uniform and Gaussian for all the values of 'n'. The remaining test cases have nearly equal execution time as shown in Table 3.1. It is because in the test case zero and sorted only the comparison is performed to the adjacent element and the swapping is not required in both. The comparison and swapping is performed by remaining test cases.

Next, we have evaluated the speedup achieved by parallel OETSN over the sequential OETSN. Speedup measures performance gain achieved by parallelizing a given application over sequential application. In the Table 3.1 and 3.2, we have evaluated the execution time in seconds of sequential and parallel OETSN. By equation (3.2.2) and results from Table 3.1 and 3.2, the speedup is calculated.

The speedup results described in Table 3.3. From Table 3.2 and 3.3, it can be observed that the execution time is minimum when the number of threads are 512. The speedup is increased by 8 times than the sequential code when $T = 512$. The performance of algorithm got degraded at $T = 1024$. The reason behind this is that, the data we have taken is not evenly divided over the threads. So, some of the threads are executed ideally and degrading the overall performance of the

Table 3.1: Execution time in sec of sequential OETSN using different types of test cases

n	Uniform	Gaussian	Zero	Staggered	Bucket	Sorted
1000	0.016	0.016	0.001	0.015	0.016	0.015
5000	0.062	0.062	0.015	0.078	0.063	0.031
10000	0.203	0.187	0.078	0.187	0.234	0.062
50000	4.602	4.681	0.905	4.145	5.704	0.842
100000	18.86	19.282	3.26	16.645	22.687	3.292
500000	496.988	501.091	82.681	425.274	584.469	101.713
1000000	2067.263	2050.446	400.33	1861.966	2734.954	577.812
1500000	4671.309	5135.357	912.34	4843.285	6035.218	1342.607
2000000	8095.204	7666.997	2072.224	7958.578	11156.45	4119.251
2500000	17099.89	17128.84	3719.095	16171.63	15732.54	6368.703

algorithm.

The speedup for all the six mentioned test cases is shown in Figure 3.4 to 3.9. The X-axis represents the number of threads, the Y-axis represents the speedup achieved by the parallel OETSN and the Z-axis represents the size of the dataset.

From Figure 3.4, the speedup for the uniform test case is observed. The 7 times more speedup is achieved when thread T=512 and data size n=2500000 in comparison to the sequential OETSN. We have also found that for T=1024, the speedup got decreased. It is because the data is not evenly distributed over the threads and in which some threads are ideal hence degrade the performance of the algorithm.

Table 3.2: Execution time in sec of parallel OETSN using different types of test cases

n/T	Test case	1	2	4	8	16	32	64	128	256	512	1024
1000	Uniform	0.019	0.014	0.009	0.006	0.005	0.005	0.004	0.004	0.004	0.004	0.005
	Gaussian	0.019	0.014	0.009	0.006	0.005	0.005	0.005	0.004	0.004	0.004	0.006
	Zero	0.019	0.014	0.009	0.006	0.005	0.005	0.004	0.004	0.004	0.004	0.005
	Staggered	0.019	0.014	0.009	0.007	0.005	0.004	0.004	0.004	0.003	0.003	0.005
	Bucket	0.019	0.014	0.009	0.006	0.005	0.004	0.004	0.004	0.004	0.004	0.007
	Sorted	0.019	0.014	0.009	0.006	0.006	0.005	0.005	0.005	0.004	0.004	0.005
5000	Uniform	0.441	0.322	0.173	0.096	0.056	0.035	0.026	0.023	0.023	0.023	0.025
	Gaussian	0.442	0.321	0.173	0.096	0.055	0.034	0.026	0.025	0.024	0.024	0.026
	Zero	0.441	0.321	0.167	0.091	0.051	0.031	0.021	0.021	0.021	0.021	0.022
	Staggered	0.441	0.322	0.171	0.093	0.052	0.033	0.029	0.023	0.022	0.021	0.025
	Bucket	0.442	0.322	0.169	0.092	0.052	0.033	0.025	0.023	0.023	0.023	0.025
	Sorted	0.439	0.319	0.168	0.168	0.091	0.054	0.024	0.024	0.023	0.023	0.032
10000	Uniform	1.742	1.265	0.667	0.358	0.197	0.114	0.071	0.066	0.062	0.061	0.079
	Gaussian	1.744	1.263	0.667	0.359	0.197	0.115	0.072	0.064	0.061	0.061	0.079
	Zero	1.733	1.257	0.643	0.336	0.182	0.104	0.065	0.056	0.053	0.052	0.071
	Staggered	1.744	1.271	0.657	0.345	0.188	0.108	0.067	0.06	0.058	0.058	0.074
	Bucket	1.744	1.265	0.649	0.339	0.185	0.108	0.07	0.065	0.062	0.062	0.079
	Sorted	1.733	1.256	0.643	0.335	0.182	0.104	0.065	0.057	0.054	0.054	0.072
50000	Uniform	43.438	31.472	16.462	8.602	4.499	2.410	1.353	1.094	1.090	1.080	1.235
	Gaussian	43.452	31.433	16.430	8.588	4.492	2.406	1.351	1.093	1.091	1.083	1.235
	Zero	43.328	31.338	15.931	8.020	4.089	2.160	1.204	0.943	0.925	0.919	1.078
	Staggered	43.574	31.690	16.276	8.273	4.249	2.247	1.255	0.999	0.988	0.984	1.134
	Bucket	43.573	31.535	16.081	8.092	4.127	2.228	1.240	1.110	1.090	1.080	1.231
	Sorted	43.248	31.282	15.884	7.996	4.073	2.153	1.195	0.939	0.919	0.916	1.069
100000	Uniform	213.99	130.446	70.589	36.806	19.111	9.994	5.457	4.151	4.129	4.125	4.948
	Gaussian	213.996	130.335	70.571	36.805	19.119	9.995	5.459	4.151	4.131	4.131	4.951
	Zero	213.105	129.583	69.112	34.693	17.491	9.063	4.882	3.579	3.547	3.539	4.378
	Staggered	213.938	130.885	70.171	35.605	18.077	9.373	5.069	3.786	3.757	3.751	4.564
	Bucket	213.831	130.282	69.621	34.961	17.626	9.991	4.975	3.735	3.811	3.133	4.947
	Sorted	213.189	129.580	69.117	34.681	17.491	9.053	4.877	3.578	3.543	3.535	4.366
500000	Uniform	4770.3	3349.3	1749.8	914.91	472.11	244.11	130.31	98.141	97.991	96.471	119.91
	Gaussian	4755.1	3340.3	1749.6	914.91	472.11	243.81	130.21	98.071	98.112	96.431	119.91
	Zero	4686.2	3264.3	1683.6	862.11	432.12	220.21	116.11	83.841	83.651	82.021	105.81
	Staggered	4705.3	3295.6	1705.1	878.7	438.91	228.21	120.51	88.861	88.781	87.211	110.21
	Bucket	4694.9	3287.8	1694.3	869.1	435.11	226.31	117.41	92.651	91.151	90.201	119.91
	Sorted	4686.2	3264.4	1683.6	861.9	431.81	220.12	115.91	83.781	83.591	83.096	105.81
1000000	Uniform	18833.7	13246.5	6886.4	3698.0	1799.5	921.41	488.11	359.71	359.61	358.11	476.81
	Gaussian	18805.3	13215.4	6855.2	3578.4	1799.5	922.31	488.11	359.61	359.41	358.71	476.41
	Zero	18716.1	13055.2	6755.8	3505.9	1719.1	873.71	459.21	331.31	330.91	330.92	420.71
	Staggered	18759.6	13170.4	6844.5	3556.4	1746.9	890.11	468.61	341.41	341.21	340.71	438.31
	Bucket	18746.3	13105.1	6821.3	3544.3	1724.8	884.5	461.91	334.81	332.31	331.61	476.71
	Sorted	18736.3	13095.8	6798.5	3526.7	1718.9	872.8	459.41	333.61	332.91	332.11	420.51
1500000	Uniform	60324.2	31243.2	15348.8	8155.1	4299.8	2078.1	1096.3	808.1	807.81	806.61	1071.5
	Gaussian	60297.8	31199.2	15329.7	8134.3	4255.2	2072.6	1096.3	807.91	807.81	806.31	1071.8
	Zero	60155.4	31056.2	15255.4	8005.8	4150.9	1964.2	1031.3	744.71	743.61	743.11	946.21
	Staggered	60266.4	31178.3	15299.8	8099.2	4239.3	1999.3	1052.4	766.81	766.61	765.17	985.81
	Bucket	60243.8	31141.2	15279.4	8055.3	4199.7	2070.6	1039.1	752.31	751.91	750.31	995.31
	Sorted	60196.4	31098.3	15299.4	8023.3	4162.9	1964.1	1030.9	744.21	743.51	743.31	946.21
2000000	Uniform	90655.3	46143.2	24199.3	12693.9	6678.4	3688.1	1948.4	1435.5	1435.1	1434.5	1903.5
	Gaussian	90605.4	46099.4	24210.3	12649.9	6648.9	3689.8	1948.5	1435.4	1434.7	1433.9	1902.7
	Zero	90395.3	45905.3	24065.4	12544.4	6533.5	3494.9	1833.7	1322.1	1321.2	1321.1	1681.6
	Staggered	90555.4	46055.3	24188.5	12627.9	6633.4	3556.4	1869.1	1361.7	1361.7	1360.4	1855.9
	Bucket	90498.9	45999.4	24148.8	12599.5	6598.9	3520.1	1842.6	1342.2	1340.2	1340.1	1806.9
	Sorted	90445.4	45972.8	24105.4	12555.2	6555.3	3494.1	1832.9	1321.8	1320.2	1318.7	1742.9
2500000	Uniform	165205.3	82815.4	42674.4	23139.4	12349.2	7299.8	3041.9	2241.6	2241.6	2221.1	2797.2
	Gaussian	165193.3	82793.5	42648.8	23099.8	12344.7	7291.4	3043.1	2241.4	2241.1	2241.1	2796.7
	Zero	164560.8	82555.3	42556.4	23005.3	12259.8	7233.3	2866.2	2063.6	2063.1	2060.7	2623.4
	Staggered	165149.3	82740.1	42631.9	23089.1	12316.4	7266.3	2917.1	2126.7	2126.5	2022.9	2677.5
	Bucket	165105.9	82693.3	42599.3	23049.8	12299.2	7249.4	2881.5	2084.3	2027.1	2021.6	2680.1
	Sorted	165060.8	82649.8	42574.7	23019.4	12268.5	7238.7	2862.1	2064.2	2072.1	2077.5	2622.1

Table 3.3: Speedup achieved by parallel OETSN using different types of test cases

n/T	Test case	1	2	4	8	16	32	64	128	256	512	1024
1000	Uniform	0.84	1.14	1.78	2.67	3.2	3.2	4	4	4	4	3.2
	Gaussian	0.84	1.14	1.78	2.67	3.2	3.2	3.2	4	4	4	2.67
	Zero	0.05	0.07	0.11	0.17	0.2	0.2	0.25	0.25	0.25	0.25	0.2
	Staggered	0.79	1.07	1.67	2.14	3	3.75	3.75	3.75	5	5	3
	Bucket	0.84	1.14	1.78	2.67	3.2	4	4	4	4	4	2.29
	Sorted	0.79	1.07	1.67	2.5	2.5	3	3	3	3.75	3.75	3
5000	Uniform	0.14	0.19	0.36	0.65	1.11	1.77	2.38	2.7	2.7	2.7	2.48
	Gaussian	0.14	0.19	0.36	0.65	1.13	1.82	2.38	2.48	2.58	2.58	2.38
	Zero	0.03	0.05	0.09	0.16	0.29	0.48	0.71	0.71	0.71	0.71	0.68
	Staggered	0.18	0.24	0.46	0.84	1.5	2.36	2.69	3.39	3.55	3.71	3.12
	Bucket	0.14	0.2	0.37	0.68	1.21	1.93	2.52	2.74	2.74	2.74	2.52
	Sorted	0.07	0.1	0.18	0.18	0.34	0.57	1.29	1.29	1.35	1.35	0.97
10000	Uniform	0.12	0.16	0.3	0.57	1.03	1.78	2.86	3.08	3.27	3.33	2.57
	Gaussian	0.11	0.15	0.28	0.52	0.95	1.63	2.6	2.92	3.07	3.07	2.37
	Zero	0.05	0.06	0.12	0.23	0.43	0.75	1.2	1.39	1.47	1.5	1.1
	Staggered	0.11	0.15	0.28	0.54	0.99	1.73	2.79	3.12	3.22	3.22	2.53
	Bucket	0.13	0.18	0.36	0.69	1.26	2.17	3.34	3.6	3.77	3.77	2.96
	Sorted	0.04	0.05	0.1	0.19	0.34	0.6	0.95	1.09	1.15	1.15	0.86
50000	Uniform	0.11	0.15	0.28	0.53	1.02	1.91	3.4	4.21	4.22	4.26	3.73
	Gaussian	0.11	0.15	0.28	0.55	1.04	1.95	3.46	4.28	4.29	4.32	3.79
	Zero	0.02	0.03	0.06	0.11	0.22	0.42	0.75	0.96	0.98	0.98	0.84
	Staggered	0.1	0.13	0.25	0.5	0.98	1.84	3.3	4.15	4.2	4.21	3.66
	Bucket	0.13	0.18	0.35	0.7	1.38	2.56	4.6	5.14	5.23	5.28	4.63
	Sorted	0.02	0.03	0.05	0.11	0.21	0.39	0.7	0.9	0.92	0.92	0.79
100000	Uniform	0.09	0.14	0.27	0.51	0.99	1.89	3.46	4.54	4.57	4.57	3.81
	Gaussian	0.09	0.15	0.27	0.52	1.01	1.93	3.53	4.65	4.67	4.67	3.9
	Zero	0.02	0.03	0.05	0.09	0.19	0.36	0.67	0.91	0.92	0.92	0.74
	Staggered	0.08	0.13	0.24	0.47	0.92	1.78	3.28	4.4	4.43	4.44	3.65
	Bucket	0.11	0.17	0.33	0.65	1.29	2.27	4.56	6.07	5.95	7.24	4.59
	Sorted	0.02	0.03	0.05	0.09	0.19	0.36	0.68	0.92	0.93	0.93	0.75
500000	Uniform	0.1	0.15	0.28	0.54	1.05	2.04	3.81	5.07	5.07	5.15	4.15
	Gaussian	0.11	0.15	0.29	0.55	1.06	2.06	3.85	5.11	5.11	5.2	4.18
	Zero	0.02	0.03	0.05	0.1	0.19	0.38	0.71	0.99	0.99	1.01	0.78
	Staggered	0.09	0.13	0.25	0.48	0.97	1.86	3.53	4.79	4.79	4.88	3.86
	Bucket	0.12	0.18	0.34	0.67	1.34	2.58	4.98	6.31	6.41	6.48	4.88
	Sorted	0.02	0.03	0.06	0.12	0.24	0.46	0.88	1.21	1.22	1.21	0.96
1000000	Uniform	0.11	0.16	0.3	0.56	1.15	2.24	4.24	5.75	5.75	5.77	4.34
	Gaussian	0.11	0.16	0.3	0.57	1.14	2.22	4.2	5.7	5.71	5.72	4.3
	Zero	0.02	0.03	0.06	0.11	0.23	0.46	0.87	1.21	1.21	1.21	0.95
	Staggered	0.1	0.14	0.27	0.52	1.07	2.09	3.97	5.45	5.46	5.46	4.25
	Bucket	0.15	0.21	0.4	0.77	1.59	3.09	5.92	8.17	8.23	8.25	5.74
	Sorted	0.03	0.04	0.08	0.16	0.34	0.66	1.26	1.74	1.74	1.74	1.37
1500000	Uniform	0.08	0.15	0.3	0.57	1.09	2.25	4.26	5.78	5.78	5.79	4.36
	Gaussian	0.09	0.16	0.33	0.63	1.21	2.48	4.68	6.36	6.36	6.37	4.79
	Zero	0.02	0.03	0.06	0.11	0.22	0.46	0.88	1.23	1.23	1.23	0.96
	Staggered	0.08	0.16	0.32	0.6	1.14	2.42	4.6	6.32	6.32	6.33	4.91
	Bucket	0.1	0.19	0.39	0.75	1.44	2.91	5.81	8.02	8.03	8.04	6.06
	Sorted	0.02	0.04	0.09	0.17	0.32	0.68	1.3	1.8	1.81	1.81	1.42
2000000	Uniform	0.09	0.18	0.33	0.64	1.21	2.19	4.15	5.64	5.64	5.64	4.25
	Gaussian	0.08	0.17	0.32	0.61	1.15	2.08	3.93	5.34	5.34	5.35	4.03
	Zero	0.02	0.05	0.09	0.17	0.32	0.59	1.13	1.57	1.57	1.57	1.23
	Staggered	0.09	0.17	0.33	0.63	1.2	2.24	4.26	5.84	5.84	5.85	4.29
	Bucket	0.12	0.24	0.46	0.89	1.69	3.17	6.05	8.31	8.32	8.32	6.17
	Sorted	0.05	0.09	0.17	0.33	0.63	1.18	2.25	3.12	3.12	3.12	2.36
2500000	Uniform	0.1	0.21	0.4	0.74	1.38	2.34	5.62	7.63	7.63	7.7	6.11
	Gaussian	0.1	0.21	0.4	0.74	1.39	2.35	5.63	7.64	7.64	7.64	6.12
	Zero	0.02	0.05	0.09	0.16	0.3	0.51	1.3	1.8	1.8	1.8	1.42
	Staggered	0.1	0.2	0.38	0.7	1.31	2.23	5.54	7.6	7.6	7.99	6.04
	Bucket	0.1	0.19	0.37	0.68	1.28	2.17	5.46	7.55	7.76	7.78	5.87
	Sorted	0.04	0.08	0.15	0.28	0.52	0.88	2.23	3.09	3.07	3.07	2.43

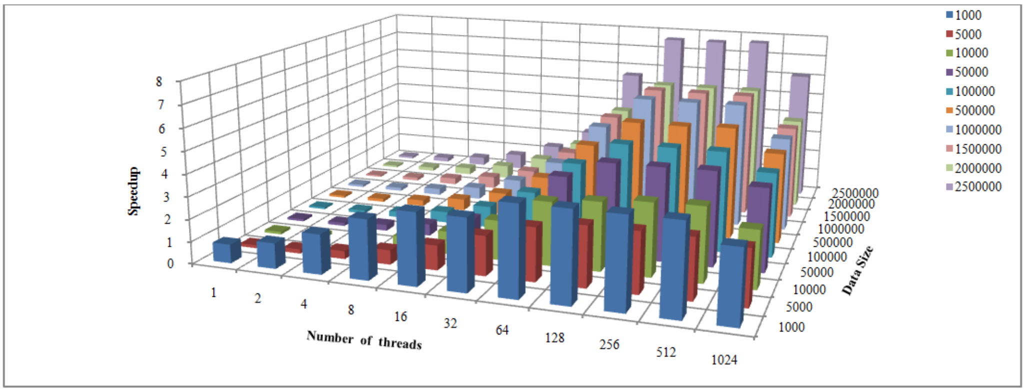


Figure 3.4: Speedup achieved by parallel OETSN using uniform test case

The Figure 3.5 shows speedup for the Gaussian test case. Here we have achieved the 7 times more speedup for the thread T=512 and data size n=2500000 in comparison to the sequential OETSN. The speedup difference can be seen at larger input or we can say that speedup is directly proportional to the number of threads and size of the input.

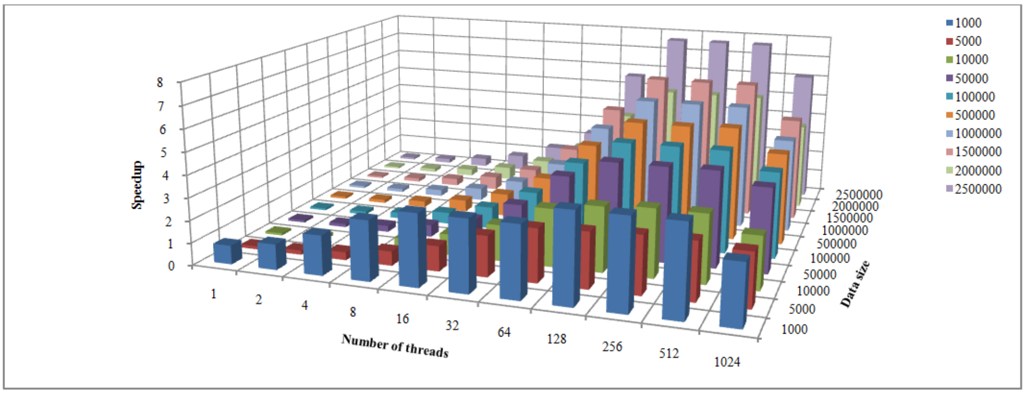


Figure 3.5: Speedup achieved by parallel OETSN using Gaussian test case

The Figure 3.6 shows the speedup for a zero test case. By analysing Figure 3.6 found that nearly 2 times speedup is achieved at T=512 & n=2500000 in comparison to the sequential OETSN. This is achieved by the zero test case.

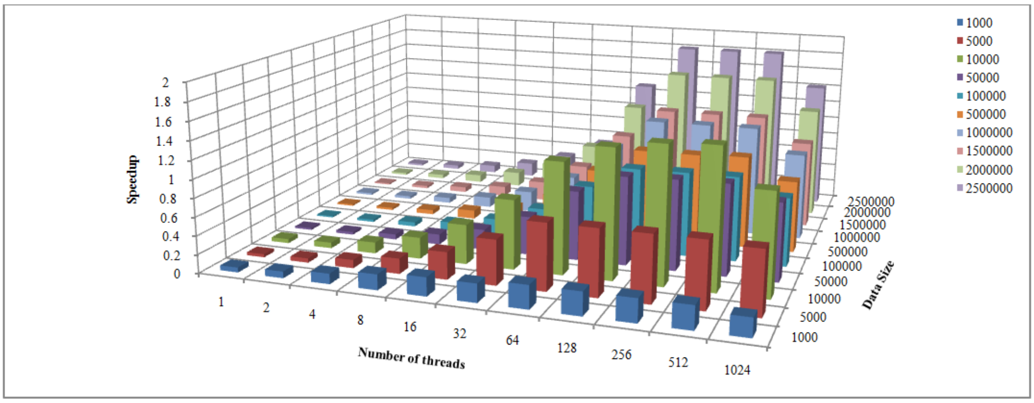


Figure 3.6: Speedup achieved by parallel OETSN using zero test case

The Figure 3.7 shows the speedup for the staggered test case. In this test case, 8 times speedup is achieved at T=512 & n=2500000 in comparison to the sequential OETSN.

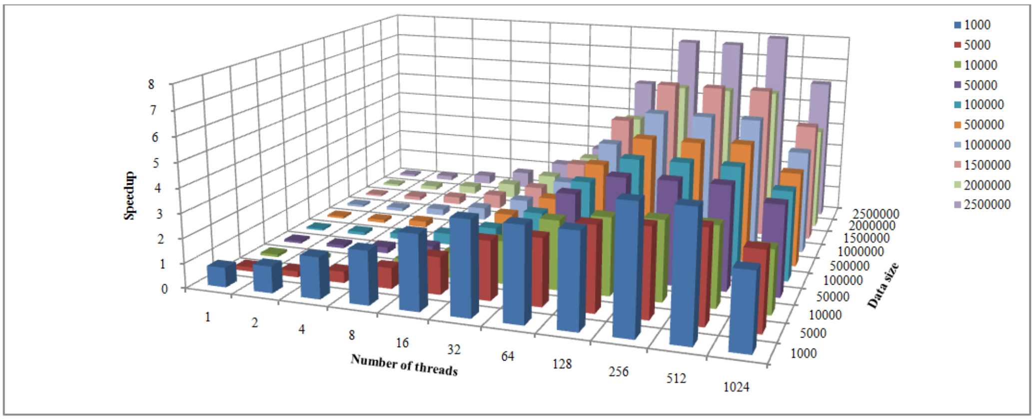


Figure 3.7: Speedup achieved by parallel OETSN using staggered test case

The Figure 3.8 shows the speedup for the bucket test case. The speedup is increased 8 times at T=512 & n=2000000 in comparison to the sequential OETSN. The speedup is achieved less at n=500 due to the reason of less amount of data.

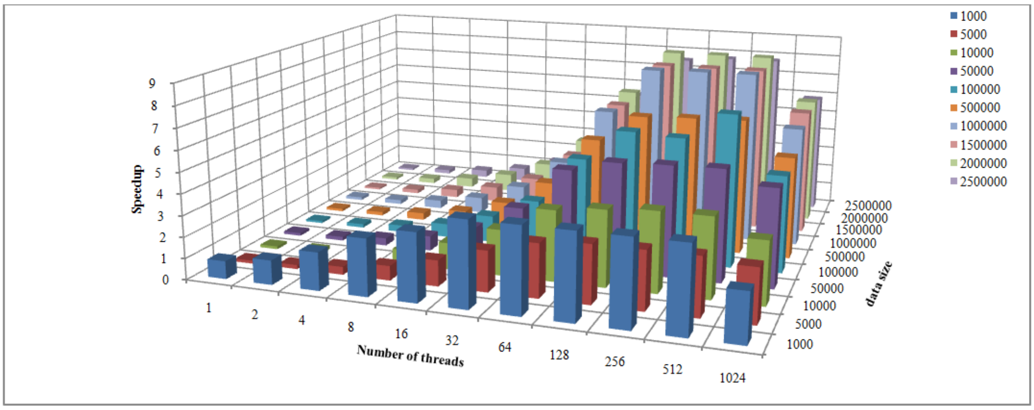


Figure 3.8: Speedup achieved by parallel OETSN using bucket test case

The Figure 3.9 shows the speedup for sorted test case. The speedup is achieved 3 times at T=512 & n=1000 in comparison to sequential OETSN. But in other test cases more speedup is achieved at n=2500000 or 2000000. It is because in the sorted test case comparison is performed to the adjacent element only. There is no swapping performed as the data is already sorted.

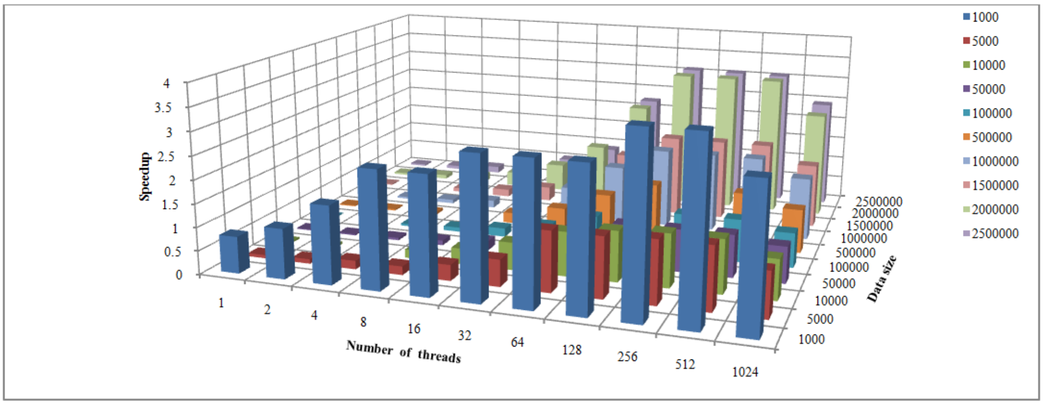


Figure 3.9: Speedup achieved by parallel OETSN using sorted test case

In conclusion, we found that speedup is directly proportional to the number of threads and size of data in most of the cases. The maximum speedup is achieved by bucket and staggered test case, i.e. 8 times in comparison to the sequential OETSN. The minimum speedup is achieved by the zero test case, i.e. 2 times. We have also found that in some cases good speedup is also achieved at

n=1000 and 5000 nearly 7 and 8 times.

3.5 Experimental Results of Proposed Modified Parallel OETSN Algorithm

Testing of proposed modified parallel OETSN algorithm has been done on the sorting benchmark using GPU computing on CUDA hardware. Table 3.4 shows the execution time in seconds of proposed modified parallel OETSN algorithm using different types of test cases. By examining the Table 3.4, we found that proposed approach is very efficient in comparison to the parallel OETSN only for zero and sorted test case.

The execution time comparison for the sorted and zero test cases of parallel and proposed modified parallel OETSN has been shown in Figure 3.10 and 3.11. The Results obtained in Table 3.4 are justified with the proposed algorithm discussed above. In the zero and sorted test case data does not require any swapping. In the odd-even module an evaluation function is being called after one pass. It is a serial function which added the number of swaps after every function has been performed.

The number of swaps is zero for the sorted and zero test case so the algorithm is terminated. Now in the case of other test cases, we do not know how the data have been placed, but still we have tried to take advantage of the proposed approach. But it has added an extra overhead on the execution time of the program of remaining test cases.

The Figure 3.10 and 3.11 has been shown with the sub-figures from (a) to (j).

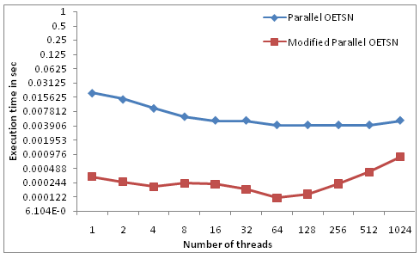
In all the sub-figures the X-axis represents the number of threads and the Y-axis represents the execution time in seconds. The execution time comparison of zero and sorted test case of parallel OETSN and proposed modified parallel OETSN is shown in Figure 3.10 and 3.11. We have analysed from Figure 3.10 and 3.11, that the execution time of the proposed modified parallel OETSN algorithm is very less as compared to the existing parallel OETSN algorithm. The scale of the Y-axis has been taken in logarithmic, using base to the power 2, because the execution time of the proposed approach is very less in comparison to the existing one.

The Figure 3.10 describes the execution time comparison of existing parallel OETSN and proposed modified parallel OETSN over zero test case. As the modified parallel OETSN is exploiting the nature of data so we are getting better results in all the cases of data size from n=1000 to 2500000. For the small data set we can see that the execution time of modified parallel OETSN is trending towards existing parallel OETSN. This is due to the fact that each treads have very few data elements to sort.

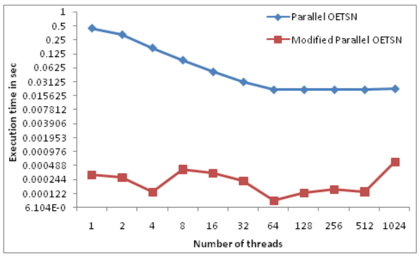
The Figure 3.11 compares the execution time comparison of parallel OETSN and modified parallel OETSN over sorted test case. The zero test case is the special case of the sorted data. There is no swapping in both the cases, thats why the trends of modified OETSN are almost similar to zero test case.

Table 3.4: Execution time in seconds of modified parallel OETSN using different types of test cases

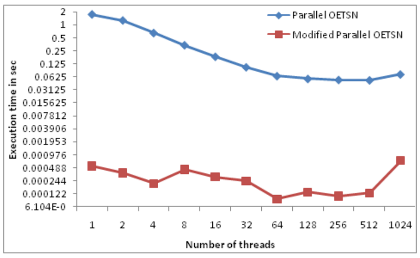
n/T	Test case	1	2	4	8	16	32	64	128	256	512	1024
1000	Uniform	0.039	0.03	0.012	0.008	0.007	0.006	0.006	0.006	0.005	0.005	0.005
	Gaussian	0.029	0.019	0.012	0.008	0.006	0.004	0.004	0.004	0.004	0.004	0.006
	Zero	0.0004	0.0003	0.0003	0.0002	0.0002	0.0002	0.0002	0.0001	0.0001	0.0001	0.0005
	Staggered	0.029	0.02	0.012	0.008	0.007	0.006	0.006	0.005	0.004	0.004	0.006
	Bucket	0.029	0.019	0.011	0.008	0.006	0.004	0.004	0.003	0.003	0.003	0.005
	Sorted	0.00017	0.00016	0.00009	0.00008	0.00007	0.00006	0.00005	0.00005	0.00004	0.00004	0.00009
5000	Uniform	0.656	0.435	0.234	0.128	0.072	0.042	0.031	0.027	0.026	0.025	0.028
	Gaussian	0.657	0.437	0.235	0.129	0.072	0.042	0.033	0.026	0.025	0.024	0.029
	Zero	0.0003	0.0003	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0006
	Staggered	0.657	0.462	0.24	0.129	0.07	0.042	0.032	0.032	0.028	0.029	0.029
	Bucket	0.656	0.43	0.227	0.122	0.069	0.041	0.033	0.026	0.025	0.025	0.029
	Sorted	0.00032	0.00025	0.00021	0.00013	0.00011	0.00008	0.00007	0.00007	0.00006	0.00006	0.00014
10000	Uniform	2.598	1.72	0.909	0.483	0.263	0.146	0.091	0.081	0.074	0.073	0.095
	Gaussian	2.597	1.718	0.909	0.484	0.263	0.147	0.091	0.082	0.075	0.074	0.096
	Zero	0.0005	0.0004	0.0002	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0007
	Staggered	2.6	1.825	0.932	0.483	0.259	0.142	0.088	0.078	0.072	0.072	0.094
	Bucket	2.6	1.695	0.879	0.457	0.245	0.139	0.09	0.082	0.075	0.074	0.095
	Sorted	0.00062	0.00037	0.00024	0.00013	0.00018	0.00009	0.00006	0.00007	0.00006	0.00006	0.00006
50000	Uniform	64.64	42.90	22.52	11.67	6.03	3.15	1.76	1.40	1.40	1.39	1.50
	Gaussian	64.66	42.91	22.52	11.66	6.03	3.15	1.76	1.41	1.40	1.40	1.50
	Zero	0.0025	0.0016	0.0009	0.0005	0.0003	0.0002	0.0001	0.0002	0.0002	0.0004	0.0007
	Staggered	64.64	45.54	23.11	11.61	5.88	3.04	1.69	1.34	1.33	1.33	1.44
	Bucket	64.64	42.25	21.81	11.00	5.58	2.89	1.62	1.40	1.40	1.40	1.50
	Sorted	0.00255	0.00167	0.00087	0.00047	0.00026	0.00015	0.00013	0.00013	0.00011	0.00011	0.00013
100000	Uniform	225.65	174.34	94.25	48.95	25.27	13.12	7.10	5.33	5.31	5.31	6.04
	Gaussian	223.12	174.21	94.24	48.96	13.12	7.11	5.34	5.34	5.32	5.31	6.04
	Zero	0.006	0.0032	0.0018	0.0009	0.0005	0.0003	0.0002	0.0002	0.0003	0.0004	0.0007
	Staggered	225.87	184.77	97.43	49.10	24.79	12.69	6.82	5.09	5.07	5.06	5.80
	Bucket	224.53	171.30	92.32	46.70	23.57	12.04	6.51	4.98	4.18	4.02	6.04
	Sorted	0.00584	0.00321	0.00178	0.00091	0.00054	0.00028	0.00017	0.00016	0.00015	0.00015	0.00024
500000	Uniform	4870.8	3550.6	1921.6	1217.1	625.3	321.5	170.3	126.2	126.1	126.0	146.3
	Gaussian	4870.2	3521.9	1911.7	1216.8	624.5	320.4	170.2	126.4	126.3	126.2	146.3
	Zero	0.0258	0.0162	0.0083	0.0044	0.0022	0.0012	0.0007	0.0006	0.0006	0.0008	0.0011
	Staggered	4770.2	3421.9	1811.7	1116.8	613.0	310.8	163.2	120.4	120.3	120.3	140.3
	Bucket	4690.2	3391.9	1791.7	1196.8	583.5	295.0	155.4	124.3	123.6	123.2	146.4
	Sorted	0.02576	0.01631	0.0084	0.00435	0.00226	0.00116	0.00072	0.00052	0.00052	0.00049	0.00057
1000000	Uniform	18999.6	13446.8	6986.6	3749.7	1999.8	1288.9	688.4	504.7	506.4	505.6	593.4
	Gaussian	18931.6	13412.7	6931.6	3721.6	1958.7	1231.6	612.6	484.3	481.6	478.6	521.3
	Zero	0.0519	0.0327	0.0169	0.0088	0.0043	0.0023	0.0013	0.001	0.0011	0.0013	0.0016
	Staggered	18998.6	13487.8	6998.5	3798.4	1998.9	1287.6	698.9	584.3	581.6	578.6	621.3
	Bucket	18811.6	13337.8	6838.5	3658.4	1985.8	1197.6	658.8	524.4	511.4	508.6	611.3
	Sorted	0.05187	0.03265	0.01701	0.0088	0.00432	0.00222	0.00132	0.00106	0.00101	0.00096	0.00113
1500000	Uniform	60576.7	31467.7	15587.6	8368.0	4599.6	2251.7	1264.8	1156.6	1145.9	1142.0	1333.6
	Gaussian	60521.7	31421.7	15531.9	8315.7	4523.6	2121.7	1212.6	1115.7	1106.7	1101.7	1312.7
	Zero	0.0761	0.049	0.0252	0.0132	0.0075	0.0033	0.0018	0.0015	0.0015	0.0016	0.0022
	Staggered	60621.7	31496.6	15588.0	8393.9	4589.9	2179.7	1289.8	1198.8	1188.7	1179.7	1389.7
	Bucket	60511.7	31336.6	15428.0	8283.9	4679.6	2119.7	1199.9	1088.8	1078.7	1069.7	1319.7
	Sorted	0.07624	0.049	0.02521	0.01322	0.00751	0.00338	0.00175	0.00138	0.00137	0.0013	0.00162
2000000	Uniform	90841.8	46324.8	24343.8	12843.7	6834.9	3873.7	2052.7	1665.9	1645.9	1611.7	2012.6
	Gaussian	90759.3	46289.6	24289.6	12789.5	6779.6	3812.6	2012.6	1612.7	1601.7	1589.6	1989.6
	Zero	0.1021	0.0652	0.0336	0.0176	0.009	0.0044	0.0023	0.0019	0.002	0.0021	0.0026
	Staggered	90859.3	46389.9	24389.5	12889.2	6879.6	3899.9	2079.9	1612.7	1609.7	1604.6	1999.6
	Bucket	90710.3	46124.9	24249.5	12779.2	6789.6	3789.9	2010.8	1582.7	1579.7	1564.6	1919.6
	Sorted	0.10196	0.06515	0.03359	0.01761	0.00896	0.00436	0.00242	0.00181	0.00179	0.0017	0.0021
2500000	Uniform	167205.5	83211.7	42817.7	23834.8	12887.7	7934.9	3476.5	2483.7	2454.9	2444.7	2984.9
	Gaussian	165803.7	83204.8	42253.6	23765.6	12754.6	7911.6	3432.7	2426.5	2423.5	2422.5	2932.3
	Zero	0.1281	0.0813	0.042	0.0221	0.0119	0.0069	0.0025	0.0019	0.0018	0.0017	0.0029
	Staggered	165898.7	83298.8	42353.9	23865.2	12854.6	7997.6	3489.9	2432.5	2424.5	2422.5	2989.3
	Bucket	165721.7	83198.8	42213.9	23745.2	12744.6	7867.7	3399.7	2329.5	2324.5	2322.5	2929.3
	Sorted	0.12802	0.0816	0.04196	0.0221	0.01186	0.00687	0.00286	0.00233	0.00228	0.0022	0.00267



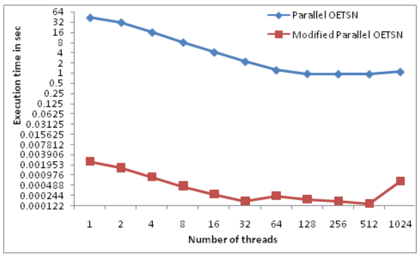
(a) n=1000



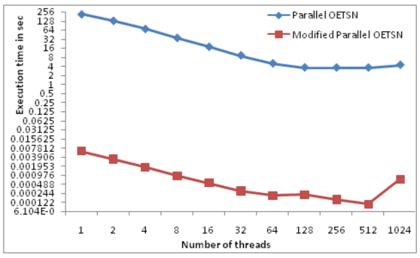
(b) n=5000



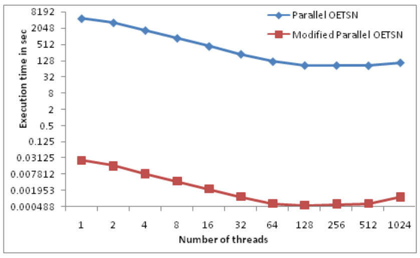
(c) n=10000



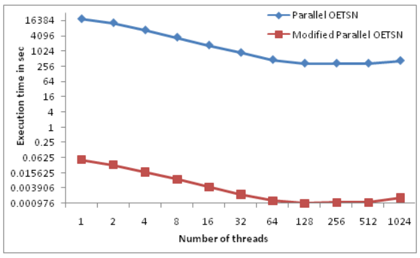
(d) n=50000



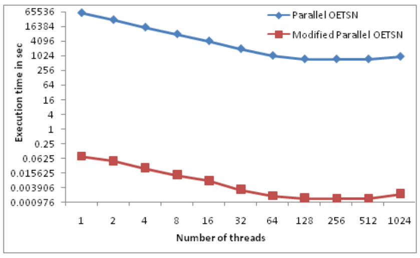
(e) n=100000



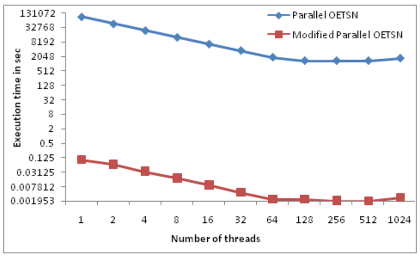
(f) n=500000



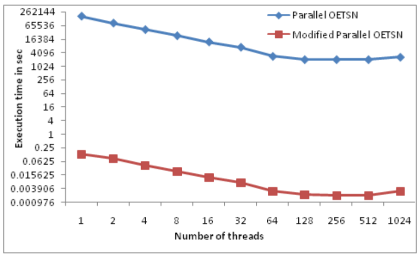
(g) n=1000000



(h) n=1500000

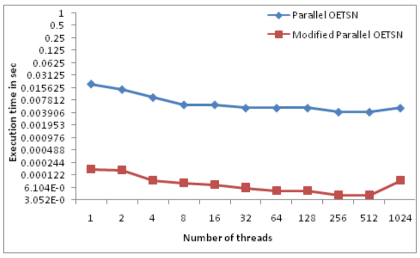


(i) n=2000000

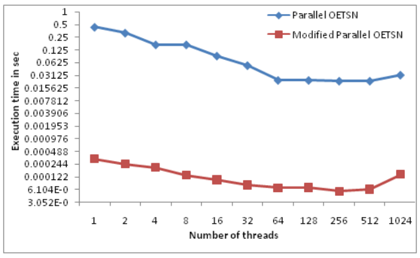


(j) n=2500000

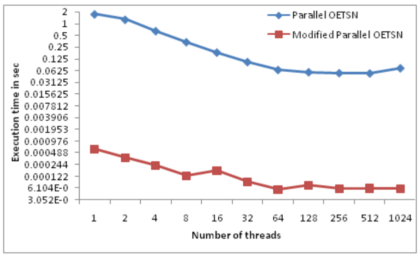
Fig. 3.10. Execution time comparison of parallel and modified OESTN using the zero test case



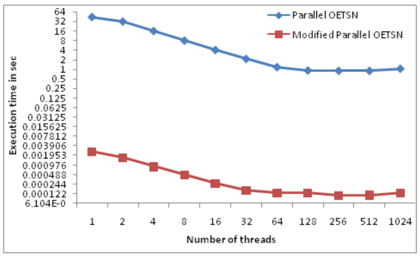
(a) n=1000



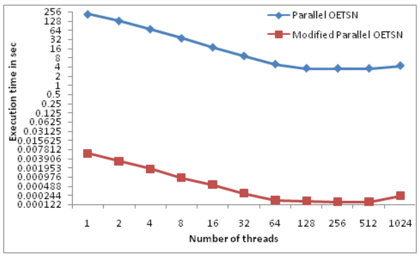
(b) n=5000



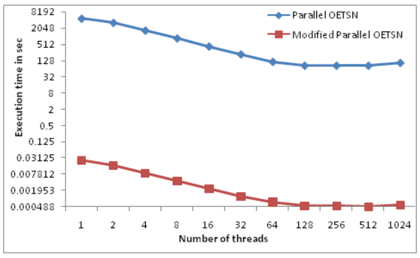
(c) n=10000



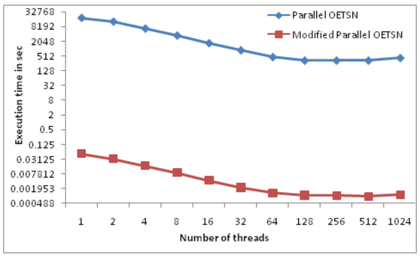
(d) n=50000



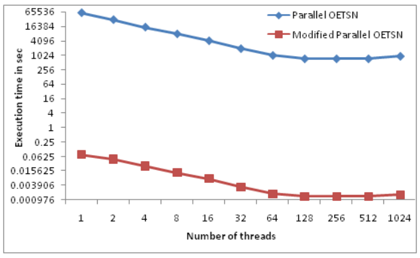
(e) n=100000



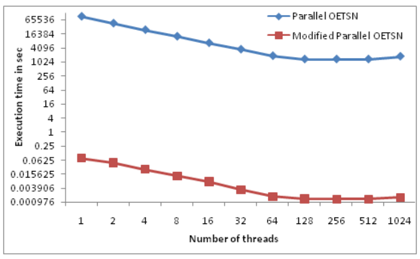
(f) n=500000



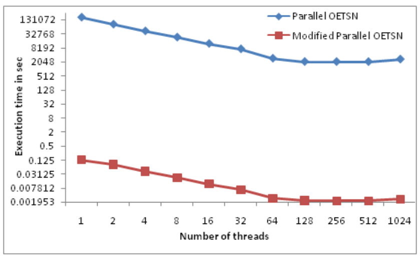
(g) n=1000000



(h) n=1500000



(i) n=2000000



(j) n=2500000

Fig. 3.11. Execution time comparison of parallel and modified OESTN using the sorted test case

3.6 Conclusion

The overall conclusion of this chapter is described that, odd-even transposition sorting is having the sequential time complexity $O(n^2)$. So, we have parallelized the OETSN using GPU computing, on CUDA hardware. After this we have proposed the modified parallel OETSN using GPU computing itself. In the proposed modified parallel OETSN, we have reduced the number of levels of the network. After testing, we found that number of levels and time complexity from $O(n)$ to $O(1)$ of the OETSN has been reduced for two types of test cases i.e. zero and sorted test cases.

Chapter 4

Performance Enhancement of Library Sort Algorithm with Uniform Gap Distribution

Many authors have invented many sorting algorithms [41][62], among them insertion sort is the one of the simplest algorithm used for sorting [42]. Insertion sort [43] [75] is less efficient on large number of items as it takes $O(n^2)$ time in worst case [44] [76], and the best case of insertion sorting occurs when data is in sorted manner and it is $O(n)$ in best case. Insertion sort is stable sorting algorithm [45]. The improvement to the insertion sort algorithm was invented by D.L Shell and the modified version is called shell sort [46]. Shell sort [47] is more efficient for large items. Library sort is an adaptive sorting [49] and also stable sorting algorithm [50]. If we leave more space, the fewer elements we move on insertion. The author achieves the $O(\log n)$ insertions with high probability using the evenly distributed gap, and the algorithm runs $O(n \log n)$ with high probability. $O(n \log n)$ is better than $O(n^2)$. The idea of leaving gaps for insertions in a data structure is used by Itai, Konheim, and Rodeh [52]. This idea has found recent application in external memory and cache-oblivious algorithms in the packed memory structure of Bender, Demaine and Farach-Colton and later used in [53-54-55].

4.1 Objective

Library sort has better run time than insertion sort, but the library sort also has some issues.

The first issue is the value of gap which is denoted by 'ε', the range of gap is given, but it is yet to be implemented to check that given range is satisfying the concept of library sort algorithm.

The second issue is re-balancing which accounts the cost and time of library sort algorithm.

The third issue is that, only a theoretical concept of library sort is given, but the concept is not implemented. So, to overcome these issues of library sort, in this section, we have implemented the concept of library sort and done the detailed experimental analysis of library sort algorithm, and measure the performance of library sort algorithm on a dataset.

4.2 Library Sort Algorithm

Library sort [48] is the formulation of insertion sort algorithm. The author has given the theoretical concept about library sort. Author has given the gaps after each insertion in the array and gaps denoted by the epsilon but he has not given the value of epsilon. He used the re-balancing concept and he re-balanced the array after inserting the 2^i elements in the array, whether re-balancing is necessary in the array, but it is also amounts cost and time. So we have to decide that is re-balancing required after inserting the 2^i elements in the array. These are the some questions of library sort, So in this paper, we are going to overcome the questions of library sort algorithm. The algorithm of library sort is as follows: Algorithm of Library Sort: There are three steps of the algorithm.

1. Binary Search with blanks

2. Insertion

3. Re-balancing

1. Binary Search with blanks: In library sort we have to search a number and the best search for an array is found by binary search. The array 'S' is sorted but has gap. As in computer, gaps of memory will hold some value and this value is fixed to sentential value that is '-1'. Due to this reason we cannot directly use the binary search for sorting. So we have modified the binary search. After finding the mid, if it comes out to be '-1' then we move linearly left and right until we get a non zero value. These values are named as $m1$ and $m2$. Based on these values we define new low, high and mid for the working. Another difference of the binary search presented below is that it not only searches the element in the list but also reports the correct position where we have to insert the number.

2. Insertion: As we know, library sort is also known by the name 'gapped insertion sort'. If the value to be inserted is in the gap, then it is ok but if there is an element in that particular position, we have to shift the elements till we find the next gap.

Algorithm 13 Library Sort: Insertion

INPUT: Data to be sorted n and pass number i

OUTPUT: Sorted list but without gaps

```
if i = = 1 then
    i1 = i - 1
    c1 = 0
end if
S1 = pow(2,i)
if S1 > size then
    S1 = size
end if
for j = (pow(2,i1-1) - c1) to S1 do
    k = search(pow(2,i)+pow(2,i+1),a[j])
    if S[k] != -1 then
        managetill(k)
    end if
    S[k] = a[j]
end for
```

Algorithm 12 Library Sort: Binary-Search with Blanks

INPUT: Data to be sorted n and Number to be searched k

OUTPUT: Position to enter the element d

```
while (low < high) do
    mid = (low + high)/2
    if (S[mid] = = -1) then
        m1 = m2 = mid
        if (m1 == 0 and m2 >= high+1) then
            if (k < S[m1]) then
                low = high = m1
            else
                low = high = m1+1
            end if
        end if
        if (m1 > 0 and m2 < high+1) then
            if (k <= S[m1]) then
                if (k == S[m]) then
                    low = high = m1
                else
                    high = m1-1
                end if
                if (k > S[m1] and k < S[m2]) then
                    low = m1 + 1
                    high = m2 - 1
                end if
                if (m2 < high) then
                    low = m2 + 1
                else
                    low = m2
                end if
            end if
            if (m1 == 0 and m2 <= high) then
                if (k >= S[m2]) then
                    if (m2 <= high) then
                        low = m2 + 1
                    else
                        low = m2
                    end if
                end if
            end if
        end if
    else
        if (S[mid] < k) then
            low = mid + 1
        end if
    end if
end while
```

3. Re-balancing: Re-balancing is done after inserting 2^i elements. This increases the size of the array. The increase in the size of array will depend on ε (number of spaces to be inserted). To do this process we will require an auxiliary array of same size so as to make a duplicate copy with a gap.

Algorithm 14 Library Sort:Re-balancing

INPUT: Sorted data but not uniformly gapped and re-balancing factor e.

OUTPUT: Sorted list of n items

```
while l < n do
    if S[j] ! = -1 then
        reba[i] = S[j]
        i++
        j++
        l++
        for k=0 to e do
            reba[i] = -1
            i++
        end for
    else
        j++
    end if
    for k = 0 to i do
        S[k] = reba[k]
    end for
end while
```

4.3 Execution time based testing of library sort algorithm

We have tested the library sort algorithm on a standard dataset [T10I4D100K (.gz)] [31] by increasing the value of gap (ε) . The dataset contains the 1010228 items. We have tested on four cases. Table 4.1 shows the execution time in microseconds of library sort algorithm using the standard dataset. By analyzing the Table 4.1, we can see that when we increase the gap value between the elements the execution time will decrease. By analyzing the Table 4.1, we can see that when we increase the gap value between the elements the execution time will decrease. The following figures show this effect. In all the figures X-

axis represents the increasing value of the gap and the Y-axis shows the time in microseconds.

Table 4.1: Execution Time of Library Sort Algorithm in Microseconds Based on Gap Values

Epsilon	Random	Nearly Sorted	Reverse Sorted	Sorted
$\varepsilon = 1$	981267433	864558882	1450636163	861929937
$\varepsilon = 2$	729981576	620115904	1065247938	609647355
$\varepsilon = 3$	119727535	358670053	278810310	356489846
$\varepsilon = 4$	23003046	117188830	263693774	116590140

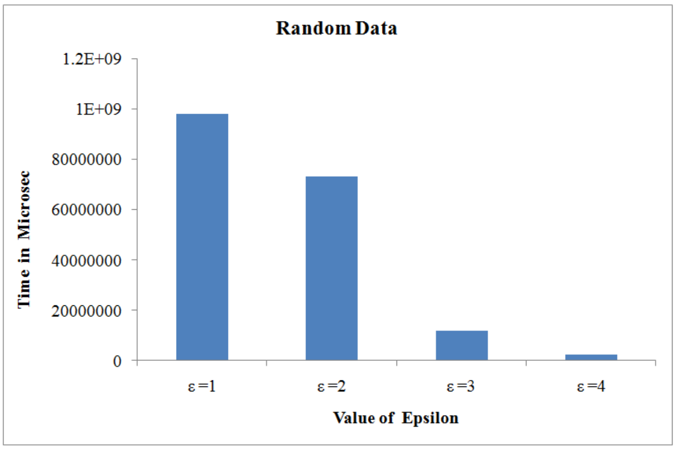


Figure 4.1: Execution time of random data using value of gaps

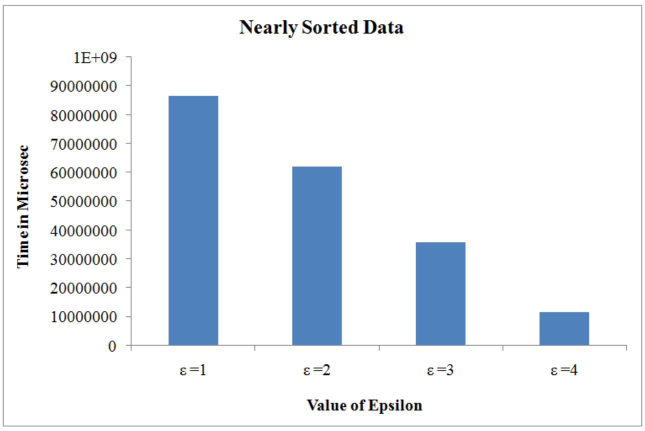


Figure 4.2: Execution time of nearly sorted data using value of gaps

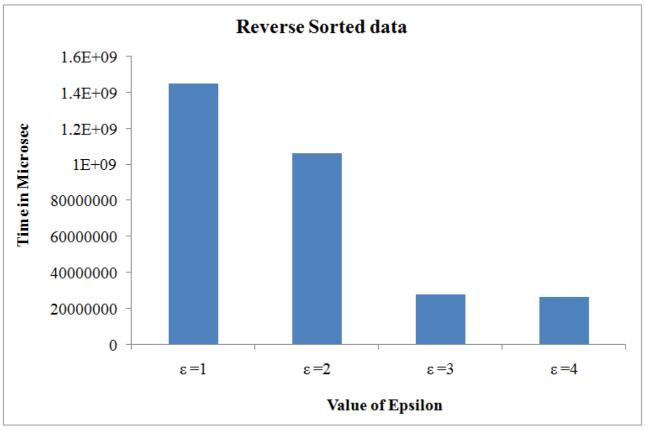


Figure 4.3: Execution time of reverse sorted data using value of gaps

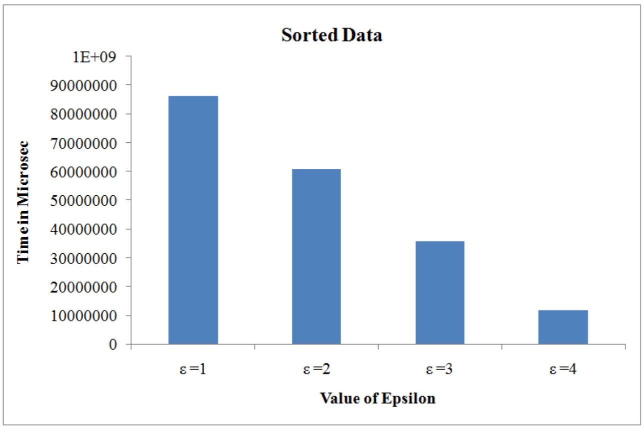


Figure 4.4: Execution time of sorted data using value of gaps

We have plotted Figure 4.1 to 4.4 by using Table 4.1. By examining these figures, we can see that how the execution time is decreasing when the gap value between items is increasing. In Figure 4.1 to 4.4, we are representing the execution time in microseconds in all the four cases of dataset.

The value of epsilon: when we increase the value of epsilon, the execution time will decrease, but at some point, value of epsilon gets saturated point because we are allocating more gaps, but these gaps are more than are actually required for the operation, so it will only be an extra memory overhead because we need more memory to store the elements. So in this way the space complexity of the algorithm increases linearly, when we increase the value of epsilon. The concept of space complexity will be explained in the next section with the help of graph.

4.4 Memory based Testing of Library Sort

Auxiliary space complexity of library sort is $O(n)$, but the space complexity is not only limited to auxiliary space. It is the total space taken by the program which includes the following.

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)

1) M_{ip}**:** For M_{ip}, we have to allocate memory of four bytes for each variable (element). As we are having total of 1010228 elements, so it will consume 1010228 $\times$ 4 = 4040912 bytes, Again, to input these items in an array we will have an index variable 'a' will of four bytes and 4 bytes for file pointer so it will be total of 4040912 bytes + 4 bytes of file pointer = 4040916 bytes, and 16 bytes are used for variable declared in the program so total space complexity taken by the M_{ip} = 4040932 bytes.

2) M_{is}**:** We will get this input as storage file in secondary storage, but in file we store this data in a stream of bytes in character. For this, it will have slightly larger memory in comparison to primary memory.

3) M_{op}**:** As we get the result either in input variable or in temporary variable, it will not have requirement for storage on primary memory, but as we have to write this data on to secondary storage it will require file pointer of 4 bytes.

4) M_{os}**:** As we get the result in a temporary variable i.e. in Borland C++, the output stored in str file and the size of M_{os} will be the size of str file and it will be same for all four cases of dataset, because we are using the 1010228 elements for all the cases of dataset.

5) M_c**:** To calculate this space, we have to find the size of .exe files created in windows for the discussed library sort program, as this program will be stored in main memory for their execution. The size of the .exe file depends on the sorting algorithms.

6) M_w**:** The space complexity of M_w of an algorithm depends on the variable declared for the allocation. In our case we divided the memory in various parts each having it own variables for specific functions.
Library Module: It is having its own three variable consuming up $4 \times 3 = 12$ Bytes of memory. In this function we have two functions called insertion and re-balancing. The insertion module requires the maximum space $(1 + \varepsilon)\times$ n $\times$ 4 bytes of memory to store the sorted data and temporary data during the processing and 20 bytes for temporary variables. This module itself has two prime modules: search and managetill, this modules consume the 20 bytes and 12 bytes of memory. The re-balancing is also require extra space for adding the space in the array which will again equal to $(1 + \varepsilon) \times n \times 4$ bytes of memory and 24 bytes of memory required for temporary variables. So the total memory will be equal to $= 2 \times ((1+ \varepsilon) \times$ n $\times 4 \times 4 + 12 + 4 + 20 + 20 + 4 + 12 + 4 + 24 + 4)$. The details of these values have been described in the Table 4.2

In Table 4.2, we have seen the total space complexity taken by the library sort using the dataset. From Table 4.2, we can see that there is no effect of re-balancing factor, but there is an effect of epsilon values. When we increase the gap value, the space taken by the program will also increase. We can see this effect with the help of graph shown in Figure 4.5. In Figure 4.5, the X-axis represents the value of epsilon and the Y-axis represents the memory occupied by the library sort algorithm in bytes. We can see that space complexity of the library sort algorithm increases linearly, when we increase the value of epsilon or gaps between the elements. It increases because we require more memory to store the elements and it is directly proportional to the value of epsilon. Due to this fact, the memory required is directly proportional to the value of epsilon, where epsilon is $(1 + \varepsilon)n$.

Table 4.2: Total Memory in Bytes of Library Sort with Increasing Value of Gaps and Re-balancing Factor

$Re-balancing$	$Value of \varepsilon$	M_{ip}	M_{os}	M_{op}	M_{os}	M_c	M_w	$Total$
2	1	4040932	4932283	4	4932283	81920	16163752	30151174
	2	4040932	4932283	4	4932283	81920	24245576	38232998
	3	4040932	4932283	4	4932283	81920	32327400	46314822
	4	4040932	4932283	4	4932283	81920	40409224	54396646
3	1	4040932	4932283	4	4932283	81920	16163752	30151174
	2	4040932	4932283	4	4932283	81920	24245576	38232998
	3	4040932	4932283	4	4932283	81920	32327400	46314822
	4	4040932	4932283	4	4932283	81920	40409224	54396646
4	1	4040932	4932283	4	4932283	81920	16163752	30151174
	2	4040932	4932283	4	4932283	81920	24245576	38232998
	3	4040932	4932283	4	4932283	81920	32327400	46314822
	4	4040932	4932283	4	4932283	81920	40409224	54396646

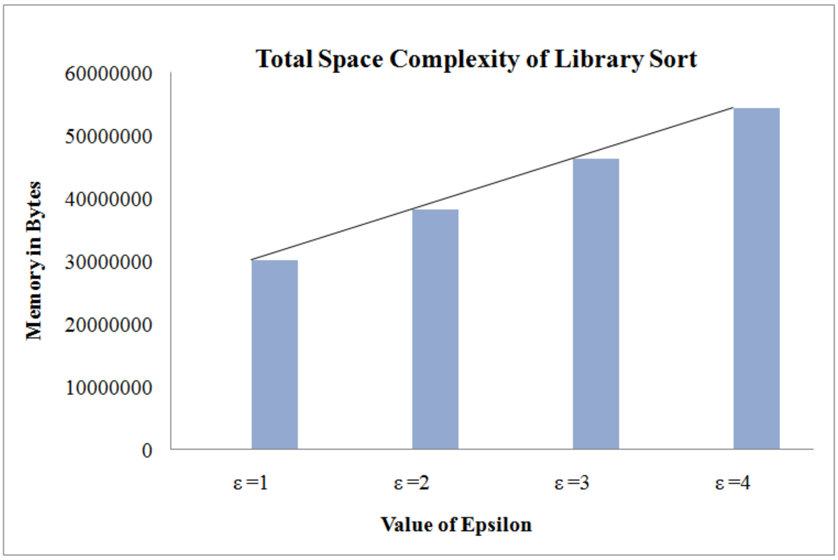


Figure 4.5: Memory occupied by library sort

4.5 Re-balancing based Testing of Library Sort

As the re-balancing is done after inserting a^i elements, this increases the size of array. The size of array will depend on ε (number of spaces to be inserted). To do this process we will require an auxiliary array of the same size so as to make a duplicate copy with gap. Whether re-balancing is necessary aftera^i elements, but it also amounts the cost and time of library sort algorithm and what will be the suitable value for 'a' is the question. We have calculated re-balancing till a^4 where a = 2,3,4 values with the value of gaps ε = 1,2,3,4. We have found that when we increase the re-balancing factor 'a' from 2 to 4 then the execution time of library sort algorithm will also increase. We can see this effect with the help of Table 4.3 and graphs described in Figure 4.6 to Figure 4.9.

Table 4.3: Time taken by Library Sort Algorithm in Microseconds during Re-balancing

$Re-balancing$	$Value of \varepsilon$	$Random$	$NearlySorted$	$ReverseSorted$	$Sorted$
2	1	981267433	864558882	1450636163	861929937
	2	729981576	620115904	1065247938	609647355
	3	119727535	358670053	278810310	356489846
	4	23003046	117188830	263693774	116590140
3	1	2622591059	2214715182	2832112301	3011802732
	2	2103580421	1964645906	2585747568	2651992181
	3	2043974421	1728175857	2195021514	1962122927
	4	1620914312	1600879365	2130261056	1620374625
4	1	2942693856	2467933298	3239333534	3281368964
	2	2705332601	2510103530	3154811065	2923182920
	3	2676681610	2613423098	3013676930	2378347887
	4	2611656774	2157740458	2993363707	2222906193

From Table 4.3, we can see that execution time of library sort is increasing when the re-balancing factor will increase in all the cases of dataset. The following graph is showing this effect.

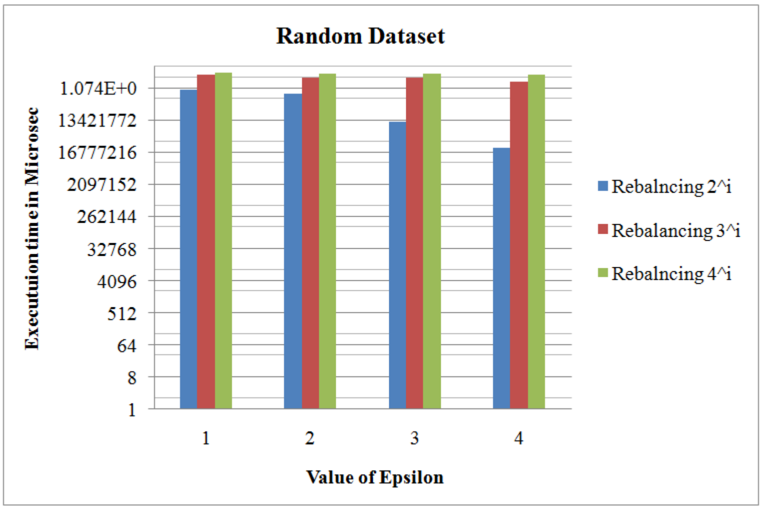


Figure 4.6: Re-balancing of library sort using random dataset

From Figure 4.6, we can see that execution time of library sort is increasing when the re-balancing factor is increasing using the random dataset.

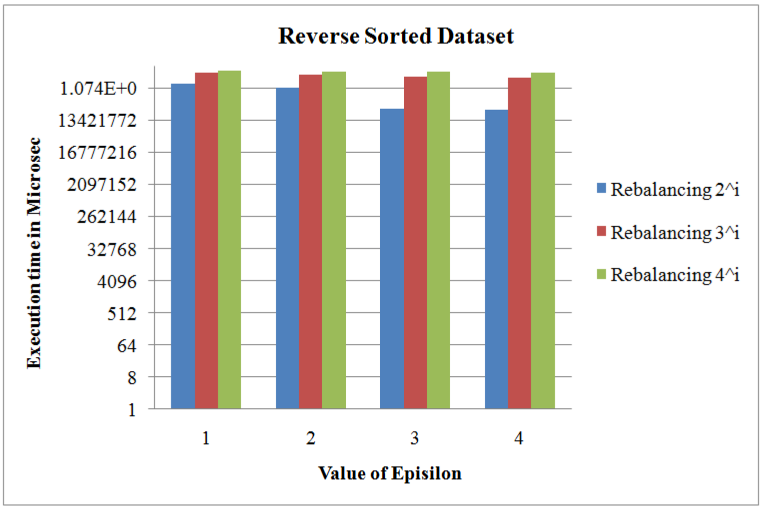


Figure 4.7: Re-balancing of library sort using reverse sorted dataset

From Figure 4.7, we can see that execution time of library sort is increasing when the re-balancing factor is increasing using the nearly sorted dataset.

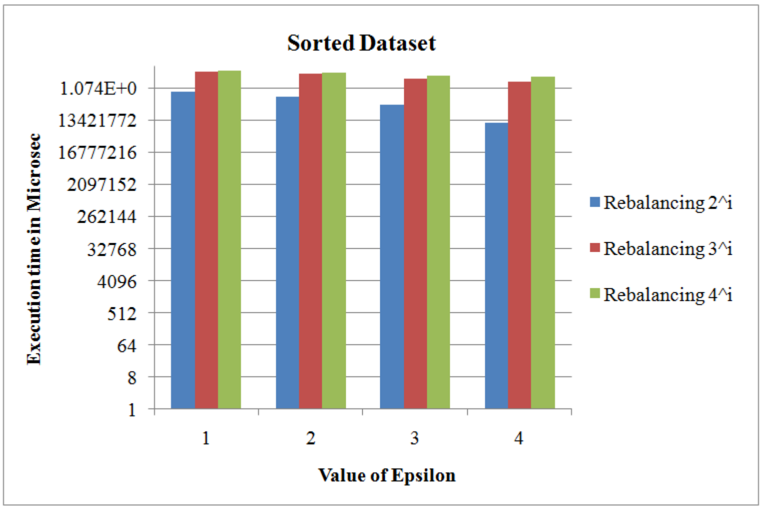


Figure 4.8: Re-balancing of library sort using sorted dataset

From Figure 4.8, we can see that execution time of library sort is increasing, when the re-balancing factor is increasing using the reverse sorted dataset.

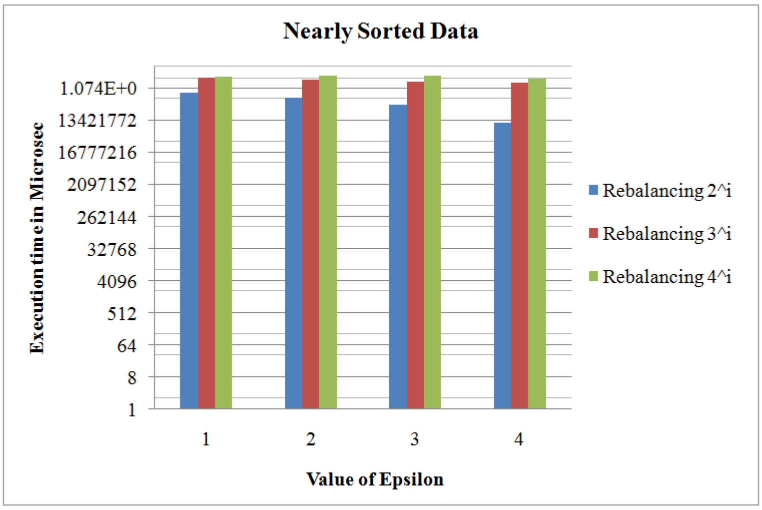


Figure 4.9: Re-balancing of library sort using nearly sorted dataset

From Figure 4.9, we can see that execution time of library sort is increasing when the re-balancing factor is increasing using the sorted dataset.

From Figure 4.6 to Figure 4.9, X-axis represents the value of epsilon and Y-

axis represents the execution time in microseconds when the re-balancing factor value is $2^i, 3^i, 4^i$. By analyzing the figures, we can see that the nature of data has marginally effected on the re-balancing factor. If the re-balancing factor is 2^i i.e we have to re-balanced the elements in the following manner $2^0, 2^1, ...2^n$. Then, the performance of algorithm is good because in the array, proper space is there to insert the new elements. But the performance of algorithm is degraded if the re-balancing factor increases from 2^i to 4^i because if we use the re-balancing factor 3^i i.e we have to re-balance the elements in the following manner $3^0, 3^1, ...3^n$. Then, in the array there is no proper space to insert the new elements in the manner of 3^i and 4^i. So shifting of data is required to insert the new elements and the spaces between many elements have been already consumed so in this way performance degrades have a larger number of swapping to generate the spaces which is same as that in the case of traditional insertion sort.

4.6 Conclusion

By execution time analysis, we have found that as we increase the value of epsilon then the execution time will decrease but at some point the value of epsilon get to a saturated point because we will have the extra spaces for the data to be inserted in between.

By space complexity analysis, we have found that space complexity of the library sort algorithm increases linearly. That is, when we increase the value of epsilon the memory consumption is also increases in the same proportion.

By execution time analysis of re-balancing, we have found that when we increase the re-balancing factor 'a' from 2 to 4 then the execution time of library sort algorithm will also increase as it moves towards traditional insertion sort. So, to find out the better result of library sort algorithm, the value of epsilon should be optimal and re-balancing factor should be minimum or ideally equal to 2.

Chapter 5

Performance Enhancement of Library Sort Algorithm with Non-Uniform Gap Distribution(LNGD)

Library sort, or gapped insertion sort is a sorting algorithm that uses an insertion sort, but with gaps in the array to accelerate subsequent insertions.

5.1 Objective

Bender *et al* has suggested the library sort algorithm with uniform gap distribution. But what happens if we have many elements that belongs to the same place in the array and there is only one gap after that element. So to overcome this problem, we have proposed the library sort with non-uniform gap distribution (LNGD).

The proposed algorithm is considered the concept of mean and median. In the proposed technique, non-uniform gap is given based on the property of insertion sort. This property tells that more updates should be done in the beginning of an array for generating more gaps. LNGD algorithm consists three steps, first two steps are same as LUGD but the third step is different.The LNGD algorithm consists of three steps. The first two steps will be the same as the LUGD algorithm [77], but the third step will be different.

Step1. Binary Search with blanks: Lets see the working of step 1 with the help of example.

Example:

1	**-1**	**3**	**-1**	**5**	**-1**	**7**	**-1**	**9**	**-1**

In the following array '-1' shows the gaps in the array. The array position is start from 0 up to 9. Now let search an element say 5.

low = 0

high = 9

mid = (0+9)/2 = 4 = $S[4]$

here $S[4]$ = 5 we got the element and terminate the search.

1	**-1**	**3**	**-1**	**-1**	**-1**	**7**	**-1**	**9**	**-1**

In this array, we do not have element 5 but we are going to search it.

Here also low = 0

High = 9

Mid == (0+9)/2 = 4 = $S[4]$

$S[4]$ = $S[\text{mid}]$ = -1

In this case, we have to find $m1$ and $m2$ as a mid which are represented by $S[m1]$ and $S[m2]$ greater than '-1' in both the direction limiting to low and high respectively. Here the value of $m1 = S[2] = 3$ and the value of $m2 = S[6] = 7$. According to $m1$ and $m2$ values, we update the low and high to perform binary search.

Step2. Insertion: Let's see the working of step 2 with the help of example.
Example:
We insert the elements in the manner of 2^i in the array. i.e in the power of 2 . This is stored in $S[i]$.
$S[i]$ = pow(2, i) where 'i' is the pass number i.e i = 0, 1, 2, 3... if $i = 0$ then $S1 = 2^0 = 1$. Now we search the position for the insertion element in the array and add the element at position returned by the search function. Next time i = 1 then $S1 = 2^1 = 2$, and $S[i]$ = pow(2, i-1) to pow(2, i) i.e the value of $S1$ is 1 to 2 and so on for all values of 'i'.

Step3. Re-balancing: Re-balancing is done after inserting 2^i elements where i = 1, 2, 3, 4... and the spaces are added when re-balancing is called. In the previous approach, the gaps were uniform in nature. In the proposed technique, non-uniform gap distribution is given based on the property of insertion sort. This property tells that more updates should be done in the beginning of an array for generating more gaps. Gaps are generated using the equation (5.1.2).

$$Ratio = n * ((\mu/\sigma)/2) \tag{5.1.1}$$

Here μ is mean and σ is standard deviation.

$$ee = 2 * (n/ratio) \tag{5.1.2}$$

Initially we have e+ee gaps, but 'ee' is decreased when we have parsed number equal to the ratio.

Algorithm 15 LNGD Re-balancing

INPUT: List of elements n and re-balancing factor e.

OUTPUT: List with non-uniform gaps.

```
Compute μ and σ
Ratio← n*( μ/σ)/2
ee←2*n/ratio
if (j% ratio ==0 and j>0 and e+ee>0 ) then
    ee–
end if
while (l < n) do
    if (S[j] != -1) then
        reba[i] = S[j]
        i++
        j++
        l++
        for (k=0 to ee+e) do
            reba[i] = -1
            i++
        end for
    else
        j++
    end if
    for (k = 0 to i) do
        S[k] = reba[k]
    end for
end while
```

5.2 Performance Evaluation

Execution Time Testing and Comparison of LUGD and LNGD

We have tested the LUGD and LNGD algorithms on a dataset [T10I4D100K(.gz)] [31] by increasing the value of the gap (ε). The dataset contains 1010228 items. We have tested four cases of the data set.

(1) Random with repeated data (Random data)

(2) Reverse sorted with repeated data (Reverse sorted data)

(3) Sorted with repeated data (Sorted data)

(4) Nearly sorted with repeated data (Nearly sorted data)

Table 5.1, shows the execution time of LUGD and LNGD algorithms in microseconds using the above mentioned cases.

Table 5.1: Execution Time of Library Sort Algorithm in Microseconds Based on Gap Values

Dataset	Random		Nearly Sorted		Reverse Sorted		Sorted	
Value of ε	LUGD	LNGD	LUGD	LNGD	LUGD	LNGD	LUGD	LNGD
$\varepsilon = 1$	981267433	862909204	864558882	306063385	1450636163	1328993502	861929937	313078205
$\varepsilon = 2$	729981576	708580455	620115904	230939335	1065247938	1022310950	609647355	234697961
$\varepsilon = 3$	119727535	101921406	358670053	185759986	278810310	125152235	356489846	195120953
$\varepsilon = 4$	23003046	10557332	117188830	107729204	263693774	116417058	116590140	106897060

The performance of the LUGD and LNGD are compared with random data, nearly sorted data, reverse sorted and sorted data. The execution time in microseconds are presented in Table 5.1. The Results are presented for different value of 'ε'. Epsilon (ε) is the minimum number of gaps between the two elements. The execution time comparison of LUGD and LNGD algorithms has also been shown in Figure 5.1 to 5.4. In all Figure 5.1 to 5.4, the X-axis represents the different value of gap and the Y-axis represents the execution time in microseconds.

Figure 5.1: Execution time comparison between LUGD and LNGD using random data

Figure 5.1 shows the comparison of LUGD and LNGD for different values of gap. It can be seen from the graph that the LNGD has outperformed LUGD.

The maximum improvement in execution time by LNGD is 36.7% for the value of $\varepsilon = 4$.

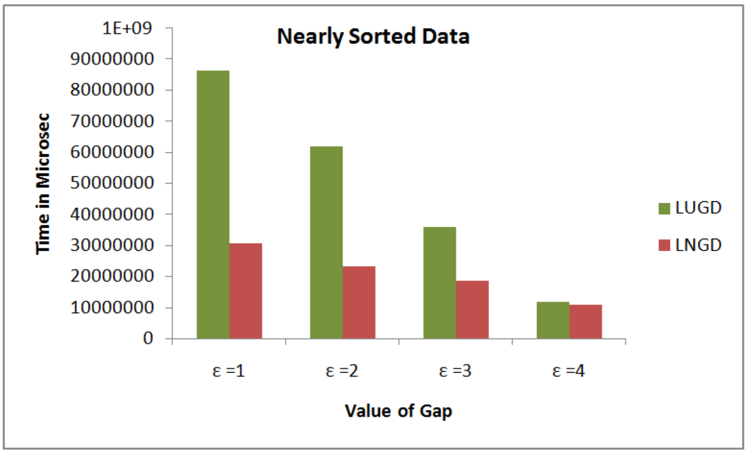


Figure 5.2: Execution time comparison between LUGD and LNGD using nearly sorted data

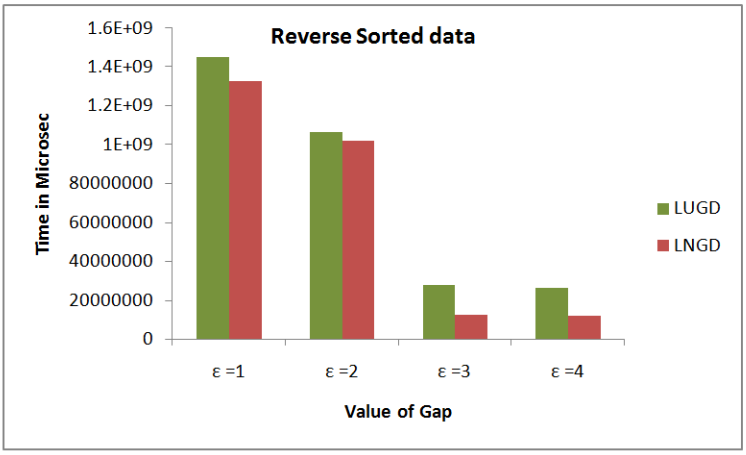


Figure 5.3: Execution time comparison between LUGD and LNGD using reverse sorted data

Figure 5.2 describes the execution time of the two algorithms LUGD and LNGD on the nearly sorted data. We found major improvement in the case of $\varepsilon = 1$. We also observed that the improvement in execution time by LNGD is 64.59% at $\varepsilon = 1$. With observations, we have found that execution time is inversely proportional to the value of 'ε'. The execution time is calculated as 8% in the case of $\varepsilon = 4$.

In the case of reverse sorted data the trend for execution time is reversed. It is nearly 8% for the $\varepsilon = 1$, and it further decreases for $\varepsilon = 2$, $\varepsilon = 3$ and $\varepsilon = 4$ up to 55%. The same has been shown in Figure 5.3.

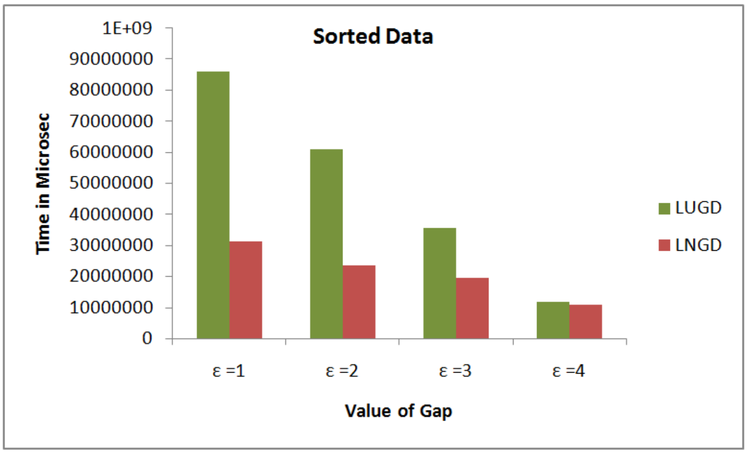


Figure 5.4: Execution time comparison between LUGD and LNGD using sorted data

Figure 5.4 describes the execution time of both the algorithms on the sorted data, the improvement can be seen from the $\varepsilon = 1$ to $\varepsilon = 4$. It is maximum at $\varepsilon = 1$ that is 63.67% and minimum at $\varepsilon = 4$ that is 8%.

5.3 Re-balancing based Testing and Comparison of LUGD and LNGD

We use re-balancing after inserting a^i element, which increases the size of the array. The size of the array depends on 'ε'. In this process, we require an auxiliary array of the same size therefore an array having the same values with gaps. We have calculated re-balancing till a^i where $a = 2, 3, 4$ and $i = 0, 1, 2, 3, 4....$ with the value of gaps $\varepsilon = 1, 2, 3, 4$.

(A). Example of re-balancing using LUGD algorithm

(1). Example for $\varepsilon = 1$ and $a = 2$.

$2^i = 2^0, 2^1, 2^2, 2^3, 2^4$

$= 1, 2, 4, 8, 16$

(1.1). Re-balance for $2^0 = 1$

1	**-1**

(1.2). Re-balance for $2^1 = 2$

1	**2**

After re-balancing, this array is as follows:

1	**-1**	**2**	**-1**

(1.3). Re-balance for $2^2 = 4$

1	**2**	**3**	**4**

After re-balancing, the array is:

1	**-1**	**2**	**-1**	**3**	**-1**	**4**	**-1**

(2). Example for $\varepsilon = 1$ and a = 3.

$3^i = 3^0, 3^1, 3^2, 3^3, 3^4$ .....

$= 1, 3, 9, 27$.....

(2.1). Re-balance for $3^0 = 1$

1	**-1**

(2.2). Re-balance for $3^1 = 3$

In the above array only one space is empty. This shows that only one element can be inserted. On the other hand, according to re-balancing factor $3^1 = 3$ we require two spaces in the array. In this situation we need to shift the data to make space for the new element. In this way performance of the algorithm degrades as we are having the larger number of swapping to generate the spaces which is same as that in the case of traditional insertion sort.

(B). Example of re-balancing using LNGD algorithm

(1). Example for a=2.

$2^i = 2^0, 2^1, 2^2, 2^3, 2^4$ ....

= 1, 2, 4, 8, 16 .....

In the proposed algorithm we have used two parameters '*ee*' and 'ratio' which is defined prior in the algorithm along with the value of gaps. To understand this concept, we consider an example; say the list to be sorted is 1, 2, 3, and 4. The average and standard deviation are calculated first. The mean and standard deviation are calculated to 2.5 and 1.2 respectively. The ratio and '*ee*' is calculated using equation (5.1.1) and (5.1.2). Ratio = 3. $ee = 8/3 = 2$ as integer The total gaps is 1+2 = 3

(1.1). Re-balance for $2^0 = 1$

After re-balancing, the array is:

1	**-1**	**-1**	**-1**

(1.2). Re-balance for $2^1 = 2$

In this case initially $j = 1$ that means we have e+ee gaps that is equal to 3. At second iteration 'j' is equal to 2, now we have the condition that is j = ratio so we decrement the value of '*ee*' by 1. Initially we have 3 gaps, then 2 gaps.

After re-balancing, the array is:

1	**-1**	**-1**	**-1**	**2**	**-1**	**-1**

(1.3). Re-balance for $2^2 = 4$

1	**2**	**3**	**4**

After re-balancing, this array is as follows:

1	**-1**	**-1**	**-1**	**2**	**-1**	**-1**	**3**	**-1**	**-1**	**4**	**-1**

Initially have 3 gaps for j=1.

- For j=2, j%ratio is equal to zero, therefore '*ee*' will be decremented by 1.
- For j=3, the value remains unchanged to 2 gaps.
- For j=4, again the value is decremented by 1 so there is only single gap.

(2). Example for $a = 3$.

$3^i = 3^0, 3^1, 3^2, 3^3, 3^4$ ....

= 1, 3, 9, 27......

(2.1). Re-balance for $3^0 = 1$

After re-balancing, the array is described as:

1	**-1**	**-1**	**-1**

(2.2). Re-balance for $3^1 = 3$

1	**2**	**3**

After re-balancing, the array is as follows:

1	**-1**	**-1**	**-1**	**2**	**-1**	**-1**	**3**	**-1**	**-1**

The spaces are calculated using the equation (5.1.2). In the similar manner, re-balancing of the array for the remaining value of the 'a' is held till the re-balancing is not possible. The reason for re-balancing not being possible is that in the array according to requirement spaces is not possible. In this way performance of algorithm degrades as we have larger number of swaps to generate the spaces which is same as that in the case of traditional insertion sort.

Table 5.2 describes the execution time of the LUGD and LNGD algorithm using different type of data set that are random data, nearly sorted data, reverse sorted data and sorted data. Along with the different dataset value, the table also describes the value of 'ε' and re-balancing factor 'a'. The re-balancing comparison of LUGD and LNGD algorithms is shown in Figure 5.5 to 5.8. From Figure 5.5 to 5.8, the X-axis represents the value of gap (ε) and re-balancing factor(a) and Y-axis represents the execution time in microseconds.

Table 5.2: Execution Time of Library Sort Algorithm in Microseconds Based on Gap Values

	Dataset	**Random**		**Nearly Sorted**		**Reverse Sorted**		**Sorted**	
$Re-balancing$	**Value of ε**	**LUGD**	**LNGD**	**LUGD**	**LNGD**	**LUGD**	**LNGD**	**LUGD**	**LNGD**
2	$\varepsilon = 1$	981267433	862909204	864558882	306063385	1450636163	1328993502	861929937	313078205
	$\varepsilon = 2$	729981576	708580455	620115904	230939335	1065247938	1022310950	609647355	234697961
	$\varepsilon = 3$	119727535	106921406	358670053	185759986	278810310	125152235	356489846	195120953
	$\varepsilon = 4$	23003046	14557332	117188830	107729204	263693774	116417058	116590140	106897060
3	$\varepsilon = 1$	2622591059	869209660	2214715182	308010395	2832112301	1556949795	3011802732	339671533
	$\varepsilon = 2$	2103580421	709709631	1964645906	231895871	2585747568	1280383725	2651992181	249815851
	$\varepsilon = 3$	2043974421	666999939	1728175857	185620741	2195021514	1239442185	1962122927	206018101
	$\varepsilon = 4$	1620914312	657130080	1600879365	155075390	2130261056	1263332585	1620374625	170410859
4	$\varepsilon = 1$	2942693856	912631839	2467933298	300698051	3239333534	1656255915	3281368964	367329723
	$\varepsilon = 2$	2705332601	484314092	2510103530	227562280	3154811065	1545638611	2923182920	266428823
	$\varepsilon = 3$	2676681610	327850683	2613423098	183893005	3013676930	1366501241	2378347887	210265375
	$\varepsilon = 3$	2611656774	146342570	2157740458	153786055	2993363707	1270043884	2222906193	181557846

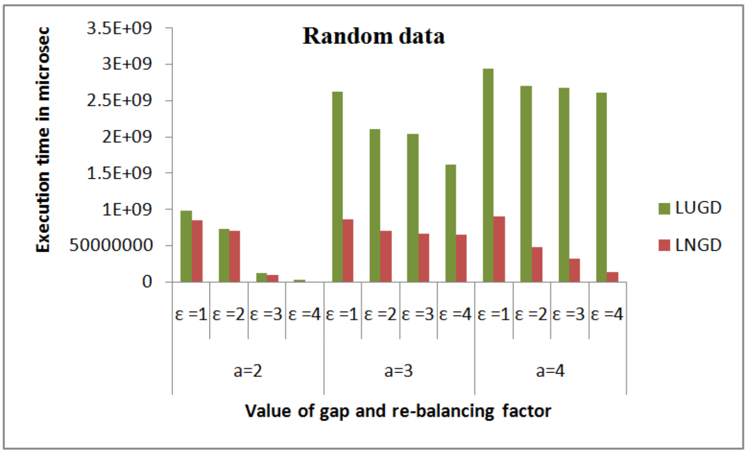


Figure 5.5: Re-balancing execution time comparison between LUGD and LNGD using random data

Figure 5.5 describes the plot at random data for the different values of 'ε' and re-balancing factor 'a'. It is observed from Figure 5.14, as if we increase the re-balance factor in the case of LUGD the execution time also increases significantly, but in the case of LNGD the improvement of execution time achieved upto 94% in comparison to LUGD.

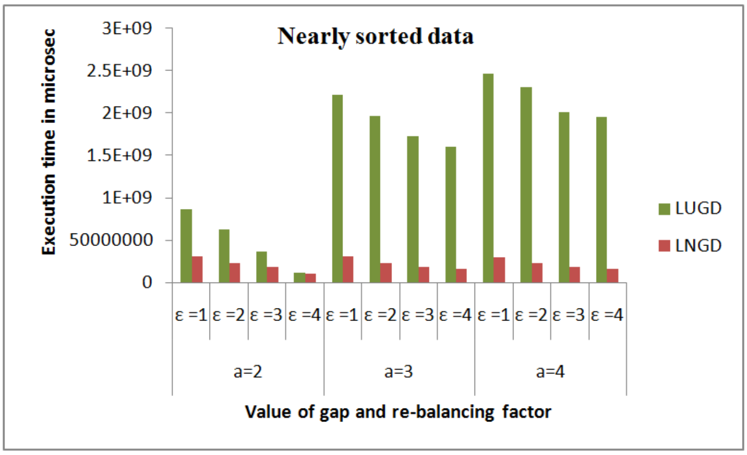


Figure 5.6: Re-balancing execution time comparison between LUGD and LNGD using nearly sorted data

Figure 5.6 shows the comparison of LUGD with LNGD at different gaps and re-balancing factors. Again the improvement is upto 92% at a = 4 and $\varepsilon = 4$.

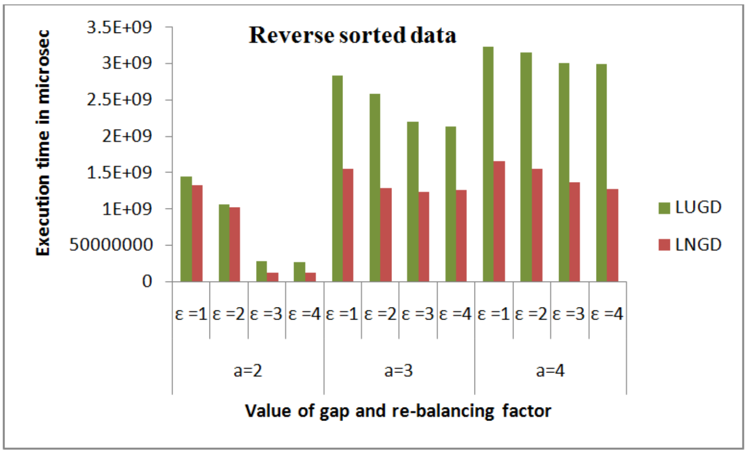


Figure 5.7: Re-balancing execution time comparison between LUGD and LNGD using reverse sorted data

Figure 5.7 shows the comparison of execution time of reverse sorted data at the different values of 'ε' and re-balancing factor 'a'. Initially, in this case results are improved by 8% but maximum upto 57% at $\varepsilon = 4$ and $a = 4$.

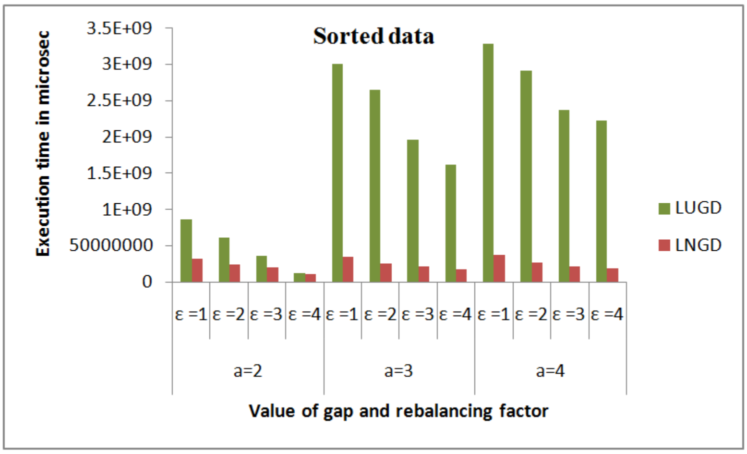


Figure 5.8: Re-balancing execution time comparison between LUGD and LNGD using sorted data

Figure 5.8 represents the result on the sorted data with the different value of gaps and re-balancing factor. The result shows that maximum improvement achieved is upto 91% in comparison to that of LUGD.

5.4 Conclusion

The final conclusion of this chapter is that, the proposed approach of LNGD proved to be a better algorithm in comparison to that of LUGD. We have achieved an improvement that ranges from 8% to 90%. The improvement of 90% has been found in the cases where the LUGD was performing poorer. We have also found that the performance of LNGD is better for different values of re-balancing factor which was not achieved in the case LUGD. The LNGD and LUGD both algorithms are implemented in C language.

In future, we will investigate the locality of data in more details. This will help not only in allocating the spaces accurately, but may also reduce the extra spaces which have been allocated and will act as an overhead both on the space and execution time of the program.

Chapter 6

Performance Enhancement of Bucket Sort using Hybrid Algorithm

The bucket sort is a simple and non-comparison sorting algorithm [78] [64]. Bucket sort is useful only when the input number is distributed over a range. In another word, bucket sort works based on the key range. The bucket sort is also called bin sort [62] [79]. The working of the bucket sort algorithm is as follows:

(1). The first array is set up for initial empty buckets.

(2). From the original array scatter each element over the buckets.

(3). Sort each bucket using some other sorting approach.

(4). After sorting, gather the elements in order from each bucket in the original array.

The same working of bucket sort is also explained with the help of example as shown in Figure 6.1.

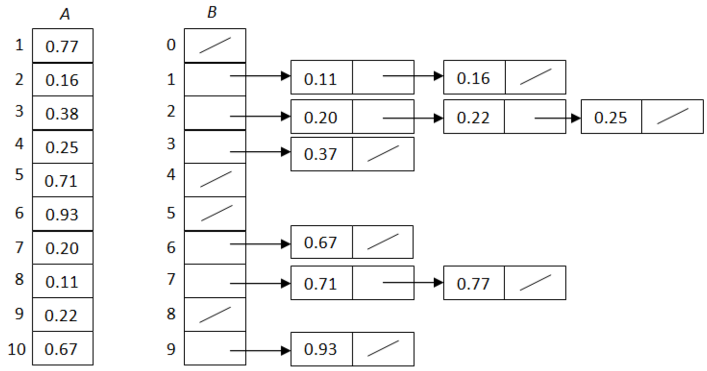


Figure 6.1: Example of Bucket Sort

In this example the array A[1...10] and the array B [0....9]. The sorted output is obtained from concatenated in the order of the list from B[0], B[1]....B [80].

6.1 Objective

The bucket sort has two issues **(1)** Firstly it has the dynamic nature and the memory is allocated for each bucket at run time. **(2)** The second issue is based on the data distribution over the buckets. If the data is not equally distributed over the buckets, then managing of buckets is a bit difficult. This allows us to perform two tasks. **(a)** Optimize the memory requirement according to the elements in a buckets, in this way the wastage of memory resources will be reduced. **(b)** Manage the buckets when data is not equally distributed.

The idea behind the hybrid sort is to save the space and can provide better results in terms of time. We would defined the threshold (τ) in order to design the hybrid sort. The threshold is calculated for each bucket and different size of data sets. It will helpful to decide the nature of data and to reduce the memory consumption. We have tested three algorithms acts as a local sort in the buckets are merge sort, count sort and proposed merge count is hybrid sort. The testing has been done using sorting benchmark which has six type of test cases. The derived results show that the proposed algorithm achieved the success in comparison of bucket with merge sort in both aspects (space and time). The splitting of buckets is shown in the picture as follows in Figure 6.2.

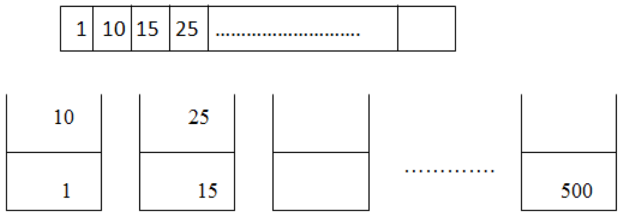


Figure 6.2: Distribution of buckets

We have solved this problem using the threshold (τ) for the size of buckets. The threshold is calculated for each bucket and different size of data sets. It will helpful to decide the nature of data and to reduce the memory consumption.

In this chapter, we have designed the hybrid sort in order to overcome this problem of bucket sort. The main contribution of the proposed work is as follows:
(1). The hybrid sort is a mixture of count and merge sort which acts as a local sort inside the buckets to sort the data.
(2). Count sort consumes more space so to overcome this, the term threshold is defined which is represented by 'τ'.
(3). The value of 'τ' depends on the range and number of buckets.
(4). In the hybrid sort, if the number of inputs in a bin is greater or equal to the threshold than count sort will run otherwise merge sort will run. This will reduce the amount of auxiliary memory required by the bins.

6.2 Proposed Hybrid Sort Algorithm

The idea behind the hybrid sort is to save the space and that can also provide better results in terms of time. We know that the count sort is a fastest sorting algorithm, but consumes more space which is based on the range of elements to be sorted. So, to provide the competitive results of the proposed hybrid sort our firstly we have used the count sort. To overcome the drawback of the count sort in terms of space we have defined the term named as threshold and is represented by the symbol 'τ'. The value of 'τ' depends on the range and number of buckets. It is given by the equation described below:

$$\tau = R/B \tag{6.2.1}$$

In the above equation R is the range of the element that can be easily identified or it is obtained from the prior knowledge of the problem. In general the value R is calculated by the expression given below.

$$R = MAX(A) - MIN(A) \tag{6.2.2}$$

Here A represents the set of elements that provides the input to the bucket sort and B is the number of buckets that are used for sorting. Now 'τ' will guarantee that if the element in a bucket are n then the maximum space required is also n. This is further proved in theorem 1. If the number of elements in a particular bucket are less than the 'τ' then we have selected the merge sort algorithm which is a second most efficient algorithm. The Auxiliary space required by the merge sort is $O(n)$. So we can say that we have designed an approach which is limited the space complexity $O(n)$. Now in case of time complexity proposed hybrid sort have the complexity limited to $O(n\log n)$ in the worst case and limited to $O(n)$ in best case.

Theorem 1: The maximum space requires by hybrid sort is restricted to the number of elements in the bucket.
Proof: We can have the following case to prove our statements
Case 1: When we have very few elements in the bucket.
If we have few elements, then 'τ' is greater than count this implies that we are using the merge sort algorithm and we know that the auxiliary space required by merge space is *'n'*.
Case 2: When we have more elements in the bucket. If we have more element in the bucket then 'τ' is less or equal to the number of elements in the bucket, as the buckets are equispaced and depend on a range. This implies that we are

using the count sort algorithm and count sort is depends on the range which is less than elements that is n. Hence, based on the case 1 and case 2 we can say that the memory required by our proposed hybrid sort algorithm is less than or equal to 'n'.

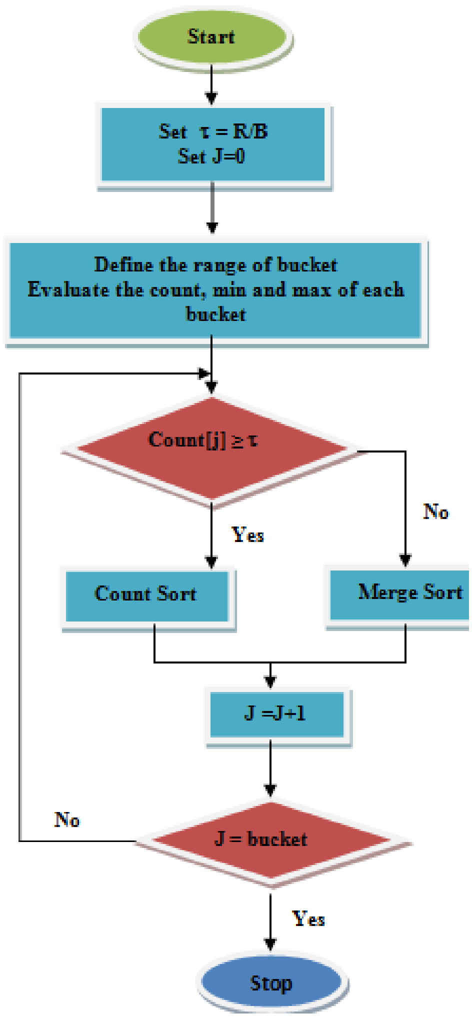


Figure 6.3: Flowchart of Proposed Hybrid Sort

Algorithm 16 Proposed Hybrid Sort

INPUT: Unsorted list (A, n, Max, Min, B)
OUTPUT: List with non-uniform gaps.

$\tau = R/B$
Evaluate count [] that contains the count in each bucket
Evaluate Min [] that contains the Min in each bucket
Evaluate Max [] that contains the Max in each bucket
For each j representing bucket B REPEAT step 6 and 7
if ($count[j] \geq \tau$) **then**
    call count sort
else
    call merge sort
end if

6.3 Implementation and Experimental Details of Execution Time

Sorting benchmark has been used to test the algorithms. The data size has varied from $n = 1000$ to $n = 10000000$. We have executed the three algorithms which are bucket with count, bucket with merge and proposed hybrid sort. The proposed hybrid sort contain the bucket with merge and count sort in order to sort the input data. The Table 6.1 summarize the execution time of bucket with merge sort in microseconds. Table 6.1 consists the execution time evaluation of six types of test cases.

Table 6.2 summarizes the execution time of bucket with count sort in microseconds. Here also we have evaluated the execution time using six types of test cases using sorting benchmark. The zero test case has achieved the lesser execution time among others. It is because we are using one unique value in this test case.

Table 6.3 summarizes the execution time of proposed hybrid approach in microseconds. Here also we have evaluated the execution time using six types of test cases using sorting benchmark. The zero test case has achieved the lesser execution time among others. It is because we are using one unique value in this test case. The zero test case has the greater execution time at n=1000 in comparison to uniform test case. It is observed that as we increase the input value the execution time also increases and it is measured and compared that where algorithms time are more predictive. So, if we analyze the Table 6.3 then we can see that zero test case is more efficient than others.

Table 6.1: Execution Time in Microseconds of Bucket with Merge Sort

Data Size	Buckets	Threshold	Uniform	Bucket	Gaussian	Sorted	Staggered	Zero
1000	10	100	288	288	255	369	299	157
5000	10	500	1561	1616	1871	893	1337	844
10000	10	1000	3187	3214	2345	1770	3847	1706
25000	10	2500	8301	8167	7453	7135	5971	4431
50000	10	5000	17326	17363	11407	13258	11804	10924
100000	10	10000	126330	45763	22332	23236	17315	21636
500000	10	50000	126330	96887	74469	44236	66083	78691
1000000	10	100000	202372	267329	106931	88435	195731	143183
2500000	10	250000	428226	411191	273599	229846	235101	263882
5000000	10	500000	921813	873912	476948	473334	540467	442345
10000000	10	1000000	1763002	1787513	1020353	979579	1031871	888723

Table 6.2: Execution Time in Microseconds of Bucket with Count Sort

Data Size	Buckets	Threshold	Uniform	Bucket	Gaussian	Sorted	Staggered	Zero
1000	10	100	210	209	126	328	104	105
5000	10	500	628	865	547	717	291	490
10000	10	1000	1992	1421	1055	1205	793	954
25000	10	2500	4164	3123	2197	3028	1979	2011
50000	10	5000	8030	6395	5282	5227	4090	4096
100000	10	10000	15658	8056	10737	10221	9380	9118
500000	10	50000	37869	38936	36503	22833	26704	26677
1000000	10	100000	66718	55180	54219	40114	45151	49566
2500000	10	250000	139848	132886	111055	103042	107318	70051
5000000	10	500000	277671	262354	233062	198703	190298	120295
10000000	10	1000000	546347	543388	462087	393468	378365	251691

Table 6.3: Execution Time in Microseconds of Proposed Hybrid Approach

Data Size	Buckets	Threshold	Uniform	Bucket	Gaussian	Sorted	Staggered	Zero
1000	10	100	277	461	338	462	401	365
5000	10	500	1553	1610	1104	1025	952	860
10000	10	1000	2919	3020	2052	1728	1904	1063
25000	10	2500	5981	7333	3840	4265	3480	2057
50000	10	5000	12492	14164	6912	7958	6720	4016
100000	10	10000	20192	24604	12318	10201	12103	9312
500000	10	50000	64534	70311	48866	27132	31762	36509
1000000	10	100000	111528	117848	102161	53504	50300	54940
2500000	10	250000	231000	288767	137958	126048	122385	94768
5000000	10	500000	570573	625717	273679	294370	248762	149459
10000000	10	1000000	1135702	1251074	574789	606673	507880	285924

The Figure 6.4 to 6.9 are presented by using the values of Table 6.1, 6.2 and 6.3. In all the Figure 6.4 to 6.9.

- P stands for proposed hybrid approach.
- M stands for bucket with merge sort.
- C stands for bucket with count sort.

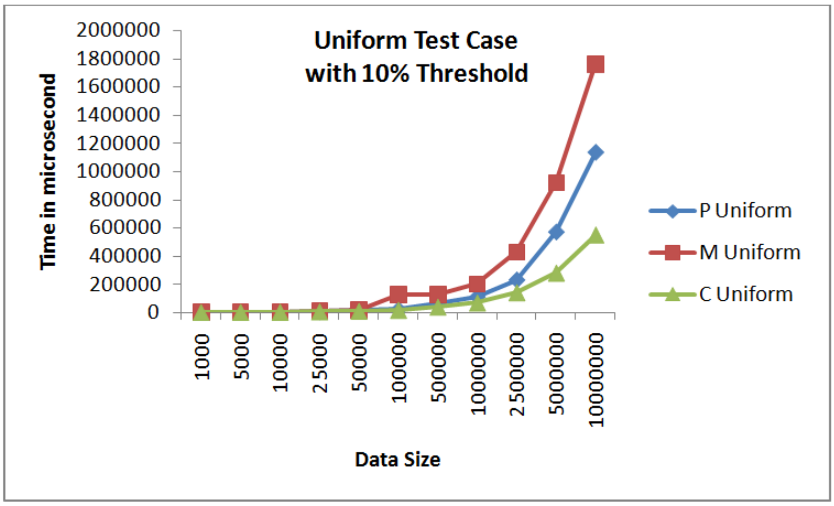


Figure 6.4: Execution time comparison of uniform test case

In Figure 6.4 to 6.9, the X-axis represents the size of the input data and the Y-axis represents the execution time in microseconds. If we compare the algorithms at the small size of the input then we will not recognize that which algorithm is more efficient. So to recognize the efficient algorithm we have tested the algorithm at the large size of the input. In the Figure 6.4 to 6.9, we can see that when the input size is small then all the discussed algorithms behave nearly equal, but we can see the difference at large. The execution time comparison of bucket sort using uniform test case is illustrated in Figure 6.4. The figure infers that proposed approach achieved the execution time greater than bucket with merge sort, but less than bucket with count sort. The proposed approach is given 35 times faster results than bucket with merge sort.

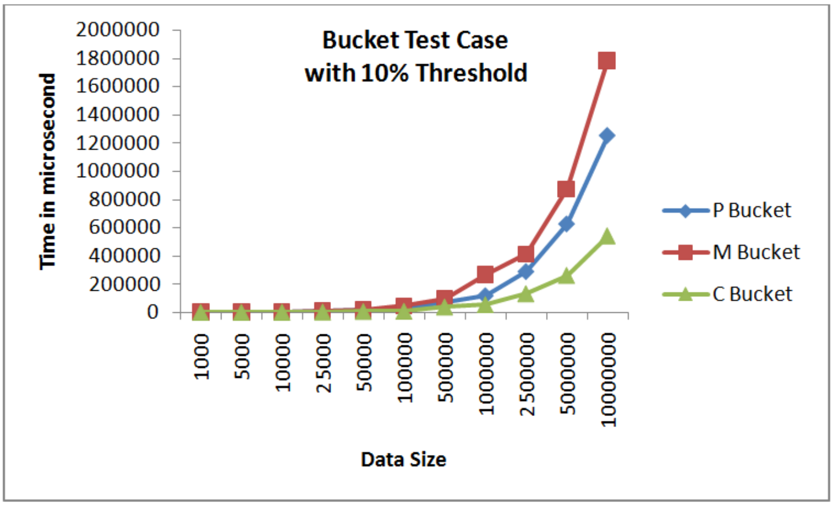


Figure 6.5: Execution time comparison of bucket test case

The execution time comparison of bucket sort using bucket test case is illustrated in Figure 6.5. The figure depicts that proposed approach achieved the execution time greater than bucket with merge sort, but less than bucket with count sort. In this test case proposed approach achieved 30 times faster results than bucket with merge sort.

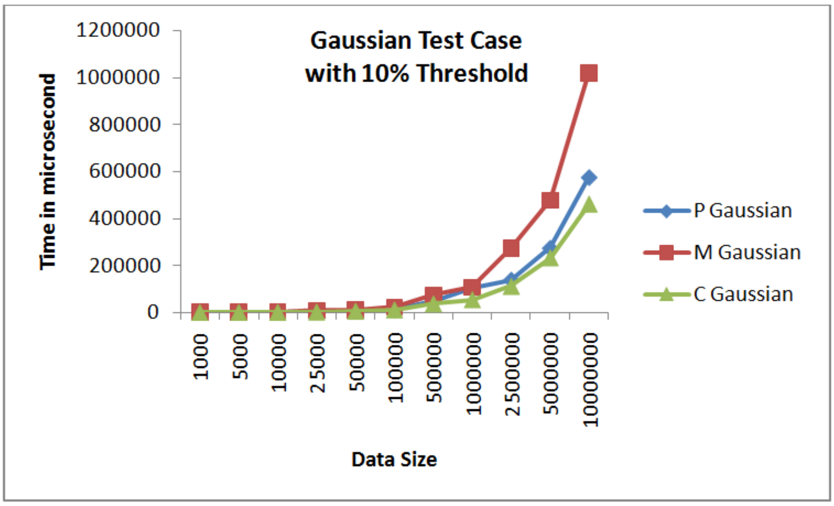


Figure 6.6: Execution time comparison of gaussian test case

The execution time comparison of bucket sort using Gaussian test case is illustrated in Figure 6.6. The figure infers that proposed approach achieved the execution time greater than bucket with merge sort, but less than bucket with count sort. In this test case proposed approach achieved the 43 times faster results than bucket with merge sort.

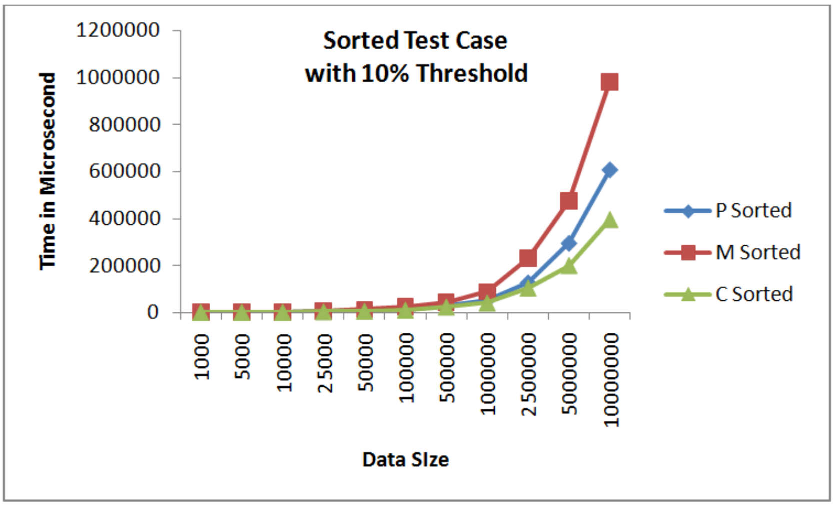


Figure 6.7: Execution time comparison of sorted test case

The execution time comparison of bucket sort using sorted test case is illustrated in Figure 6.7. The figure depicts that proposed approach achieves the execution time greater than a bucket with merge sort, but less than bucket with count sort. Here proposed approach achieves the 38 times faster results than existing one.

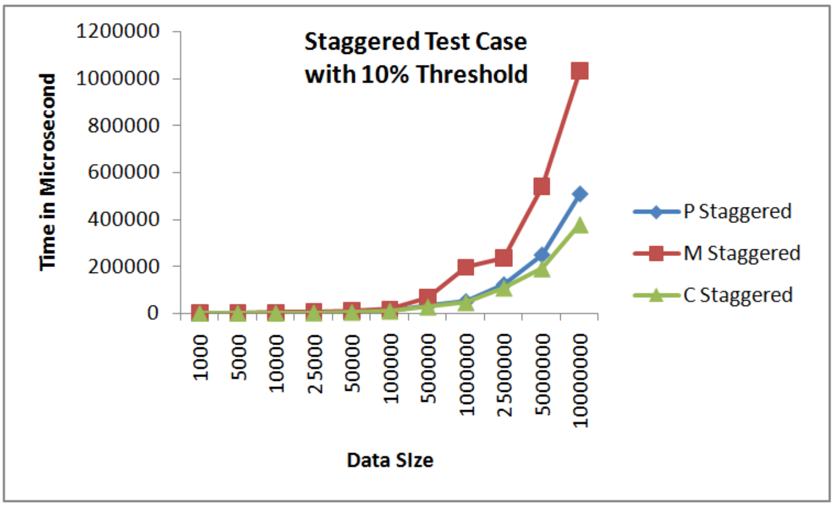


Figure 6.8: Execution time comparison of staggered test case

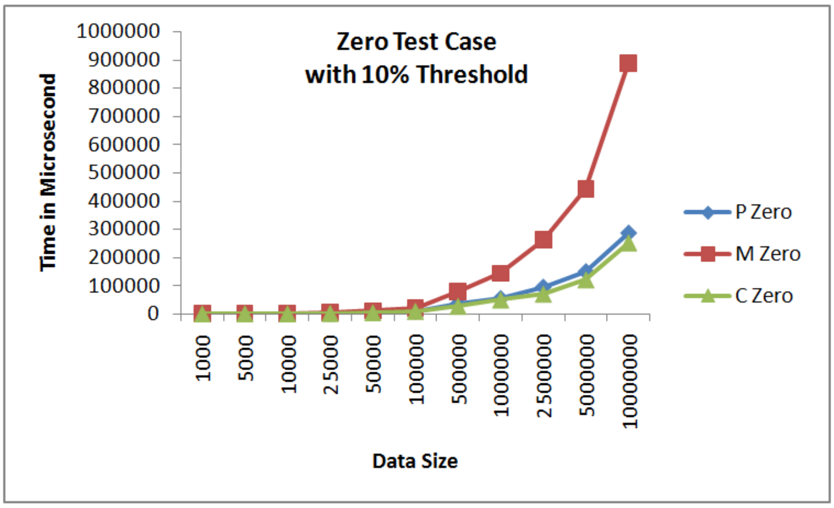


Figure 6.9: Execution time comparison of zero test case

The execution time comparison of bucket sort using staggered test case is illustrated in Figure 6.8. The figure infers that proposed approach achieved the execution time greater than bucket with merge sort, but less than bucket with count sort. The proposed approach achieved 50 times faster results in comparison to bucket with merge using staggered test case. The staggered test case achieved second most efficient results in comparison to other test cases.

The execution time comparison of bucket sort using zero test case is illustrated in Figure 6.9. The figure infers that proposed approach achieved the execution time better than bucket with merge sort, but almost comparable to bucket with count sort. Here proposed approach achieved 63 times faster results in comparison to the existing one. It is the for most efficient test case among others.

6.4 Implementation and Experimental Details of Auxiliary Memory Occupied by The Algorithms

Table 6.4: Auxiliary Memory Occupied by Bucket with Merge Sort in Bytes

Data Size	Buckets	Threshold	Uniform	Bucket	Gaussian	Sorted	Staggered	Zero
1000	10	100	4000	4000	4000	4000	4000	4000
5000	10	500	20000	20000	20000	20000	20000	20000
10000	10	1000	40000	40000	40000	40000	40000	40000
25000	10	2500	100000	100000	100000	100000	100000	100000
50000	10	5000	200000	200000	200000	200000	200000	200000
100000	10	10000	400000	400000	400000	400000	400000	400000
500000	10	50000	2000000	2000000	2000000	2000000	2000000	2000000
1000000	10	100000	4000000	4000000	4000000	4000000	4000000	4000000
2500000	10	250000	10000000	10000000	10000000	10000000	10000000	10000000
5000000	10	500000	20000000	20000000	20000000	20000000	20000000	20000000
10000000	10	1000000	40000000	40000000	40000000	40000000	40000000	40000000

In this section we have calculated the total auxiliary memory occupied by the already discussed algorithms. The memory is calculated for all the six types of test cases. We have varied the data from $n = 1000$ to $n = 10000000$ for memory calculation.

The Table 6.4 summarizes the results of memory occupied by the bucket with merge sort in bytes. As we know that the space complexity of merge sort is $O(n)$. Here inside the bucket we are using the merge sort as a local sort in order to sort the data. So, we can easily calculate the memory by using the input value multiplied by the size of integer.

The Table 6.5 summarizes the results of auxiliary memory occupied by the bucket with count sort in bytes. The memory occupied by count sort is based on range of the key value. In this paper, we have used the range of the key value of count sort from 0 to 65565. If the range of key is less then the memory will be occupied by the count sort is less and vice versa. In Table 6.5, Zero test case has very less memory in comparison to other test cases. It is because in Zero test case we use only one unique value which comes in one range value. In Table 6.5 some values are repeated. It is because we have multiple entries for the same element in the test case then the memory is calculated as the multiplication of number of times the element occurs and size of integer.

Table 6.5: Auxiliary Memory Occupied by Bucket with Count Sort in Bytes

Data Size	Buckets	Threshold	Uniform	Bucket	Gaussian	Sorted	Staggered	Zero
1000	10	100	257912	257912	205096	257912	1548	4
5000	10	500	261544	261544	222884	261544	312	4
10000	10	1000	261884	261884	230908	261884	144	4
25000	10	2500	261976	261976	246196	261976	48	4
50000	10	5000	262064	262064	248040	262064	24	4
100000	10	10000	262084	262084	248836	262084	24	4
500000	10	50000	262096	262096	253488	262096	24	4
1000000	10	100000	262096	262096	254892	262096	24	4
2500000	10	250000	262096	262096	255840	262096	24	4
5000000	10	500000	262096	262096	255840	262096	24	4
10000000	10	1000000	262096	262096	257988	262096	24	4

The Table 6.6 summarizes the results of auxiliary memory occupied by the proposed hybrid sort in bytes. The hybrid approach is a mixture of count and merge sort. So in Table 6.6 some test cases have, the less memory and some have more memory in comparison to bucket with count sort. If we increase the range of key, then bucket with count will occupy the more space and in this case our proposed approach will be more efficient in both aspects (time as well as space).

Table 6.6: Auxiliary Memory Occupied by Proposed Hybrid Sort in Bytes

Data Size	Buckets	Threshold	Uniform	Bucket	Gaussian	Sorted	Staggered	Zero
1000	10	100	79136	2504	104160	79136	2260	4
5000	10	500	140140	2520	108400	140140	10052	4
10000	10	1000	128164	2520	111904	128164	20024	4
25000	10	2500	196328	2520	122368	196328	50008	4
50000	10	5000	223824	2520	139800	223824	100004	4
100000	10	10000	329596	2520	173480	329596	200000	4
500000	10	50000	955712	2520	449492	955712	1000000	4
1000000	10	100000	2127696	2520	793968	2127696	2000000	4
2500000	10	250000	4147604	2520	1828480	4147604	5000000	4
5000000	10	500000	12090312	2520	3550968	12090312	10000000	4
10000000	10	1000000	24083760	2520	6996692	24083760	20000000	4

The Figure 6.10 to 6.15 is represented by using the values of Table 6.4, 6.5 and 6.6. In all the Figure 6.10 to 6.15.

- P stands for proposed hybrid approach.
- M stands for bucket with merge sort.
- C stands for bucket with count sort.

In Figure 6.10 to 6.15, the X-axis represents the size of the input data and the Y-axis represents the memory in bytes.

The memory comparison between proposed hybrid sort, bucket with merge sort and bucket with count sort using Uniform test case is shown in Figure 6.10. By analyzing this figure we found that proposed approach has taken less amount of memory in comparison to bucket with merge sort. The hybrid sort achieved 39 times more efficient memory consumption than bucket with merge sort. The suggested approach is not showing the better result in comparison to bucket with count. It is because the range of key element is used from 0 to 65565. So the elements in bucket with count sort are repeated so consumes less space, but if we increase the range of key element then our proposed approach will be efficient.

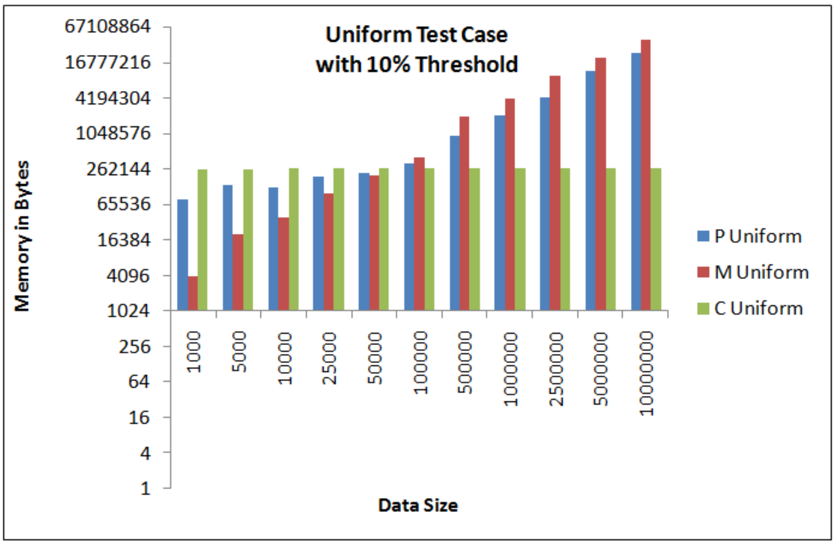


Figure 6.10: Memory comparison of uniform test case

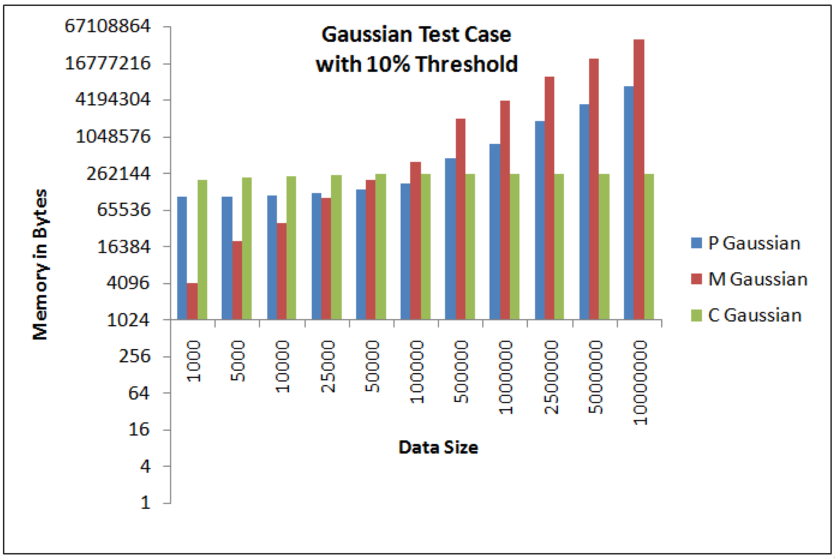


Figure 6.11: Memory comparison of bucket test case

The memory comparison between proposed hybrid sort, bucket with merge sort and bucket with count sort using Bucket test case is shown in Figure 6.11. The hybrid sort has achieved more efficient memory results compared to bucket with merge and count sort. It is because in bucket test case the nature of data is random and repetition of data is less. So the bucket with count sort is not more efficient than proposed hybrid sort till the range of key element is not changed. The hybrid sort gives 99 times more efficient results of memory in comparison to bucket with merge sort in the case of memory.

The memory comparison between proposed hybrid sort, bucket with merge sort and bucket with count sort using Gaussian test case is shown in Figure 6.12. In this test case the proposed hybrid sort is more efficient than bucket with merge and can be more efficient than bucket with count sort when we increase the range of key element.The hybrid sort achieved 82 times more efficient results of memory than bucket with merge sort.

The memory comparison between proposed hybrid sort, bucket with merge sort and bucket with count sort using Sorted test case is shown in Figure 6.13. The hybrid sort achieved 39 times more efficient results of memory than bucket with merge sort. The hybrid sort will also be more efficient than bucket with count sort when the range of key element will be high.

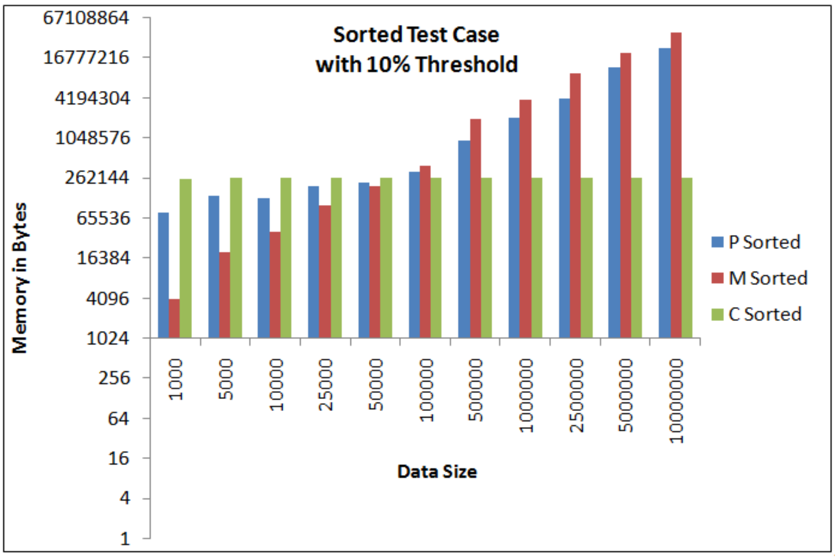


Figure 6.12: Memory comparison of gaussian test case

The memory comparison between proposed hybrid sort, bucket with merge sort and bucket with count sort using Staggered test case is shown in Figure 6.14. The hybrid sort achieved 50 times more efficient results of memory than bucket with merge sort. The hybrid sort will also be more efficient than bucket with count sort when the range of key element is high.

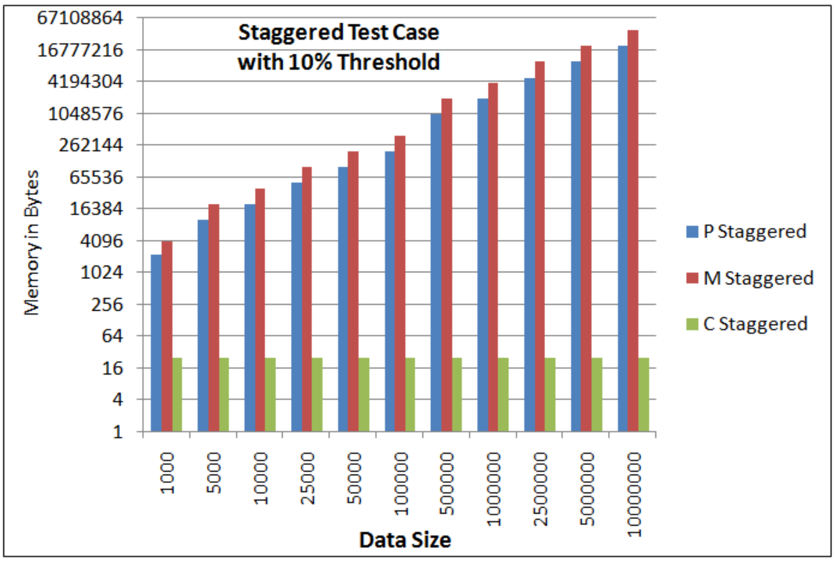


Figure 6.13: Memory comparison of sorted test case

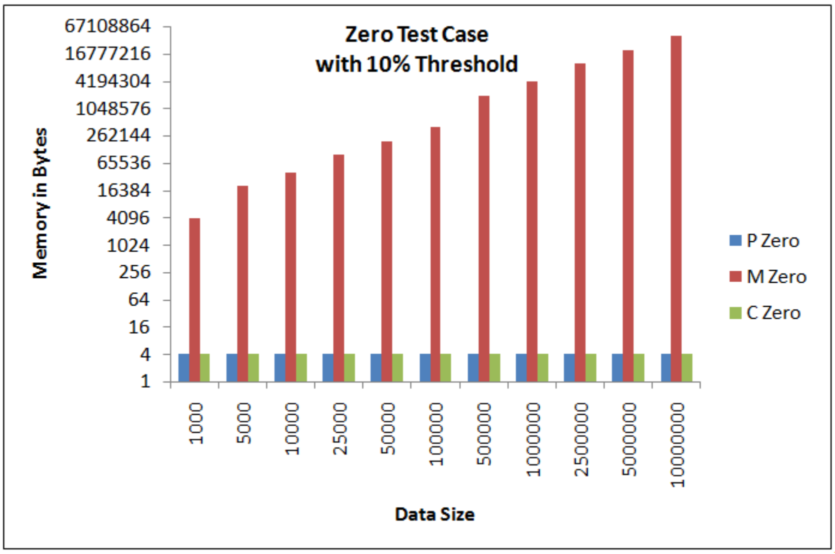


Figure 6.14: Memory comparison of staggered test case

6.5 Conclusion and Future Work

The final conclusion of this chapter is that, the obtained results shows that hybrid sort is efficient in terms of execution time and memory consumption than the bucket with merge sort. In case of bucket with count sort algorithm the results are always seen on top this is due to the reason that the range of the key elements is fixed. The range of key element taken between 0 to 65565 still in some cases we managed and got the space consumption almost equal to the count sort.

In future we can further classify other sorting algorithm like quick sort based on the number of elements in bucket which will not only make the working faster of bucket sort but also will reduce the time.

Chapter 7

Performance Enhancement of Bubble Sort using GPU Computing

7.1 Objective

Bubble sort is a comparison based sorting algorithm. The analysis of bubble sort using many core GPUs was previously unknown. The paper also presents the speedup achieved by the parallel bubble sort over sequential. The bubble sort (parallel & sequential) is tested using sorting benchmark. The sorting benchmark consists various test cases which are Uniform, Gaussian, Zero, Bucket, Staggered and Sorted test cases. On the basis of experimental analysis, parallel bubble sort achieved 37229 times faster execution time using zero test case and 6375 times faster using sorted test case at $n = 2500000$ and $T = 512$. The best case time complexity of the parallel bubble sort is reduced $O(n)$ to $O(1)$ because of the GPU.

7.2 Roadmap of GPU Sorting Algorithms

Greb *et al* presented the parallel sorting based on stream processing architecture in the year 2006. The proposed sorting is based on bitonic sort who is adaptive. The optimal time complexity of proposed approach achieved $O(n\log n)/p)$. The proposed algorithm is faster than sequential sorting. The proposed algorithm is designed on modern GPU, so the name GPU-ABiSort $O(n\log n)/p)$[81].

Inoue *et al* proposed the AA-sort which is parallel sorting algorithm. AA-sort stands for Aligned-Access Sort. AA-sort proposed for shared memory multiprocessors. The sequential version of the AA-sort is more beneficial for IBMs optimized sequential sorting using SIMD instructions[82].

Sintorn *et al* presented the fast algorithm to sort huge data using modern GPU. The implementation of the algorithm is fast due to the GPU. The proposed algorithm performed better than bitonic sort algorithms for the input list with more than 512k elements. The suggested approach is 6-14 times quicker than the single CPU quick sort of 1-8M elements [83].

Cederman *et al* presented the GPU Quick Sort. The proposed algorithm is extremely capable and suitable for parallel multi-core graphics processors. GPU quick sort performance represents the better performance than the fastest known GPU based sorting algorithms such as radix and bitonic sort [84].

Rozen *et al* presented the adoption of the bucket sort algorithm. The proposed algorithm is entirely run on the GPU. The proposed algorithm is implemented on GPU using OpenGL API [85]

Baraglia *et al* showed that how the graphics processor used as a coprocessor to speed up the algorithm and CPU also allowed doing the some other task. The proposed algorithm is used to memory efficient data access pattern to maintain the minimum number of access to the memory of the chip. The implementation results show the improvement in the GPU based sorting in order to CPU based sorting [86].

Leischner *et al* presented the GPU sample sort algorithm. The sample merge sort is the efficient comparison based sorting algorithm for distributed memory architecture. Previously the sample sort algorithm was unknown for the GPU [29].

Kukunas *et al* presented the GPU merge sort. In todays life high data throughput and computational power are increasing. The GPGPU architecture is created by NVIDIA. The GPU merge sort is highly efficient in comparison to a sequential version [87].

Oat *et al* presented the technique for sorting data into spatial bins using GPU. The proposed technique takes the unsorted data as input and scatters the points in sorted order into the buckets. The author proposed method is used to implement a form of bucket sort using GPU [88].

Huang *et al* proposed the empirical optimization technique. The empirical optimization technique is also important for sorting routines using GPU. The radix sort generated the highly productive code for NVIDIA GPU with a variety of architecture specification. The paper outcome showed that the empirical optimization technique is quite successful. The resulting code was more efficient than radix sort [89].

Ye *et al* presented GPU warp sort to carry out a comparison based parallel sort on the GPU. The warp sort is nothing but contain the bitonic sort followed by merge sort. The proposed algorithm achieved the high staging by depicting the sorting task on the GPU. The experimental results of GPU- Warpsort work well on various kinds of input distribution [90].

Peters *et al* presented the Batchers bitonic sorting network using CUDA hardware with GPUs. The arbitrary numbers has been taken as input and assigned compare-exchange operation to threads using adapted bitonic sort. The proposed algorithm has greatly increased the performance of implementation [91].

Peters *et al* presented the merge-based external sorting algorithm using CUDA sanction GPUs. The production influence of memory transfer is reduced using GPU. The better utilization of the GPU and load balancing is achieved. The performance of the algorithm is demonstrated by extended testing. The two main problems occur when using external sorting on GPUs [92].

Satish *et al* reported the comparison and non-comparison based sorting algorithms on CPUs and GPUs. The author has extended the work to the Intel Many Integrated Core (MIC) architecture. The radix sort evaluated on Knights Ferry and obtained the performance gain of 2.2X and 1.7 X. The production of the GPU radix sort improves nearly 1.6X over previous outcomes [93].

Helluy presented the portable OpenCL implementation of the radix sort. The algorithm was tested on several GPUs or CPUs in order to access the good performance. The implementation was also applied to the Particle-In-Cell (PIC) sorting. The application of the PIC is plasma physics simulations [94].

Krueger *et al* presented a technique, differential updates which are used to permit rapid modifications. The lead storage is allowed to the database to maintain data storage for accommodating the modifying queries. The author also presented the parallel dictionary slice merge algorithm and also GPU parallel merge algorithm that achieves 40% more throughput in comparison to CPU [95].

Misic *et al* represented an effort of sorting algorithms to analyze and implement in the graphics processing unit. Three sorting algorithms evaluated on the CUDA architecture. The evaluated algorithms are quick, merge and radix sort. CUDA platform used the NVIDIA GPU to execute applications [96].

Peters *et al* presented the novel optimal sorting algorithm which is similar to the adaptive bitonic sort. The popular parallel merge based sorting algorithm is the adaptive bitonic sort. It uses the tree like data structure to achieve the optimal complexity called a bitonic tree. The author presented the execution of the hybrid algorithm for GPUs based on bitonic sort [97].

Jan *et al* al presented examines three extensively used parallel sorting algorithms. The algorithms are Odd-Even sort, rank sort and bitonic sort. The comparative analysis is performed in terms of sorting rate, sorting time and speedup on CPU and different GPU architectures. The author achieved the high speed-up of NVIDIA quadro 6000 GPU for min-max butterfly network reaching much lower sorting for high data [98].

Munavalli developed a novel sorting algorithm on the GPU. Author focused on the vital problem. Author presented an efficient sorting algorithm which is Fine Sample Sort (FSS). The proposed algorithm extends and outperforms the sample sort algorithm. The results have shown that FSS outperforms sample sort by at least by 26% and on average 37% of data size ranging from 40 million and above for various input distributions [99].

Thouti *et al* presented the comparative performance analysis of various sorting algorithms. The algorithms are bitonic and parallel radix sort. Author implemented both the algorithms in OpenCL and compared with the quick sort algorithm. The author used the Intel Core2Due CPU 2.67 GHz and NVIDIA Quadro FX 3800 as GPU for the implementation [100].

Zurek *et al* described the implementation results for a few diverse parallel sorting algorithms using GPU cards and multi-core processors. The author presented the hybrid algorithm and executed on both platforms CPU and GPU. The comparison of many core and multi-core is lacking. The threads are grouped in blocks and the blocks are grouped in grids [37].

Panwar *et al* used the GPU architecture for solving the sorting problem. The highly parallel computing nature of GPU architecture is utilized for sorting purposes. The author considered the input array in the form of 2D matrix which is used for sorting. The modified version of merge sort is applied in that matrix. This work performed much efficient sorting algorithm with reduced complexity [101].

Garcial *et al* presented the fast data parallel implementation of radix sort using the Direct Compute software development kit (SDK). Author also discussed the optimization strategies in detail that are used to increase the performance of radix sort. The paper share the insights should be used in GPGPU (General

Purpose Graphics Processing Unit). Finally the author discussed how radix sort can be used to accelerate ray tracing [102].

Gluck *et al* introduced a method for fast quadtree construction on the GPU. The level-by-level approach is used to construct a quadtree. Quadtree is used for the spatial segmentation of lidar data points using a grid digital evaluation model (DEM). The author introduced an algorithm which is suitable for quadtree construction using GPU. The suggested algorithm reduces the construction problem of bucket sort [103].

Ye *et al* presented the GPU based sorting algorithm which is GPUMemSort. It achieved the highest performance in sorting. It has consisted two algorithms [104].

Polok *et al* focused on the implementation of extremely productive sorting routines for the sparse linear algebra operations. Testing results show that the suggested approach outperforms the other similar implementations. Author implementation is bandwidth efficient because sorting rate is achieved by it compare to the theoretical upper bound on memory bandwidth [105].

Mu *et al* described the bitonic sort algorithm in detail and implementation is done on CUDA architecture. The two effective optimization implementation details are conducted at the same time using the characteristics of the GPU which improves the efficiency. The experimental results show that GPU bitonic sort have 20 times more speed up to the CPU quick sort [106].

Xiao *et al* proposed the high performance approximate sort algorithm based on the CUDA parallel computing architecture running on multi-core GPUs. The algorithm divides the input into distribution multiple small intervals. The results showed that approximate sort is two times faster than radix sort and far exceeds all the GPUs-based sorting [107].

Ajdari *et al* described the modification of the Odd-Even sort. The modification of the algorithm consists in the ability to work with the blocks of elements instead working with individual elements. The modification is done using the

CUDA technology. The results showed that sorting of integers in CUDA environment is dozens of times faster [108].

Neetu *et al* presented the GPU merge and quick sort. The objective of the paper is to evaluate and analyze the achievement of merge and quick sort using GPU technology. Author also evaluated the parallel time and space complexity of both algorithms using dataset [63].

Table 7.1: Summary of the various articles

Author	Year	Work Done
Greb *et al*	2006	The authors proposed the ABiSort
Inoue *et al*	2007	The authors proposed the AA-sort
Centurion *et al*	2008	GPU-sorting using a hybrid algorithm
Cederman *et al*	2008	GPU quicksort for graphics processors
Rozen *et al*	2008	GPU bucket sort algorithm
Oat *et al*	2008	Efficient spatial binning on the GPU
Baraglia *et al*	2009	Sorting using bitonic network with CUDA
Leischner *et al*	2009	GPU sample sort
Kukunas *et al*	2009	GPGPU Parallel Merge Sort Algorithm
Huang *et al*	2009	An empirically optimized radix sort for gpu
Ye *et al*	2010	Comparison based sorting algorithm using GPU
Peters *et al*	2010	In-place sorting which is fast using cuda based on bitonic sort
Peters *et al*	2010	Parallel external sortusing CUDA-enabled GPU
Satish *et al*	2010	Fast sort which is based on cpus, gpus and intel mic architectures
Helluy	2011	Radix sort algorithm in OpenCL
Harada *et al*	2011	Introduction to GPU Radix Sort
Krueger *et al*	2011	Efficient Merge in In-Memory Databases using GPU
Misic *et al*	2011	Data sorting using graphics processing units
Peters *et al*	2012	Adaptive bitonic sort for many-core architecture
Jan *et al*	2012	Fast parallel sorting algorithms on GPUs
Munavalli	2012	Efficient Algorithms for Sorting on GPUs
Thouti *et al*	2012	Parallel Sorting Algorithms for GPU Architecture using Open Cl method
Zurek *et al*	2013	Parallel sorting compared with many hardware
Panwar *et al*	2014	GPU Matrix Sort (An Efficient Implementation of Merge Sort)
Garcia *et al*	2014	Fast Data Parallel Radix Sort
Gluck *et al*	2014	Fast GPGPU Based Quadtree Construction
Ye *et al*	2014	GPUMemSort
Polok *et al*	2014	Radix sort which is fast for sparse linear algebra on GPU
Mu *et al*	2015	The implementation and optimization of Bitonic sort algorithm based on CUDA
Xiao *et al*	2015	High Performance Approximate Sort Algorithm Using GPUs
Ajdari *et al*	2015	A Version of Parallel Odd-Even Sorting Algorithm Implemented in CUDA Paradigm
Neetu *et al*	2015	Merge and quick sort using GPU computing

7.3 Experimental Analysis of Sequential and Parallel Bubble Sort

Sorting benchmark has been used for testing the bubble sort. We have practiced the sequential and parallel bubble sort on six types of test cases using GPU computing with CUDA hardware. Table 7.2, displays the execution time in seconds of the sequential bubble sort. The 'n' is the size of the data used for the algorithm. The value 'n' is varied from 500000 to 2500000.

Table 7.2: Execution time in seconds of sequential bubble sort

n	Uniform	Gaussian	Zero	Staggered	Bucket	Sorted
500000	499.956	506.071	86.671	429.294	588.499	107.913
1000000	2071.283	2057.546	408.43	1868.969	2739.964	582.816
1500000	4677.309	5139.387	918.44	4849.295	6039.228	1348.617
2000000	8099.214	7669.999	2079.229	7964.588	11159.55	4126.261
2500000	17099.99	17134.94	37229.195	16179.63	15738.54	6375.703

Table 7.3, displays the execution time in seconds of the parallel bubble sort using different types of test cases. The input size is represented by 'n' and threads is denoted by 'T'. The values of 'T' vary from 1 to maximum 1024. The threads increase in the power of 2. The CUDA hardware version 2.1 has the total of 1024 threads per block so the maximum value of thread is selected as 1024.

Next, we evaluated the speedup achieved by a parallel bubble sort over the sequential bubble sort. Speedup measures performance gain achieved by parallelizing a given application over sequential application. From Table II, it can be observed that the execution time is minimum when the number of threads is 512. The performance of algorithm got degraded at $T = 1024$. The reason behind this is that, the data we have taken is not evenly distributed over the threads. So, some of the threads are executed ideally and degrading the overall performance of the algorithm. The speedup for all the six mentioned test cases is shown in Table 7.3 and Figure 7.1 to 7.6. The X-axis represents the size of data and the Y-axis represents the speedup achieved by the parallel bubble sort. We have calculated the speedup only for $T = 512$, similarly speedup can be calculated for the remaining values of T.

Table 7.3: Execution time in seconds of parallel bubble sort

n/T	Test case	1	2	4	8	16	32	64	128	256	512	1024
500000	Uniform	4874.8	3555.6	1925.6	1218.1	629.3	326.5	176.3	129.2	129.1	129	149.3
	Gaussian	4875.2	3525.9	1915.7	1219.8	629.5	325.4	175.2	129.4	129.3	129.2	149.3
	Zero	0.0358	0.0262	0.0093	0.0054	0.0032	0.0022	0.0017	0.0016	0.0016	0.0018	0.0021
	Staggered	4775.2	3425.9	1815.7	1119.8	616	315.8	167.2	125.4	125.3	125.3	145.3
	Bucket	4695.2	3394.9	1794.7	1198.8	585.5	298	159.4	129.3	129.6	129.2	149.4
	Sorted	0.03576	0.02631	0.0094	0.00535	0.00326	0.00216	0.00082	0.00062	0.00062	0.00059	0.00067
1000000	Uniform	18989.6	13449.8	6989.6	3753.7	1999.8	1289.9	689.4	507.7	505.4	505.3	583.4
	Gaussian	18935.6	13416.7	6935.6	3725.6	1959.7	1236.6	615.6	486.3	484.6	479.6	520.3
	Zero	0.0619	0.0427	0.0269	0.0098	0.0053	0.0033	0.0023	0.002	0.0021	0.0023	0.0026
	Staggered	18988.6	13477.8	6988.5	3788.4	1988.9	1277.6	688.9	574.3	571.6	568.6	611.3
	Bucket	18815.6	13339.8	6839.5	3659.4	1989.8	1199.6	659.8	527.4	513.4	511.6	615.3
	Sorted	0.06187	0.04265	0.02701	0.0098	0.00532	0.00322	0.00232	0.00206	0.00201	0.00099	0.00213
1500000	Uniform	60566.7	31457.7	15577.6	8358	4589.6	2241.7	1254.8	1146.6	1135.9	1132	1323.6
	Gaussian	60511.7	31411.7	15521.9	8305.7	4513.6	2111.7	1202.6	1105.7	1102.7	1101.7	1310.7
	Zero	0.0861	0.059	0.0352	0.0232	0.0085	0.0043	0.0028	0.0025	0.0025	0.0026	0.0032
	Staggered	60521.7	31396.6	15488	8293.9	4489.9	2079.7	1189.8	1098.8	1088.7	1079.7	1289.7
	Bucket	60411.7	30336.6	14428	8183.9	4579.6	2019.7	1099.9	1078.8	1068.7	1059.7	1309.7
	Sorted	0.08624	0.059	0.03521	0.02322	0.00851	0.00438	0.00275	0.00238	0.00237	0.0023	0.00262
2000000	Uniform	90845.8	46329.8	24345.8	12845.7	6838.9	3875.7	2056.7	1668.9	1649.9	1616.7	2016.6
	Gaussian	90659.3	46189.6	24189.6	12689.5	6679.6	3712.6	2002.6	1602.7	1600.7	1509.6	1909.6
	Zero	0.2021	0.0752	0.0436	0.0276	0.0019	0.0054	0.0033	0.0029	0.012	0.0011	0.0036
	Staggered	90759.3	46289.9	24289.5	12789.2	6779.6	3799.9	2069.9	1602.7	1509.7	1504.6	1899.6
	Bucket	90610.3	45124.9	23249.5	11779.2	6689.6	3689.9	2000.8	1482.7	1479.7	1464.6	1819.6
	Sorted	0.20196	0.07515	0.04359	0.02761	0.00996	0.00536	0.00342	0.00281	0.00279	0.0027	0.0031
2500000	Uniform	166205.5	82211.7	41817.7	22834.8	11887.7	7834.9	3376.5	2383.7	2354.9	2344.7	2884.9
	Gaussian	155803.7	82204.8	41253.6	22765.6	11754.6	7811.6	3332.7	2326.5	2323.5	2322.5	2832.3
	Zero	0.2281	0.0913	0.052	0.0321	0.0219	0.0079	0.0035	0.0029	0.0028	0.0027	0.0039
	Staggered	155898.7	82298.8	41353.9	22865.2	11854.6	7897.6	3389.9	2332.5	2324.5	2322.5	2889.3
	Bucket	155721.7	82198.8	41213.9	22745.2	11744.6	7767.7	3299.7	2229.5	2224.5	2222.5	2829.3
	Sorted	0.22802	0.0916	0.05196	0.0321	0.02186	0.00787	0.00386	0.00333	0.00328	0.0032	0.00367

Table 7.4: Speedup achieved by parallel bubble sort at T=512

T	n	**Uniform**	**Gaussian**	**Zero**	**Staggered**	**Bucket**	**Sorted**
512	500000	3.875628	3.389625	48150.56	3.426129	4.554946	161064.2
512	1000000	4.099115	4.290129	177578.3	3.286966	5.355676	588703
512	1500000	4.131898	4.664961	353246.2	4.491336	5.698998	586355.2
512	2000000	5.00972	5.080815	1890208	5.293492	7.619521	1528245
512	2500000	7.29304	7.3778	13788591	6.966471	7.814581	1992407

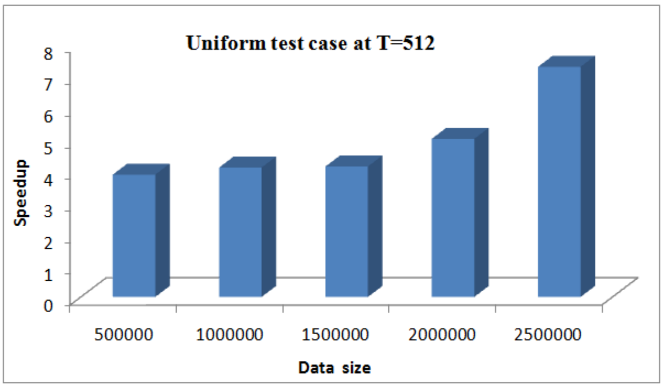


Figure 7.1: Speedup achieved by parallel bubble sort using uniform test case

The speedup achieved by the parallel bubble sort over sequential using uniform test case is presented in Figure 7.1. The topmost speedup obtained 7 times at $n = 2500000$.

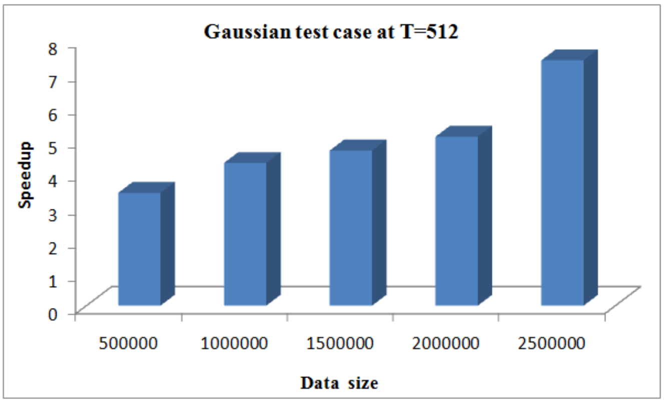


Figure 7.2: Speedup achieved by parallel bubble sort using gaussian test case

The speedup acquired by the parallel bubble sort over sequential using Gaussian test case is demonstrated in Figure 7.2. The maximum speedup achieved 7 times at $n = 2500000$.

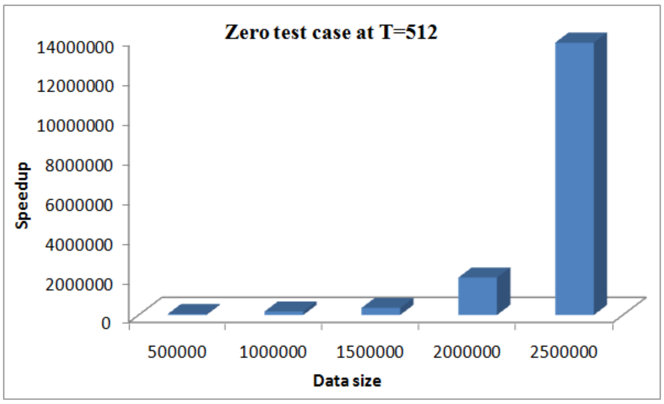


Figure 7.3: Speedup achieved by parallel bubble sort using zero test case

The speedup concludes by the parallel bubble sort over sequential using zero test case is represented in Figure 7.3. The best case of bubble sort occurs, when the data is sorted or unique. In zero test case, one unique value is picked as input. So in this test case, we found major improvement.

The speedup gained by the parallel bubble sort over sequential using staggered test case is described in Figure 7.4. The topmost speedup obtained nearly 7 times at $n = 2500000$.

The speedup acquired by the parallel bubble sort over sequential using bucket test case is demonstrated in Figure 7.5. The maximum speedup obtained nearly 8 times.

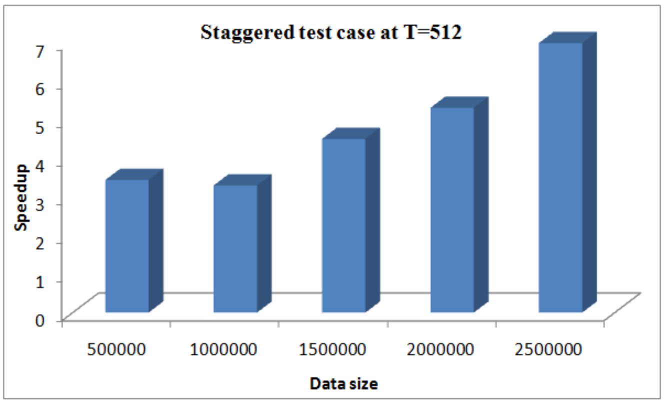


Figure 7.4: Speedup achieved by parallel bubble sort using staggered test case

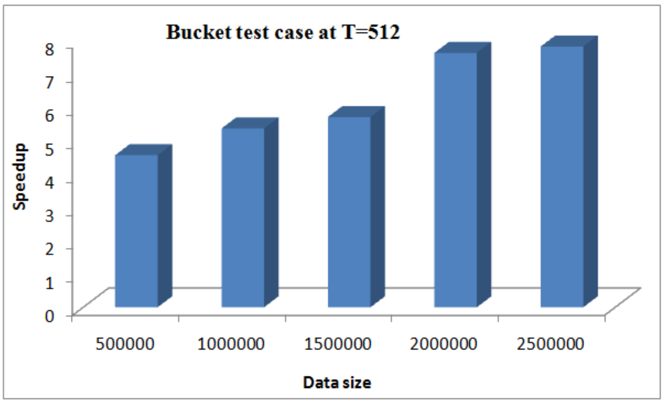


Figure 7.5: Speedup achieved by parallel bubble sort using bucket test case

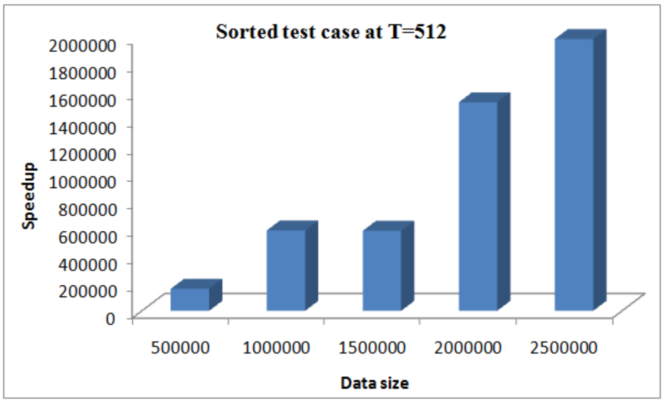


Figure 7.6: Speedup achieved by parallel bubble sort using sorted test case

The Figure 7.6 shows, the speedup acquired by the parallel bubble sort over sequential using sorted test case. As the best case of bubble sort occur in the sorted test case, so in this manner, it achieves maximal speedup among other test cases.

7.4 Execution Time Comparison of Sequential and Parallel Bubble Sort

We have calculated the execution time of sequential and parallel bubble sort using sorting benchmark which is listed in Table 7.2 & 7.3. The execution time of parallel and sequential bubble sort is compared in the Figure 7.7 to 7.12. The X-axis represents the value of 'n' and Y-axis represents the execution time in seconds. The Figure 7.7 to 7.12, has been drawn from using the values of Table 7.2 & 7.3.

The Figure 7.7, represents the execution time comparison of parallel and sequential bubble sort using uniform test case. The maximum improvement in execution time by the parallel bubble sort is 7.29% for the value of $n = 2500000$ and $T = 512$ i.e. the parallel bubble sort is 7.29% more efficient than sequential.

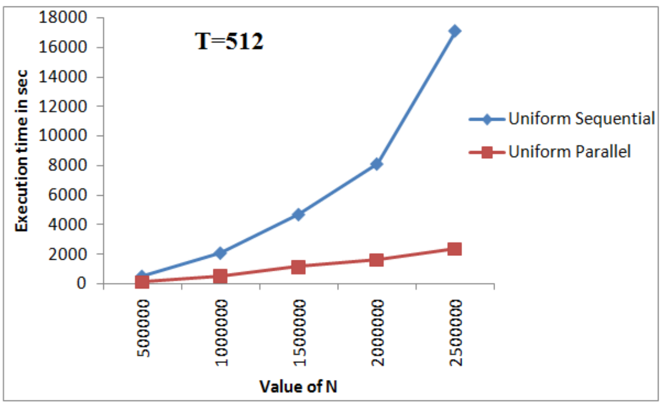


Figure 7.7: Execution time comparison of parallel and sequential bubble sort using uniform test case

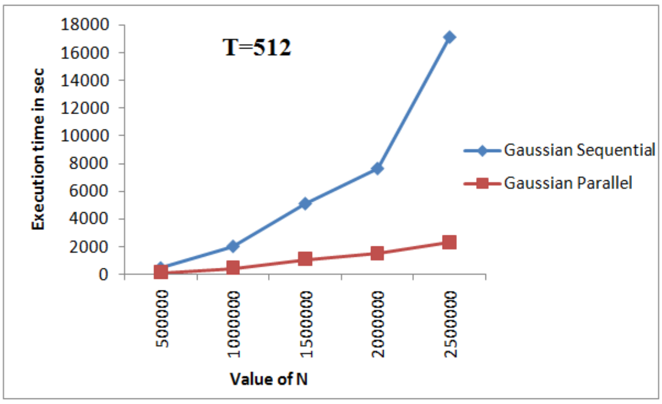


Figure 7.8: Execution time comparison of parallel and sequential bubble sort using gaussian test case

The execution time comparison of parallel and sequential bubble sort using Gaussian test case is represented in Figure 7.8. The maximum progress in execution time is achieved 7.38% at $n = 2500000$ and $T = 512$ i.e. the parallel bubble sort is 7.38% more efficient than sequential.

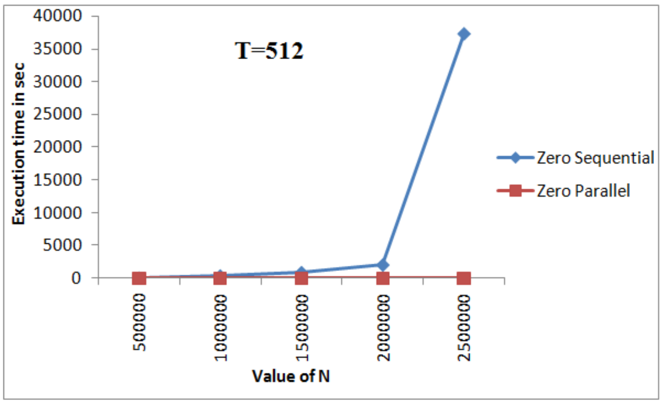


Figure 7.9: Execution time comparison of parallel and sequential bubble sort using zero test case

The execution time comparison of parallel and sequential bubble sort using zero test case is listed in Figure 7.9. As in zero test case only one unique value is used as an input so the parallel bubble sort obtained the $O(1)$ time complexity.

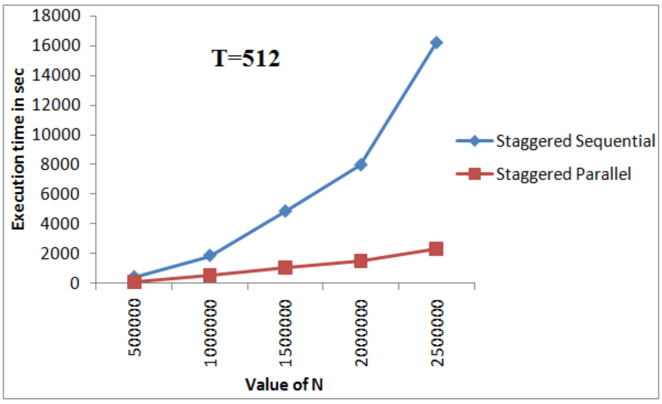


Figure 7.10: Execution time comparison of parallel and sequential bubble sort using staggered test case

The execution time comparison of parallel and sequential bubble sort using staggered test case is listed in Figure 7.10. The maximum progress in execution time is achieved 6.96% at $n = 2500000$ and $T = 512$ i.e. the parallel bubble sort is 6.96% more efficient than sequential.

The execution time comparison of parallel and sequential bubble sort using bucket test case is listed in Figure 7.11. The maximum progress in execution time is achieved 7.08% at $n = 2500000$ and $T = 512$ i.e. the parallel bubble sort is 7.08% more efficient than sequential.

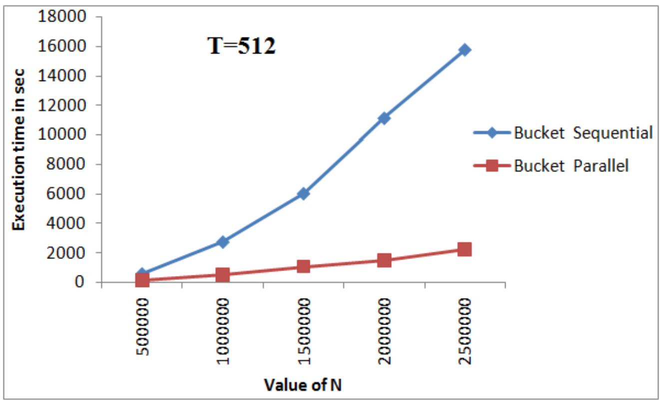


Figure 7.11: Execution time comparison of parallel and sequential bubble sort using bucket test case

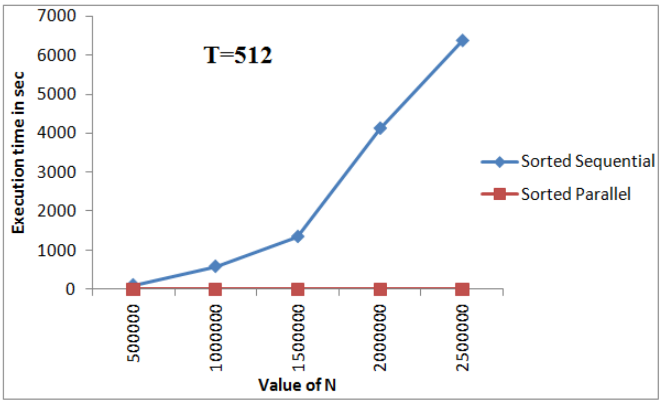


Figure 7.12: Execution time comparison of parallel and sequential bubble sort using sorted test case

The execution time comparison of parallel and sequential bubble sort using sorted test case is listed in Figure 7.12. As the bubble sort occur the best case when the data is already sorted test case. So the parallel bubble sort obtained the $O(1)$ time complexity using sorted test case.

7.5 Conclusion

The parallel bubble sort achieved the $O(1)$ time complexity, when data is not requiring any swapping, i.e. when the data is zero or sorted. The parallel bubble sort achieves 37229 times faster execution time using zero test case and 6375 times faster using sorted test case at $n = 2500000$ and $T = 512$. The best case time complexity of bubble sort is reduced $O(n)$ to $O(1)$. It is because we have executed the bubble sort using GPU computing with CUDA hardware. The testing is done using sorting benchmark. The input value varied from n= 500000 to 2500000 and thread in the multiple of 2 from 1 to 1024.

Chapter 8

Conclusions and Future Scope

8.1 Conclusion

In this thesis, the following algorithms have been tested using sorting benchmark and standard dataset with GPU computing.
1. GPU Merge Sort using CUDA hardware.
2. GPU Quick Sort using CUDA hardware.
3. GPU Count Sort using CUDA hardware.
4. GPU Bubble Sort using CUDA hardware.

In this thesis, we have also tested the various sorting algorithms on a standard dataset. The various algorithms are following.
1. Insertion Sort
2. Selection Sort
3. Bubble Sort
4. Heap Sort
5. Shell Sort
6. Quick Sort
7. Merge Sort
8. Radix Sort

9. Count Sort

The Following algorithms have been proposed.

1. Proposed Modified parallel OETSN algorithm.

2. Library sort algorithm with uniform gap distribution.

3. Library sort algorithm with non-uniform gap distribution.

4. Proposed Hybrid Sort Algorithm.

The performance measures have been done of all the listed algorithms in terms of space and time complexity. In the future, the performance measures can be tested of all the listed algorithms in terms of stability and adaptivity.

8.2 Future Scope

We can further classify other sorting algorithm like quick sort based on the number of elements in bucket which will not only make the working faster of bucket sort but also will reduce the time.

We can still find a gap to use the knowledge about the data to implement the sorting algorithm. Future research will refine the performance of sorting algorithms using GPU architecture and Thrust library.

The parallel version of library sort using CUDA hardware can be designed in future. The GPU LNGD (Library sort using non-uniform gap distribution) can also be designed.

We have used the GPU computing using CUDA hardware having the compute capability 2.1 to test the algorithms. But, if the same algorithms has been used on the hardware having the compute capability 3.0, then it will give an added advantage of unified memory architecture. The performance of GPU algorithms can be enhance by using different CUDA hardware versions and using Thrust Library.

References

[1] Kocher, Gaurav, and Nikita Agrawal, "Analysis and Review of Sorting Algorithms," *International Journal of Scientific Engineering and Research (IJSER)*, vol. 2, No. 3, pp. 81-84, 2014.

[2] Pahuja, & Sanjay, "a practical approach to data structures and algorithms," *Data Structure Book*, 2007.

[3] Salaria, R.S, "Data Structures & Algorithms Using C," *Data Structure Book*, 2004.

[4] Knuth, Donald E, "The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. Pearson Education India," *Data Structure Book*, 2011.

[5] Downey, Rod G., and Michael Ralph Fellows, "Parameterized complexity," *Heidelberg: springer*, vol. 3, 1999.

[6] Elisseeff, Andre, Theodoros Evgeniou, and Massimiliano Pontil, "Stability of randomized learning algorithms," *Journal of Machine Learning Research*, vol. 6, pp. 55-79, 2005.

[7] Estivill-Castro, Vladmir, and Derick Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, No. 4, pp. 441-476, 1992.

[8] Blelloch, Guy E., Leiserson, C. E., Maggs, B. M., et al, "A comparison of sorting algorithms for the connection machine CM-2," *Proceedings of the third*

annual ACM symposium on Parallel algorithms and architectures, pp. 3-16 , 1991.

[9] Marx, Dniel, "Parameterized complexity and approximation algorithms," *The Computer Journal*, vol. 51, No. 1, pp. 60-78, 2008.

[10] Papadimitriou, Christos H, "Computational complexity," *John Wiley and Sons Ltd*, 2003.

[11] Cormen, Thomas H., Leiserson, C. E., Rivest, R. L., & Stein, C, "Introduction to algorithms," *MIT press*, 2009.

[12] Bhalchandra, Parag, Deshmukh, N., Lokhande, S., & Phulari, S, "A comprehensive note on complexity issues in sorting algorithms," *Adv. Comput. Res* , vol. 1, No. 2, pp. 1-9, 2009.

[13] Fix, James D., and Richard E. Ladner, "Sorting by parallel insertion on a one-dimensional subbus array," *IEEE Transactions on Computers*, vol. 47, No. 11, pp. 1267-1281, 1998.

[14] Papadimitriou, Christos H, "Computational complexity," *John Wiley and Sons Ltd*, vol. 47, No. 11, pp. 1267-1281, 1998.

[15] Wegener, Ingo, "Bottom-up-heap sort, a new variant of heap sort beating on average quick sort (if n is not very small)," *Mathematical Foundations of Computer Science Springer Berlin Heidelberg*, vol. 118, pp. 81-98, 1990.

[16] Horowitz, Ellis, and Dinesh Mehta, "Fundamentals of data structures in C++," *Galgotia Publications*, 2006.

[17] Alnihoud, Jehad, and Rami Mansi, "An Enhancement of Major Sorting Algorithms," *Int. Arab J. Inf. Technol*, vol. 7, No. 1, pp. 55-62, 2010.

[18] Dawra, Malika, and Priti Priti, "Proposed Model for Sorting Algorithms," *International Journal on Computer Science and Engineering*, vol. 4, No. 6, 2012.

[19] Nyhoff, L., "Introduction to Data Structures," *Nyhoff Publishers, Amsterdam*, 2005.

[20] Blelloch, Guy E., et al, "A comparison of sorting algorithms for the connection machine CM-2," *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pp. 3-16, 1991.

[21] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar, "Gpgpu processing in cuda architecture," *arXiv preprint arXiv:1202.4347*, 2012.

[22] Jason Sanders and Edward Kandrot, "CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents," *Addison-Wesley Professional*, 2010.

[23] Massimiliano Fatica, P LeGresley, I Buck, J Stone, J Phillips, S Morton, and P Micikevicius, "High performance computing with cuda," *SC08*, 2008.

[24] Makoto Matsumoto and Takuji Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, No. 1, pp. 330, 1998.

[25] Alexandru Pirjan, "Improving software performance in the compute unified device architecture," *Informatica Economica*, vol. 14, No. 4, 2010.

[26] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips, "A comprehensive performance comparison of cuda and opencl," *IEEE International Conference on Parallel Processing (ICPP)*, pp. 216-225, 2011.

[27] Jaumin Ajdari, Bujar Raufi, Xhemal Zenuni, and Florije Ismaili, "A version of parallel odd-even sorting algorithm implemented in cuda paradigm," *International Journal of Computer Science Issues (IJCSI)*, vol. 12, No. 3, 2015.

[28] Luebke, David, "CUDA: Scalable parallel programming for high-performance scientific computing," *5th IEEE International Symposium on. IEEE Biomedical Imaging: From Nano to Macro*, 2008.

[29] Leischner, Nikolaj, Vitaly Osipov, and Peter Sanders, "Gpu-sample Sort," *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1-10, 2010.

[30] Daniel Cederman and Philippas Tsigas, "Gpu-quicksort: A practical quick-sort algorithm for graphics processors," *Journal of Experimental Algorithmics (JEA)*, vol. 14, No. 4, 2009.

[31] 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.

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

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

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

[35] Matloff, Norm, "Programming on parallel machines," *University of California*, 2011.

[36] Miic, Marko J., and Milo V.Tomaevic, "Data sorting using graphics processing units," *Telfor Journal*, vol. 4, No. 4, 2012.

[37] Zurek, Dominik, et al, "The comparison of parallel sorting algorithms implemented on different hardware platforms," *Computer Science*, vol. 14, No. 4, 2013.

[38] 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.

[39] 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.

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

[41] C. E. Leiserson, R. L. Rivest, C. Stein, T. H. Cormen, "Introduction to algorithms," *MIT press*, 2001.

[42] J. D. Fix, R. E. Ladner, "Sorting by parallel insertion on a one-dimensional subbus array," *IEEE Transactions on Computers*, vol. 47, No. 11, pp. 12671281, 1998.

[43] R. L. Wainwright, "A class of sorting algorithms based on quicksort," *Communications of the ACM*, vol. 28, No. 4, pp. 396402, 1985.

[44] C. D. Thompson, "Area-time complexity for vlsi," *Proceedings of the eleventh annual ACM symposium on Theory of computing, ACM*, pp. 8188, 1979.

[45] V. Estivill-Castro, D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, No. 4, pp. 441476, 1992.

[46] E. Horowitz, S. Sahni,, "Fundamentals of data structures," *Pitman*, 1983.

[47] V. Estivill-Castro, D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, No. 4, pp. 441476, 1992.

[48] D. L. Shell, "A high-speed sorting procedure," *Communications of the ACM*, vol. 2, No. 7, pp. 3032, 1959.

[49] M. A. Bender, M. Farach-Colton, M. A, "Mosteiro, Insertion sort is o(nlogn)," *Theory of Computing Systems*, vol. 39, No. 3, pp. 391397, 2006.

[50] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, L. Rauchwerger, "A framework for adaptive algorithm selection in stapl," *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 277288, 2005.

[51] W. Janko, "A list insertion sort for keys with arbitrary key distribution," *ACM Transactions on Mathematical Software (TOMS)*, vol. 2, No. 2, pp. 143153, 1976.

[52] A. Itai, A. G. Konheim, M. Rodeh, "A sparse table implementation of priority queues,"*Springer*, 1981.

[53] M. A. Bender, Z. Duan, J. Iacono, J. Wu, "A locality-preserving cache-oblivious dynamic dictionary," *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics*, pp. 2938, 2002.

[54] G. S. Brodal, R. Fagerberg, R. Jacob, "Cache oblivious search trees via binary trees of small height," *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics*, vol. 2, No. 2, pp. 3948, 2002.

[55] G. Franceschini, V. Geffert, "An in-place sorting with o(nlogn) comparisons and o(n) moves," *Journal of the ACM (JACM)*, vol. 52, No. 4, pp. 515537, 2005.

[56] Creeger M, "Multicore CPUs for the masses," *ACMQueue*, vol. 3, No. 7, pp. 64-65, 2005.

[57] HackerH,TrinitisC, et al, "Considering GPGPU for HPC centers: Is it worth the effort? In Facing the Multicore-Challenge," *Lecture Notes in Computer Science Springer*, vol. 63, No. 10, 2010.

[58] NickollsJ, Dally W J, "The GPU computing era," *IEEE Micro*, vol. 30, No. 2, pp. 5669, 2010.

[59] Zhang Y, Owens J D, “A quantitative performance analysis model for GPU architectures,” *IEEE 17th International Symposium on High Performance Computer Architecture, San Antonio, TX*, pp. 38293, 2011.

[60] Garland M, “Parallel computing with CUDA,” *IEEESymposium on Parallel & Distributed Processing IPDPS, Atlanta, GA*, pp. 1-10, 2010.

[61] KindratenkoV.V, EnosJ, ShiG, et al, “GPU clusters for high-performance computing,” *In IEEE International Conference on Cluster Computing Workshops, CLUSTER*, pp. 18, 2009.

[62] Faujdar N,Ghrera SP, “Analysis and Testing of Sorting Algorithms on a Standard Dataset,” *IEEE Fifth International Conference on Communication Systems and Network Technologies (CSNT),Gwalior, India*, pp. 962-967, 2015.

[63] Faujdar N, Ghrera S P, “Performance Evaluation of Merge and Quick Sort using GPU Computing with CUDA,” *International Journal of Applied Engineering Research (IJAER)*, vol. 10, No. 18, pp. 39315-39319, 2015.

[64] Joshi R, Panwar G S, Pathak P, “Analysis of Non-Comparison Based Sorting Algorithms: A Review,” *International journal of emerging research in management and technology*, vol. 2, No. 12, pp. 61-65, 2013.

[65] MishraA D, Garg D, “Selection of Best Sorting Algorithm,” *International Journal of Intelligent Information Processing*, vol. 2, No. 2, pp. 363-368, 2008.

[66] Keshav B, Kots A, “Implementing and Analyzing an Efficient Version of Counting Sort (E-Counting Sort),” *International Journal of Computer Applications*, vol. 98, No. 9, pp. 1-2, 2014.

[67] Svenningsson, Josef David, et al, “Counting and occurrence sort for GPUs using an embedded language,” *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, ACM*, pp. 37-46, 2013.

[68] Sun, Weidong, and Zongmin Ma, "Count sort for gpu computing," *IEEE 15th International Conference Parallel and Distributed Systems (ICPADS)*, pp. 919-924, 2009.

[69] Chitra E. Vigneswaran T, "An Efficient Low Power and High Speed Distributed Arithmetic Design for FIR Filter," *Indian Journal of Science and Technology*, vol. 9, No. 4, 2016.

[70] Dowd, Martin, et al, "The periodic balanced sorting network," *Journal of the ACM (JACM)*, vol. 36, No. 4, pp. 738-757, 1989.

[71] Ushijima, Mizue, and Akihiro Fujiwara, "Sorting algorithms based on the odd-even transposition sort and the shearsort with DNA strands," *In FCS*, pp. 52-58, 2005.

[72] Quinn, Michael J, "Parallel computing: theory and practice," *McGraw-Hill, Inc*, 1994.

[73] Farmahini-Farahani, Amin, et al, "Modular design of high through put, low-latency sorting units," *IEEE Transactions on Computers*, vol. 62, No. 7, pp. 1389-1402, 2013.

[74] Grama, Ananth, Anshul Gupta, and George Karypis, "Introduction to parallel computing: design and analysis of algorithms," *Redwood City, CA: Benjamin/Cummings Publishing Company*, 1994.

[75] Brodal GS, Fagerberg R, Jacob R, "Cache oblivious search trees via binary trees of small height," *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics*, pp. 3948, 2002.

[76] Pardo LT, "Stable sorting and merging with optimal space and time bounds," *SIAM Journal on Computing*, vol. 6, No. 2, pp. 351372, 1977.

[77] Neetu Faujdar SPG, "A detailed experimental analysis of library sort algorithm," *12th IEEE India International Conference (INDICON)*, 2015.

[78] Zhao, Zhongxiao, and Chen Min, “An Innovative Bucket Sorting Algorithm Based on Probability Distribution,” *World Congress on Computer Science and Information Engineering WRI*, vol. 7, 2009.

[79] Canaan, C., M. S. Garai, and M. Daya, “Popular sorting algorithms,” *World Applied Programming*, vol. 1, No. 1, pp. 42-50, 2011.

[80] Chlebus, Bogdan S, “A parallel bucket sort,” *Information processing letters*, vol. 27, No. 2, pp. 57-61, 1988.

[81] Greb, Alexander, and Gabriel Zachmann, “GPU-ABiSort: Optimal parallel sorting on stream architectures,” *IEEE 20th International on Parallel and Distributed Processing Symposium*, 2006.

[82] Inoue, Hiroshi, et al, “AA-sort: A new parallel sorting algorithm for multi-core SIMD processors,” *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. IEEE Computer Society*, 2007.

[83] 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.

[84] Cederman, Daniel, and Philippas Tsigas, “A practical quicksort algorithm for graphics processors,” *Algorithms-ESA, Springer Berlin Heidelberg*, pp. 246-258, 2008.

[85] Rozen, T., Krzysztof Boryczko, and Witold Alda, “GPU bucket sort algorithm with applications to nearest-neighbour search,” *Algorithms-ESA, Springer Berlin Heidelberg*, 2008.

[86] Baraglia, Ranieri, et al, “Sorting using bitonic network with CUDA,” *7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR), Boston, USA*, 2009.

[87] Kukunas, Jim, and James Devine, "GPGPU Parallel Merge Sort Algorithm," 2009.

[88] Oat, Christopher, Joshua Barczak, and Jeremy Shopf, "Efficient spatial binning on the GPU," *SIGGRAPH Asia*, 2008.

[89] Huang, Bonan, Jinlan Gao, and Xiaoming Li, "An empirically optimized radix sort for gpu," *IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2009.

[90] Ye, Xiaochun, et al, "High performance comparison-based sorting algorithm on many-core GPUs," *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[91] Peters, Hagen, Ole Schulz-Hildebrandt, and Norbert Luttenberger, "Fast in-place sorting with cuda based on bitonic sort," *Parallel Processing and Applied Mathematics. Springer Berlin Heidelberg*, pp. 403-410, 2010.

[92] Peters, Hagen, Ole Schulz-Hildebrandt, and Norbert Luttenberger, "Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead," *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[93] Satish, Nadathur, et al, "Fast sort on cpus, gpus and intel mic architectures," *Intel Labs*, pp. 77-80, 2010.

[94] Helluy, Philippe, "A portable implementation of the radix sort algorithm in OpenCL," *http://hal.archives-ouvertes.fr/hal-00596730/fr/, Technical report*, 2011.

[95] Krueger, Jens, et al, "Applicability of GPU Computing for Efficient Merge in In-Memory Databases," *ADMS@ VLDB*, 2011.

[96] Misic, Marko J., and Milo V. Tomasevic, "Data sorting using graphics processing units," *19th. IEEE Telecommunications Forum (TELFOR)*, 2011.

[97] Peters, Hagen, Ole Schulz-Hildebrandt, and Norbert Luttenberger, "A novel sorting algorithm for many-core architectures based on adaptive bitonic sort," *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*, 2012.

[98] Jan, Bilal, et al, "Fast parallel sorting algorithms on GPUs," *International Journal of Distributed and Parallel Systems*, pp. 107-118, 2012.

[99] Munavalli, Seema M, "Efficient Algorithms for Sorting on GPUs," *bealto.com/gpu-sorting.html*, 2012.

[100] Thouti, Krishnahari, and S. R. Sathe, "An OpenCL Method of Parallel Sorting Algorithms for GPU Architecture," *International Journal of Experimental Algorithms (IJEA)*, vol. 3, No. 1, pp. 1-8, 2012.

[101] Panwar, Mukul, Monu Kumar, and Sanjay Bhargava, "GPU Matrix Sort (An Efficient Implementation of Merge Sort)," *International Journal of Computer Applications*, vol. 89, No. 18, pp. 9-11, 2014.

[102] Arturo Garcia, Jose Omar Alvizo Flores, Ulises Olivares Pinto, Felix Ramos, "Fast Data Parallel Radix Sort Implementation in DirectX 11 Compute Shader to Accelerate Ray Tracing Algorithms)," *EURASIA GRAPHICS: International Conference on Computer Graphics, Animation and Gaming Technologies*, pp. 27-36, 2014.

[103] Gluck, Joshua, "Fast GPGPU Based Quadtree Construction)," *Dep. C.Sc. Carnegie Mellon University,*, 2014.

[104] Ye, Yin, et al, "GPUMemSort: A High Performance Graphics Coprocessors Sorting Algorithm for Large Scale In-Memory Data)," *Journal on Computing (JoC)*, vol. 1, No. 2, 2014.

[105] Polok, Lukas, Viorela Ila, and Pavel Smrz, "Fast radix sort for sparse linear algebra on GPU)," *Proceedings of the High Performance Computing Symposium. Society for Computer Simulation International*, 2014.

[106] Mu, Qi, Liqing Cui, and Yufei Song, "The implementation and optimization of Bitonic sort algorithm based on CUDA," *arXiv preprint arXiv:1506.01446* , 2015.

[107] Xiao, Jun, Hao Chen, and Jianhua Sun, "High Performance Approximate Sort Algorithm Using GPUs," *Advances in Computer Science Research*, 2015.

[108] Ajdari, Jaumin, et al, "A Version of Parallel Odd-Even Sorting Algorithm Implemented in CUDA Paradigm," *International Journal of Computer Science Issues (IJCSI)* , vol. 12, No. 3, 2015.

Publications Based on PhD Work

Journals:

1. Neetu Faujdar, S.P. Ghrera. "Performance Evaluation of Merge and Quick Sort using CUDA," *International Journal of Applied Engineering Research (IJAER)*, vol. 10, Issue 18, pp.39315-39319, 2015. [**Scopus, EBSCOhost etc**]**(SJR: 0.113)(Pu blished)**

2. Neetu Faujdar, S.P. Ghrera. "Performance Evaluation of Count Sort with GPU Computing using CUDA," *Indian Journal of Science & Technology*, vol. 9(15), DOI: 10.17485/ijst/2016/v9i1 5/80080, 2016. [**Scopus, EBSCOhost etc**]**(IF: 1.05)(Published)**

3. Neetu Faujdar, S.P. Ghrera. "Modified Level of Parallel Odd-Even Transposition Sorting Network (OETSN) with GPU Computing using CUDA," *Pertanika Journal of Science & Technology (JST)*, vol. 24 (2), pp. 331 350, 2016. [**Scopus, DOAJ, etc**]**(IF: 0.013, H-Index 2)(Published)**

4. Neetu Faujdar, S.P. Ghrera. "Performance Analysis of Parallel Sorting Algorithms using GPU Computing," *International Journal of Computer Applications*, vol. 2, pp. 5-11, September 2016.[**(DOAJ, Google Sch olar etc**]**(IF: 3.12)(Published)**

5. Neetu Faujdar, S.P. Ghrera. "An Efficient Bucket Sort using Hybrid Algorithm," *International Journal of Computer Science*

and Information Security (IJCSIS), 2016. **[ESCI Indexed] (Accepted))(IF: 0.519)**

6. Neetu Faujdar, S.P. Ghrera. "Library Sort Algorithm with Non-Uniform Gap Distribution," *Pertanika Journal of Science & Technology (JST).* **[Scopus, DOAJ, etc](IF: 0.013, H-Index 2)(Major Revision Submitted)**

Conferences:

1. Neetu Faujdar, S.P. Ghrera. "Analysis and Testing of Sorting Algorithms on a Standard Dataset," *IEEE Fifth International Conference on Communication System and Network Technologies (CSNT)*, pp. 962 967, DOI. 10.1109/CSNT.2015.98, 2015. **(Published)**

2. Neetu Faujdar, SP Ghrera. "A Detailed Experimental Analysis of Library Sort Algorithm," *12^{th} IEEE India International Conference (INDICON)*, pp. 1-6, DOI. 10.1109/INDICON.2015.7443165, 2015. **(Scopus Indexed)(Published)**

3. Neetu Faujdar, SP Ghrera. "A Practical Approach of GPU Bubble Sort with CUDA," *IEEE 7th International Conference Confluence*, 2017. **[Scopus Indexed](Accepted)**

4. Neetu Faujdar, SP Ghrera. "The Detailed Experimental Analysis of Bucket Sort," *IEEE 7th International Conference Confluence*, 2017. **[Scopus Indexed](Accepted)**

5. Neetu Faujdar, SP Ghrera. "A Roadmap of Parallel Sorting Algorithms using GPU Computing," *IEEE International Conference on Computation, Communication, and Automation*, 2017. [**Scopus Indexed**]**(Communicated)**