Pertanika J. Sci. & Technol. 24 (2): 331 - 350 (2016)

SCIENCE & TECHNOLOGY

Journal homepage: http://www.pertanika.upm.edu.my/

Modified Levels of Parallel Odd-Even Transposition Sorting Network (OETSN) with GPU Computing using CUDA

Neetu Faujdar* and SP Ghrera

Jaypee University of Information Technology Waknaghat, P.O. Waknaghat, The Kandaghat, Distt. Solan, India

ABSTRACT

Sorting huge data requires an enormous amount of time. The time needed for this task can be minimised using parallel processing devices like GPU. The odd-even transposition sorting network algorithm is based on the idea that each level uses an equal number of comparators to arrange data. The existing parallel OETSN algorithm compares the elements in each phase for any type of test case. If the elements are not in the increasing order, then they are swapped. In this way, the algorithm takes the same time for sorting and for unique test cases. In this paper, we propose an algorithm that is the modified version of the existing OETSN algorithm. Our approach reduces the number of levels in the OETSN based on the nature of the data. Time complexity is also reduced from $O(n)$ to $O(1)$ for sorted and zero test cases. The proposed algorithm is tested for six types of test case, which are uniform, Gaussian, zero, bucket, staggered and sorted. The comparison with existing techniques is also presented in this paper. After evaluation, the proposed modified version of OETSN is found to be more efficient in two types of test case i.e. sorted and zero test cases. GPU computing using CUDA hardware is used to test the algorithms. The speedup achieved by the parallel OETSN algorithm over sequential OETSN is also computed. The proposed approach achieves an improvement in execution time that is 981661.6 times faster in the sorted test case and 904620.7 times faster in the zero test case using 2500000 elements and 1024 threads in comparison to the existing parallel OETSN.

Keywords: Sorting, GPU computing, CUDA, comparators, OETSN

INTRODUCTION

Nowadays, sorting is in big demand. There are many sorting algorithms available to arrange data (Ye et al., 2014). In computer science, sorting network algorithms also exist to arrange data over the network. In sorting networks, comparators (Amin et al., 2013) are

Article history:
Received: 8 September 2015
Accepted: 17 February 2016

E-mail addresses:
neetu.faujdar@mail.juit.ac.in, neetu.faujdar@gmail.com (Neetu Faujdar),
sp.ghrera@juit.ac.in, spghrera@rediffmail.com (SP Ghrera)
*Corresponding Author

ISSN: 0128-7680

used to compare and exchange data. Compare-exchange operation is used in sorting networks (Martinet et al., 1989). There are two types of comparators: 1) Increasing (low to high) and 2) Decreasing (high to low) comparator. These comparators (Amin et al., 2013) are shown in Figure 1.

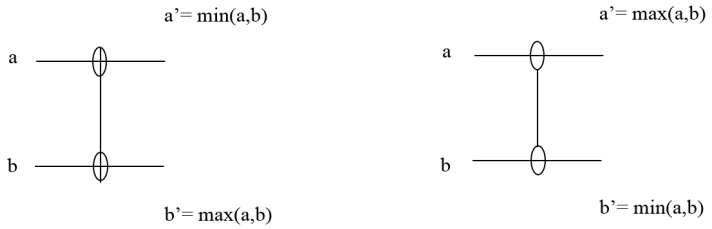


Figure 1. (a) Increasing comparator (b) Decreasing comparator.

The odd-even transposition sorting network (OETSN) (Perrie et al., 1999; Behzad et al., 2010) algorithm is designed for network models. The comparators are used to rearrange the numbers in network models. In the odd-even transposition sorting network, an increasing comparator is used to compare and exchange data. The OETSN algorithm performs $n/2$ iteration. Each iteration has two phases: 1) Odd-even exchange and 2) Even-odd exchange. The concept of OETSN is explained with the help of an example shown in Figure 2 (Ushijima & Fugiwara, 2005).

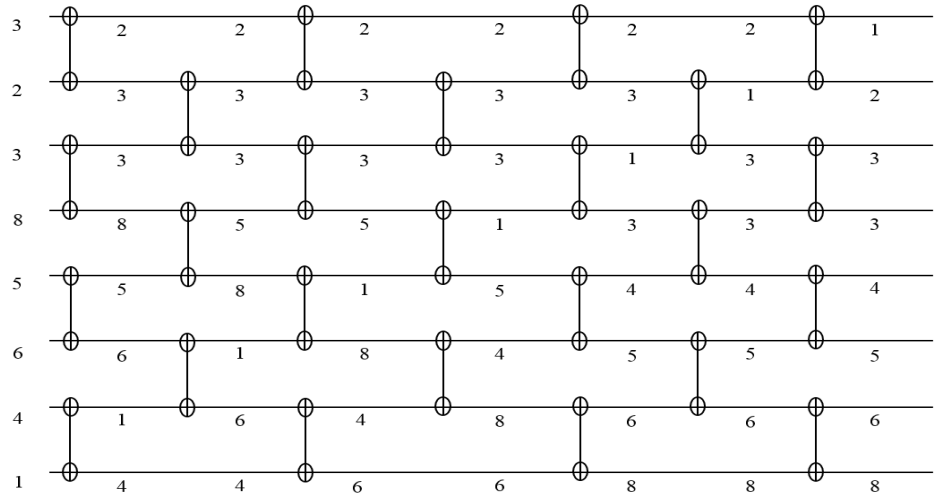


Figure 2. Example of OETS network.

The OETSN algorithm operates alternatively for odd and even phases. In the odd phase, the odd position number of items is compared with the adjacent element. Similarly in the even phase, the even position number of items is compared with the adjacent element. If the items are not in increasing order, swapping is performed as shown in Figure 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. There are a total of 'n' phases; thus, the sequential complexity of OETSN is $O(n^2)$. In this paper, we parallelised the OETSN algorithm using CUDA with C language. We tested the OETSN using a benchmark of sorting. We used the following distributions for benchmark to compare the performance of the OETSN algorithm. We tested the OETSN algorithm in six types of test case, which are uniform, sorted, zero, bucket, Gaussian and staggered. The parallel time complexity of OETSN is $O(n)$. In this work, we introduced a modified parallel odd-even transposition sorting algorithm, which is more efficient in comparison to parallel OETSN on the zero and sorted test cases.

The literature review on GPU is presented in Section 2. The problem statement is given in Section 3. A review of the literature on parallel OETSN is provided in Section 4. The proposed algorithm is described in Section 5. The benchmark of sorting is described in Section 6. Section 7 describes the hardware configuration. Experimental evaluation of parallel OETSN and modified parallel OETSN are given in Sections 8 and 9. In Section 10, we make our conclusions and suggest future work in this area.

RELATED WORK

Sorting is a common problem in computer science. A huge number of algorithms have been suggested to find an efficient solution to solve the problem. In this work, we focused on GPU technology using the CUDA paradigm. In this section, we briefly review the related work by other researchers on GPU sorting algorithms using CUDA.

A version of the parallel odd-even sorting algorithm implemented using CUDA was presented by Ajdari et al. in 2015. The design of parallel radix sort and merge sort using CUDA was given by Nadathur et al. (2009). They achieved higher performance in their experiments.

Shifu et al. (2009) proposed a sorting algorithm that is a combination of the bucket sort and internal bitonic sort types. This algorithm achieved acceleration of many times over the STL Quicksort implementation. They also showed that their implementation had higher performance than the GPU quick and GPU radix sort.

Daniel and Philippas in (Daniel et al., 2009) proposed a parallel Quicksort algorithm designed to take advantage of the high bandwidth of GPUs by minimising the amount of bookkeeping and inter-thread synchronisation needed. They showed that their GPU-Quicksort implementation performed better than the fastest known sorting implementations for GPU, such as radix and bitonic sort.

Performance Evaluation of Merge and QuickSort using GPU Computing with CUDA was presented by Neetu et al. (2015). In this paper the authors achieved better experimental results using GPU technology.

Greb et al. presented parallel sorting based on stream processing architecture. The proposed sorting was based on adaptive bitonic sorting. The input of 'n' requires 'p' stream processor to sort. The optimal time complexity of the proposed approach achieved $O(n\log n)/p)$. The proposed approach was more competitive than sequential sorting from the theoretical as well as a practical perspective viewpoint. The proposed algorithm was faster than sequential sorting as well as previous non-optimal sorting approaches on the GPU. The proposed algorithm was specially designed for practicability on modern GPU, so the name GPU-ABiSort was used for it (Grab et al., 2006).

Peters et al. presented Batcher's bitonic sorting network using CUDA hardware with GPUs. The arbitrary numbers were taken as input and assigned compare-exchange operation to threads using the adapted bitonic sort. The proposed algorithm greatly increased the performance of implementation (Peters et al., 2010). Jan et al. presented the analysis of three widely used parallel-sorting algorithms. The algorithms were odd-even sort, rank sort and bitonic sort. The comparative analysis was performed in terms of sorting rate, sorting time and speedup on CPU and different GPU architecture. The author also implemented the parallel algorithm: min-max butterfly network. The min-max butterfly network sorting was used to find minimum and maximum numbers in huge data sets. The purpose of all algorithm implementation was used to exploit the data parallelism model in order to achieve high performance on available GPU using the OpenCL specification. The results showed minimum speedup 19x of bitonic sort against the odd-even sort. The implementation results of the full-butterfly network were relatively better than the three sorting techniques: bitonic, odd-even and rank sort. The author achieved the high speedup of NVIDIA quadro 6000 GPU for min-max butterfly network, reaching much lower sorting for high data (Jan et al., 2012).

Ajdari et al. described the modification of the odd-even sort. The modification of the algorithm consisted of the ability to work with the blocks of elements instead of working with individual elements. The modification was done using CUDA technology. The experimental analysis of odd-even sort was done in both theoretical and experimental with its parallel implementation. The results showed that sorting of integers in a CUDA environment was much faster (Ajdari et al., 2015).

PROBLEM STATEMENT

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

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 used the the bubble sort technique. If the data are sorted and unique, bubble sorting requires only one pass and then terminates the programme. In our approach we used this technique, and are able to reduce the number of levels in the network and the time complexity for sorted and zero test cases.

PARALLEL OETSN ALGORITHM

It is easy to parallelise OETSN algorithm (Grama et al., 1994). Compare-exchange operation was performed simultaneously on each pair of elements. There are two cases: 1) when $n=p$ where 'p' is the number of processing elements and 'n' is the number of elements to be sorted. In both the phases compare-exchange operation is performed on the right adjacent element. This required time $O(1)$. A total of 'n' phases is performed. So the parallel run time of this formulation is $O(n)$. 2) When $p<n$ or $p>n$ initially, each process is assigned a block of n/p

elements, which it sorted internally in $O((n/p)(n/p))$ time. After these processes, 'p' phases ($p/2$ odd and $p/2$ even) are executed. During each phase $O(n)$ comparisons are performed and time $O(n)$ is spent in communication. We did not use any local sort before the odd-even phase. The parallel run time of this formulation is shown in equation (1).

$$T_p = O\left(\frac{n^2}{p^2}\right) + O(n) + O(n) \quad [1]$$

Since the sequential complexity of OETSN is $O(n^2)$, the speedup (S) of this formulation is shown in equation (2).

$$S = \frac{O\left(n^2\right)}{O\left(\frac{n^2}{p^2}\right) + O\left(n\right)} \quad [2]$$

PROPOSED MODIFIED PARALLEL OETSN ALGORITHM

ALGORITHM 1: Proposed Modified OETSN Algorithm
Input: Unsorted List A, Number of threads
Output: Sorted List A
for i= 1 to N/2 **do**
    Initialise the P array to zero for GPU *; /* N/2 number of passes */*
    Odd Phase(A, P, N) ; */* comparison of odd positions of array */*
    Even Phase(A, P, N) ; */* comparison of even positions of array */*
if (i==0 OR i== N/4 OR i==N/8 OR i==N/16)*;/* check whether list has been sorted, then not performing any swaps at 0,1/8,1/4,1/2 passes */*
    then
        Evalute (P) ; */* sum the number of swap of various threads */*
        Read sum from GPU ;
        if sum == 0 **then** */* number of swap has been done */*
        break: */* terminate if no swap performed */*
    end
    end
end

The proposed sorting algorithm is inspired by the traditional bubble sorting algorithm. In the traditional bubble sorting algorithm, we compared the adjacent elements. If the elements are sorted, no swapping is done. Traditional bubble sorting takes 'n' passes to complete the sorting in the best case. In the modified version of bubble sorting, we used the flag variable to keep the track of swapping. If the variable highlighted swapping, the next pass is executed. The same concept is applied to the odd-even transposition sorting algorithm using GPU. Let the number of elements to be sorted be 'N' and 'T' is the number of threads. The number of threads

is restricted to a maximum limit of the hardware. According to the hardware configuration, our hardware support is T=1024 threads in one block. In our algorithm, we have used three functions.

1. Odd ()
2. Even ()
3. Evaluate ()

The odd () function can generate a maximum number of threads of T=1024 while the blocks are N/1024, where N is the data element. Similarly, the even () function has the same configuration as the odd () function. The evaluate () function does not execute with every iteration. Its data element is based on the number of threads and varies according to T. This evaluate () function is evaluated in sequential manner in a single block and uses single data. The reason for keeping it sequential is that the thread value is limited to T=1024.

In GPU instead of using a single variable array we used 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. The odd-even pass is executed. If there is no swapping then the sum of 'P' was zero and we got the sorted array. This gave an added advantage as the sorted and unique test cases did not need to be executed in the code on the GPU unnecessarily as is the case when the data are sorted or unique. On the other hand, a slight increase in the execution time for the uniform, staggered, bucket and Gaussian test cases was noted. This made them unable to take advantage of the above proposed approach. We observed the same with N/2, N/4 and N/8 of the data.

We used the GPU NVDIA GeForce GTX 460 with compute capability 2.1 but the new version of GPU cards come with the compute capability 3.0, which has unified memory for GPU and CPU, and can therefore, further enhance the performance of the suggested algorithm. Future enhancements may be possible to get a further speedup. For instance, we may use the scan function to speed up the sum up. The functionality of the proposed algorithm is described through the flowchart shown in Figure 3. The green-coloured 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.

Sorting Benchmark

We have tested the sequential, parallel and proposed modified parallel OETSN algorithms on six types of test case (uniform, sorted, zero, bucket, Gaussian and staggered) (Matsumoto & Nishimura, 1998; Daniel et al, 2009; Leischner, Sanders , 2010). We varied the data from 1000 to 2500000 and the thread in multiples of two from 1 to 1024.

1. Uniform test case: In this test case values are picked randomly from 0 to 2^{31}.
2. Gaussian test case: In this test case the distribution of data is created by taking the average of four randomly picked values from the uniform distribution.
3. Zero test case: In this test case a constant value is used.

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 are random numbers in $[0, 2^{31}/p-1]$, the second n/p^2 elements in $[2^{31}/p, 2^{32}/p- 1]$, 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: In this test case sorted uniformly distributed values are taken.

Hardware Configuration

We ran the algorithms on a Windows 7 32-bit operating system Intel® core™ I3 processor 530@ 2.93 GHz machine. The system has a GeForce GTX 460 graphic processor with (7 multiprocessors X (48) CUDA cores\MP) = 336 CUDA cores. There are a maximum of 1536 threads per multiprocessor and 1024 threads per block. A system having the CUDA runtime version is 6.0. The total amount of global memory of 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. The system has a total number of registers available per block of 32768 and its warp size is 32. The maximum size of each dimension of a block is 1024 x 1024 x 64 and the maximum size of each dimension of a grid is 65535 x 65535 x 65535.

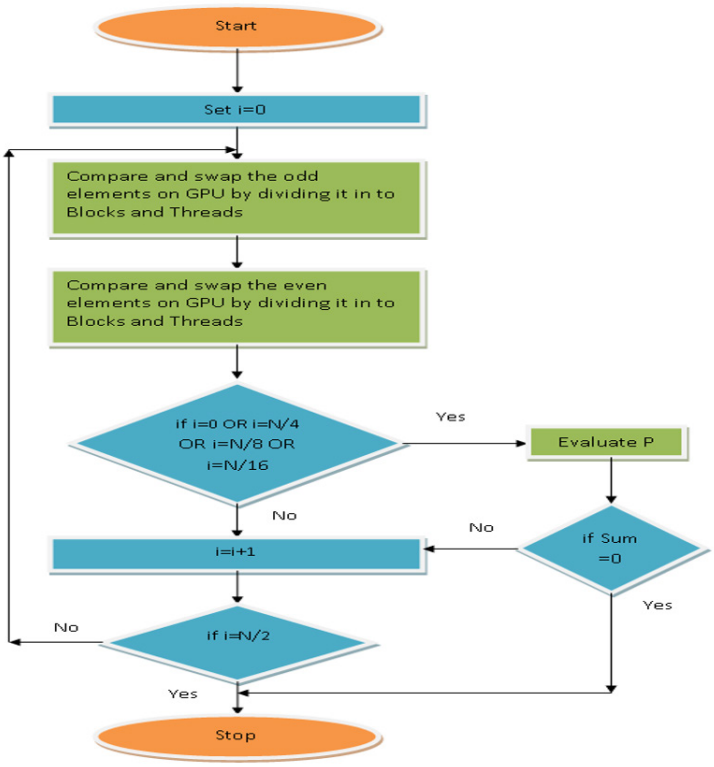


Figure 3. Flowchart for the proposed modified parallel OETSN.

Experimental Evaluation of Sequential and Parallel OETSN Algorithm

Sorting benchmark is used for testing the algorithms. We have tested the sequential and parallel OETSN algorithms on six types of test case using GPU computing that used CUDA hardware. Table 1 shows the execution time in seconds of the sequential OETSN algorithm. '*N*' is the size of the data used for the particular cases here for the performance analysis of the algorithm. The value of '*N*' varied from 1000 to 2500000. Table 2 shows the execution time in seconds of the parallel OETSN algorithm using different types of test cases. The size of the data is denoted by '*N*'. The number of threads is denoted by '*T*'. The values of '*T*' varied from 1 to the maximum of 1024. The threads increased in the power of 2. The CUDA hardware version 2.1 has a total of 1024 threads per block so the maximum value of thread is selected as 1024. In Table 1, the sequential execution time is shown for the six types of test case. Table 1 shows, that the zero test case has less execution time in comparison to the others. It has less execution time for all the values of '*N*'. The sorted test case has less execution time in comparison to the bucket, staggered, uniform and Gaussian tests for all the values of '*N*'. The remaining test cases has nearly equal execution time as shown in Table 1. This is because in the zero and sorted test cases, the comparison is performed to the adjacent element and swapping is not required for both. Comparison and swapping are performed in the remaining test cases.

Table 1
Execution Time in Seconds of Sequential OETSN Using Different Types of Test Case

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

Next, we have evaluated the speedup achieved by the parallel OETSN over the sequential OETSN. Speedup measures the performance gain achieved by parallelising a given application over sequential application. In Tables 1 and 2, we have evaluated the execution time in seconds of sequential and parallel OETSN. Using equation (2) and the results from Table 1 and 2, the speedup is calculated. The speedup results are described in Table 3. From Table 2 and 3, it can be observed that the execution time is minimum when the number of threads is 512. The speedup is increased by eight times more than the sequential code when T=512. The performance of the algorithm got degraded at T=1024. The reason behind this is that the data we took is not evenly divided over the threads. So, some of the threads are executed ideally and degraded the overall performance of the algorithm. The speedup for all the six mentioned test cases is

shown in Figures 4 to 9. The *X*-axis represent the number of threads, the *Y*-axis represent the speedup achieved by the parallel OETSN and the *Z*-axis represent the size of the dataset.

Table 2
Execution Time in Seconds of Parallel OETSN Using Different Types of Test Case

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	2.41	1.353	1.094	1.09	1.08	1.235
	Gaussian	43.452	31.433	16.43	8.588	4.492	2.406	1.351	1.093	1.091	1.083	1.235
	Zero	43.328	31.338	15.931	8.02	4.089	2.16	1.204	0.943	0.925	0.919	1.078
	Staggered	43.574	31.69	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	1.24	1.11	1.09	1.08	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	129.58	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	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

Table 2 (*continue*)

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

Speedup Achieved by Parallel OETSN Using Different Types of Test Case

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

Table 3 (*continue*)

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

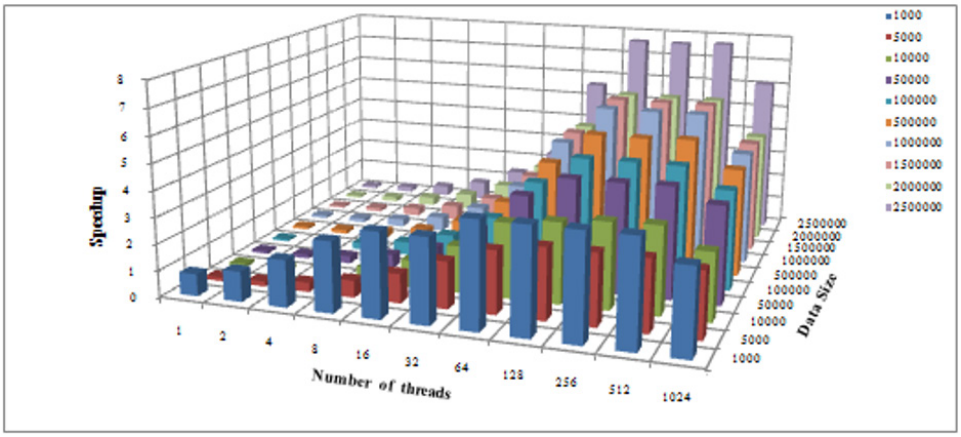


Figure 4. Speedup achieved by parallel OETSN using the uniform test case.

From Figure 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. This is because the data is not evenly distributed over the threads and some threads are ideal, hence the performance of the algorithm is degraded.

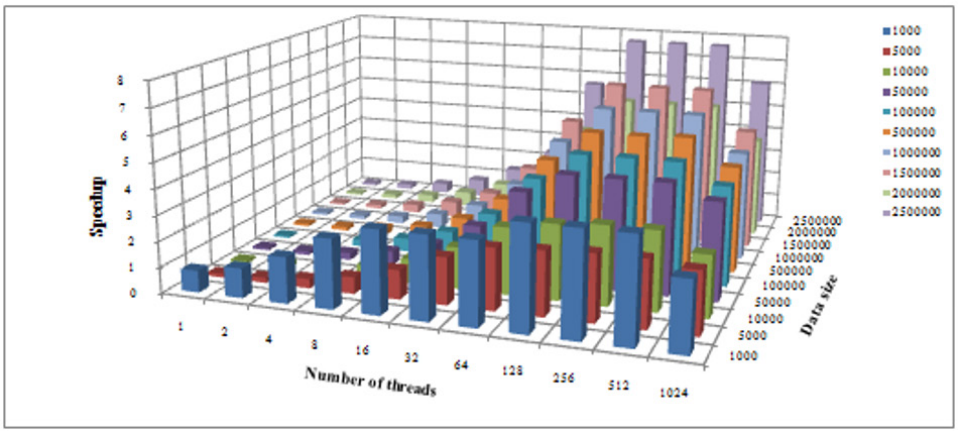


Figure 5. Speedup achieved by parallel OETSN using the Gaussian test case.

Figure 5 shows, the speedup for the Gaussian test case. Here we have achieved speedup seven times greater for the thread (T)=512 and data size (N)=2500000 in comparison to the sequential OETSN. The speedup difference can be seen at larger inputs, or we may say that speedup is directly proportional to the number of threads and size of the input.

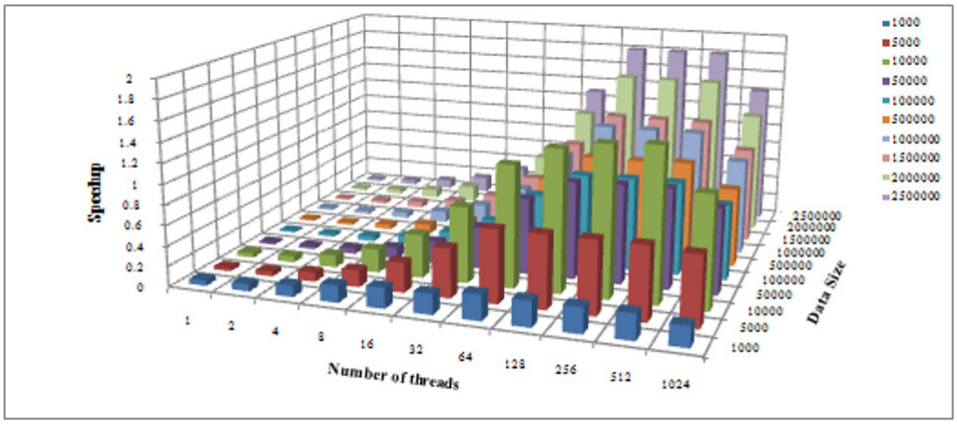


Figure 6. Speedup achieved by parallel OETSN using the zero test case.

Figure 6 shows, the speedup for a zero test case, highlighting that speedup is at least two times greater at T=512 and N=2500000 in comparison to the sequential OETSN. This is achieved in the zero test case.

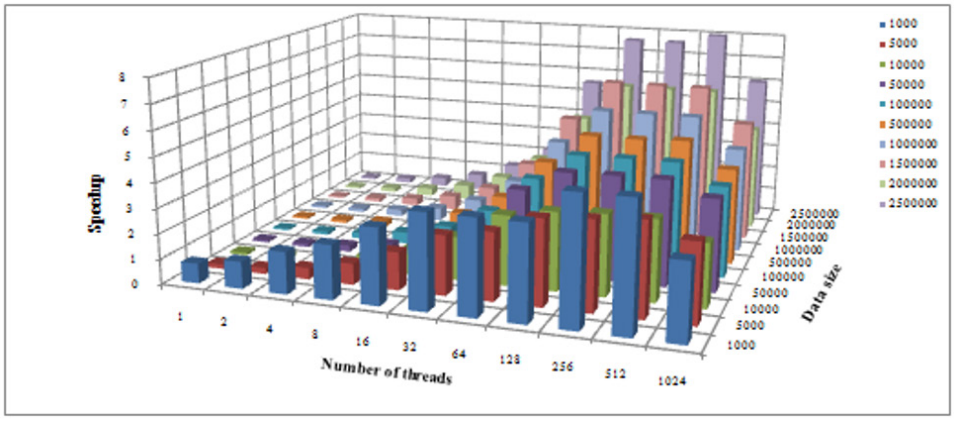


Figure 7. Speedup achieved by parallel OETSN using the staggered test case.

Figure 7 shows, the speedup for the staggered test case. In this test case, speedup is eight times greater at T=512 and N=2500000 in comparison to the sequential OETSN.

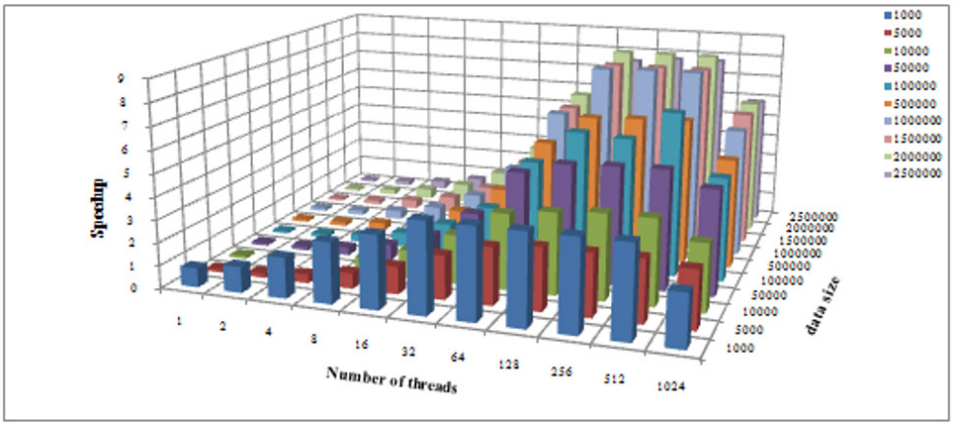


Figure 8. Speedup achieved by parallel OETSN using the bucket test case.

Figure 8 shows, the speedup for the bucket test case. The speedup increased by eight times at T=512 & N=2000000 in comparison to the sequential OETSN. Speedup is less at N=500 due to the smaller amount of data.

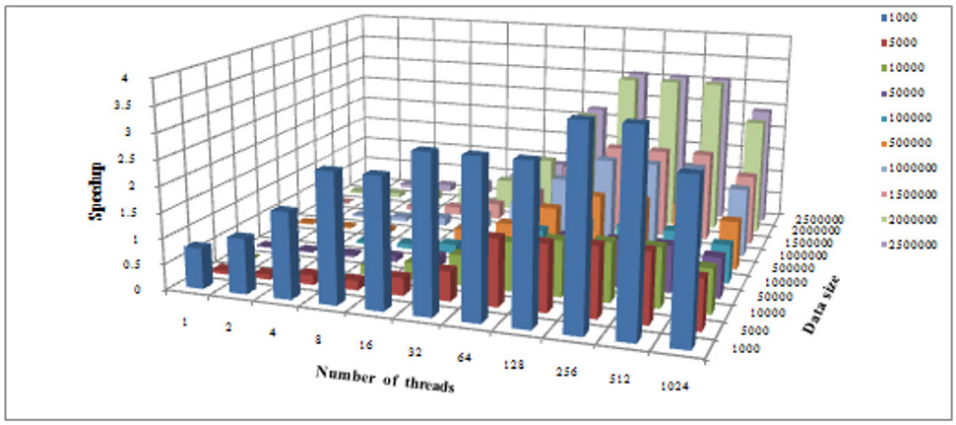


Figure 9. Speedup achieved by parallel OETSN using the sorted test case.

Figure 9 shows, the speedup for the sorted test case. The speedup is achieved three times greater at T=512 and N=1000 in comparison to the sequential OETSN. But in other test cases more speedup is achieved at N=2500000 or 2000000. This is because comparison is performed to the adjacent element only. There is no swapping performed as the data is already sorted.

In conclusion, we found that speedup is directly proportional to the number of threads and size of data in most of the tests. Maximum speedup is achieved by the bucket and staggered test case i.e. eight times greater in comparison to the sequential OETSN. Minimum speedup is achieved by the zero test case i.e. two times. We also found that in some cases good speedup is also achieved at N=1000 and 5000 at nearly seven and eight times greater.

EXPERIMENTAL EVALUATION OF PROPOSED MODIFIED PARALLEL OETSN ALGORITHM

Testing of proposed modified parallel OETSN algorithm is done on the sorting benchmark using GPU computing on CUDA hardware. Table 4 shows the execution time in seconds of proposed modified parallel OETSN algorithm using different types of test case. By examining Table 4, we found that the proposed approach is very efficient in comparison to the parallel OETSN only for the zero and sorted test cases. The execution time comparison for the sorted and zero test cases of parallel and proposed modified parallel OETSN are shown in Figures 10 and 11.

The results obtained in Table 4 are justified with the proposed algorithm discussed above. In the zero and sorted test cases, data did not require any swapping. In the odd-even module an evaluation function is called after one pass. It is a serial function, which added the number of swaps after every function was 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 are placed, but we still tried to take advantage of the proposed approach. However, it added an extra overhead on the execution time of the programme of the remaining test cases.

Figures 10 and 11 are shown with the sub-figures from (a) to (j). In all the sub-figures the *X*-axis represent the number of threads and the *Y*-axis represent the execution time in seconds. The execution time comparison of the zero and sorted test case of parallel OETSN and the proposed modified parallel OETSN is shown in Figure 10 and 11. The analysis of Figures 10 and 11 show that the execution time of the proposed modified parallel OETSN algorithm is much less compared to that of the existing parallel OETSN algorithm. The scale of the *Y*-axis was taken in logarithmic, using base to the power 2 because the execution time of the proposed approach is much less in comparison to the existing one.

Figure 10 describes the execution time comparison of the existing parallel OETSN and proposed modified parallel OETSN over the zero test case. As the modified parallel OETSN is exploiting the nature of the data, we got better results in all the cases of data size from *N*=1000 to 2500000. For the small data set we could see that the execution time of modified parallel OETSN was trending towards the existing parallel OETSN. This is due to the fact that each tread had very little data elements to sort.

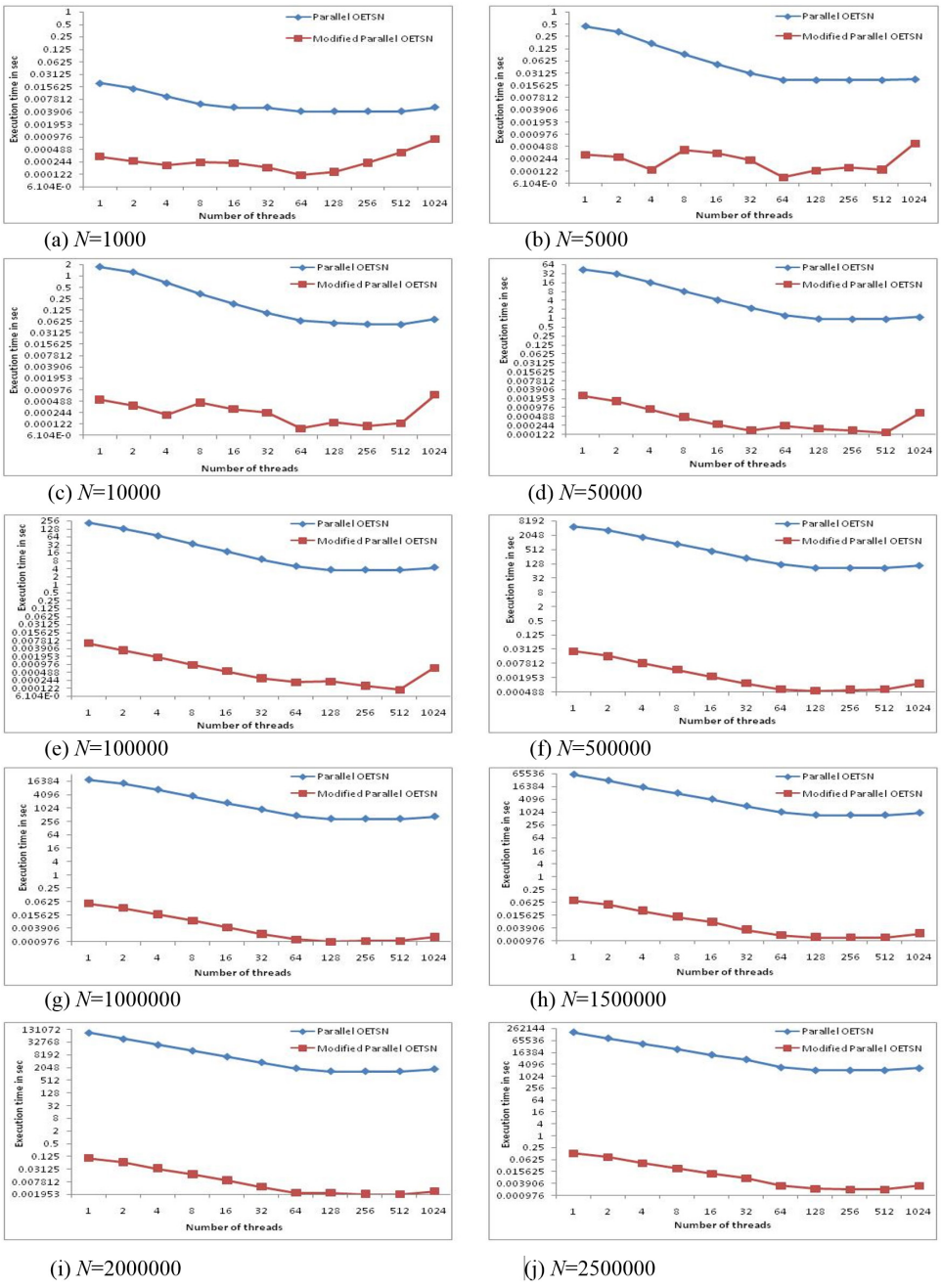
Figure 11 compares the execution time comparison of the parallel OETSN and modified parallel OETSN over the sorted test case. The zero test case was the special case of the sorted data. There is no swapping in both cases, and so, the trends of the modified OETSN are almost similar to those of the zero test case.

Table 4
Execution Time in Seconds of Modified Parallel OETSN Using Different Types of Test Case

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	42.9	22.52	11.67	6.03	3.15	1.76	1.4	1.4	1.39	1.5
	Gaussian	64.66	42.91	22.52	11.66	6.03	3.15	1.76	1.41	1.4	1.4	1.5
	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	5.58	2.89	1.62	1.4	1.4	1.4	1.5
	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	7.1	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	49.1	24.79	12.69	6.82	5.09	5.07	5.06	5.8
	Bucket	224.53	171.3	92.32	46.7	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

Table 4 (*continue*)

500000	Uniform	4870.8	3550.6	1921.6	1217.1	625.3	321.5	170.3	126.2	126.1	126	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	310.8	163.2	120.4	120.3	120.3	140.3
	Bucket	4690.2	3391.9	1791.7	1196.8	583.5	295	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	4599.6	2251.7	1264.8	1156.6	1145.9	1142	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	8393.9	4589.9	2179.7	1289.8	1198.8	1188.7	1179.7	1389.7
	Bucket	60511.7	31336.6	15428	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

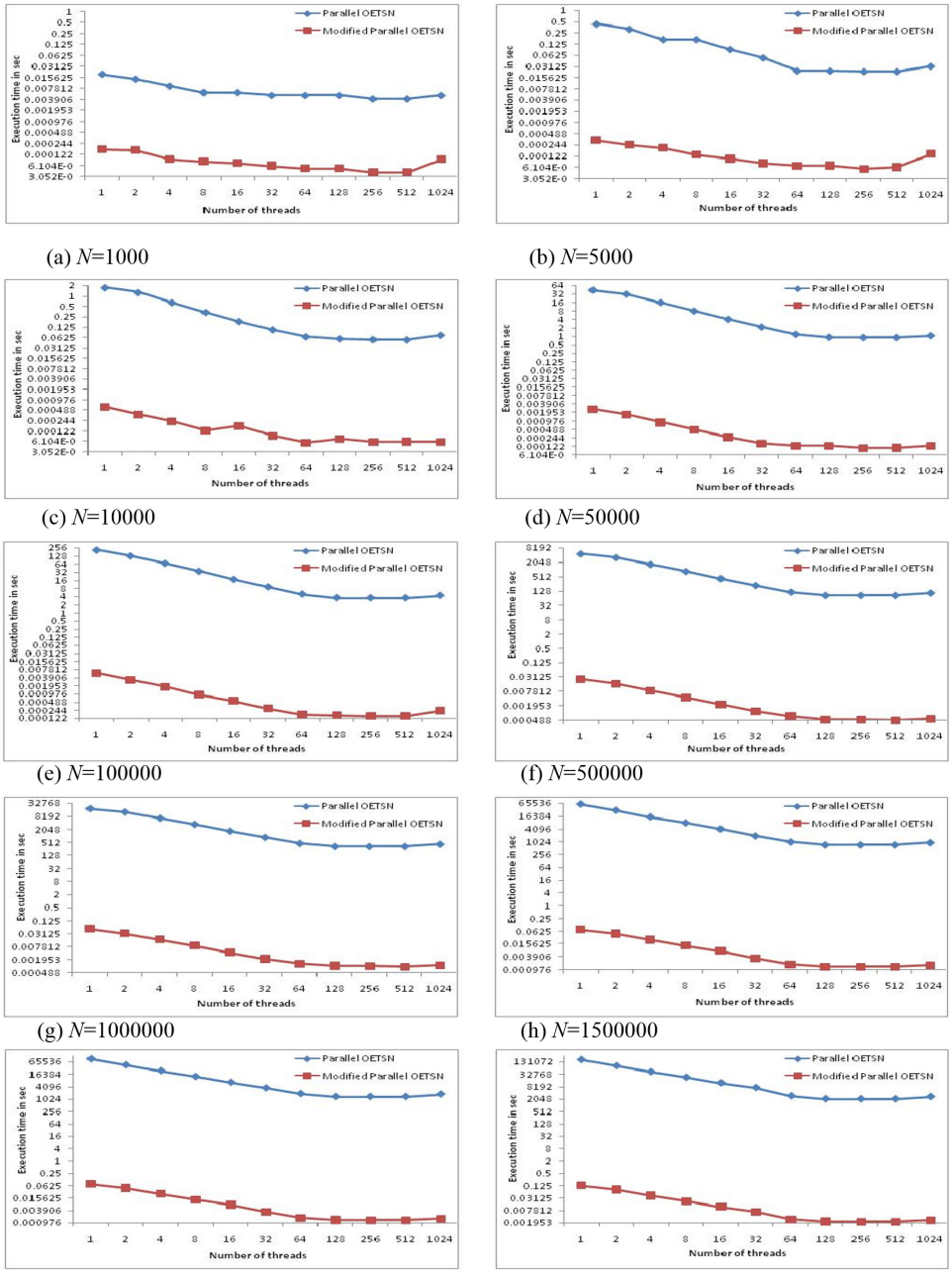
(c) N=10000 (d) N=50000

(e) N=100000 (f) N=500000

(g) N=1000000 (h) N=1500000

(i) N=2000000 (j) N=2500000

Figure 10. Execution time comparison of parallel and modified parallel OETSN 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

Figure 11. Execution time comparison of parallel and modified parallel OETSN using the sorted test case.

The overall conclusion of the paper is that odd-even transposition sorting has the sequential time complexity, $O(n^2)$. So, we parallelised the OETSN using GPU computing on CUDA hardware and then proposed the modified parallel OETSN using GPU computing. In the

proposed modified parallel OETSN, we reduced the number of levels of the network. After testing, we found that the number of levels of the OETSN are reduced for two types of test case i.e. zero and sorted test cases.

CONCLUSION AND FUTURE WORK

The suggested approach provided the best results when the data did not require swapping i.e. the data are sorted or unique. This significantly reduced the execution time in comparison to the existing one. The proposed approach achieved an improvement in execution time of 981661.6 times faster in the sorted test case and 904620.7 times in the zero test case using 2500000 elements and 1024 threads in comparison to the existing parallel OETSN. The time complexity is reduced from $O(n)$ to $O(1)$ because the proposed approach is executed using GPU. Some tasks are done sequentially on parallel machines so the time required in other types of test cases is slightly increased. We tested six types of test case. We varied the data from 1000 to 2500000 and the thread in multiples of two from 1 to 1024. The suggested approach can be further improved by using parallel reduction as we used linear addition. We used GPU computing using CUDA hardware with the computing capability 2.1 to test the algorithms. However, if the same algorithms will be used on hardware with the computing capability 3.0, then they will give an added advantage of unified memory architecture.

ACKNOWLEDGEMENTS

This work is performed in the frame of research-concerted action. All experiments are done in the research lab of Jaypee University of Information Technology, Waknaghat Solan, India. My co-author supervised me in using GPU computing with CUDA hardware.

REFERENCES

Ajdari, J., Raufi, B., Zenuni, X., & Ismaili, F. (2015). A Version of Parallel Odd-Even Sorting Algorithm Implemented in CUDA Paradigm. *International Journal of Computer Science Issues, 12*(3), 68-75.

Cederman, D., & Tsigas , P. (2009). GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA), 14*(4), 1-22.

Dowd, M., Perl, Y., Rudolph, L., & Saks, M. (1989). The Periodic Balanced Sorting Network. *Journal of the ACM, 36*(4): 738-757..

Farmahini-Farahani, A., Duwe, H. J., Schulte, M. J., & Compton, K. (2013). Modular design of high throughput, low-latency sorting units. *IEEE Transactions on Computers 62*(7), 1389-1402.

Grama, A., Gupta, A. & Karypis, G. (1994). I*ntroduction to parallel computing: Design and analysis of algorithms*. Redwood City, CA: Benjamin/Cummings Publishing Company.

Greb, A., & Zachmann, G. (2006). GPU-ABiSort: Optimal parallel sorting on stream architectures. *Parallel and Distributed Processing Symposium, IPDPS. 20th International*. IEEE, 1-10.

Jan, B., Montrucchio, B., Ragusa, C., Khan, F. G., & Khan, O. (2012). Fast parallel sorting algorithms on GPUs. *International Journal of Distributed and Parallel Systems*, *3*(6), 107-118.

Leischner, N., Osipov, V., & Sanders, P. (2010). GPU sample sort. *Parallel and distributed processing (IPDPS), IEEE International Symposium on IEEE*, 1-55.

Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS), 1*(8), 3-30.

Mozaffari, B. (2010). Optimization of odd-even transposition network. *2nd International Conference on Education Technology and Computer, 3,* 364-366.

Nadathur, S., Mark, H., & Michael, G . (2009). Designing efficient sorting algorithms for manycore GPU. *IEEE International Parallel and Distributed Processing Symposium*, 1-10.

Faujdar, N., & Ghrera, S. P. (2015). Performance evaluation of merge and quick sort using GPU Computing with CUDA. *International Journal of Applied Engineering Research*, *10*(18), 39315-393192 .

Peters, H., Schulz-Hildebrandt, O., & Luttenberger, N. Fast in-place sorting with CUDA based on bitonic sort. *Parallel Processing and Applied Mathematics* (pp. 403-410). Springer: Berlin Heidelberg.

Quinn, M. J. (1994). *Parallel computing: Theory and practice*. McGraw-Hill, Inc.

Salloum, S. N., & Perrie, A. L. (1999). Fault tolerance analysis of odd-even transposition sorting networks. *Communications, Computers and Signal Processing, IEEE Pacific Rim Conference on. IEEE*, *1,* 193-196.

Shifu, C., Qin, J., Xie, Y., Zhao, J., & Heng, P. A. (2009). A fast and flexible sorting algorithm with CUDA (pp. 281-290). In S.-L. C. Arrems Hua (Ed.). *Algorithms and Architectures for Parallel Processing* .

Ushijima, M., & Fujiwara, A. (2005). Sorting algorithms based on the odd-even transposition sort and the shearsort with DNA strands. *FCS*, 52-58.

Ye, Y., Du, Z., Bader, D. A., Yang, Q., & Huo, W. (2014). GPUMemSort: A high performance graphics co-processors sorting algorithm for large scale in-memory data. *Journal on Computing, 1*(2), 23-28.