ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645
Indian Journal of Science and Technology, *Vol 9(15), DOI: 10.17485/ijst/2016/v9i15/80080, April 2016*

Performance Evaluation of Parallel Count Sort using GPU Computing with CUDA

Neetu Faujdar* and SatyaPrakash Ghrera

Jaypee University of Information Technology, Department of CSE, Waknaghat, Himachal Pradesh, India; neetu.faujdar@mail.juit.ac.in, sp.ghrera@juit.ac.in

Abstract

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 paper, 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 paper 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 paper, 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. **Improvement:** After testing, we have achieved 66 times more efficient results of the parallel count sort in the case of execution time using Gaussian test case. We found that the parallel count sort performs, the better experimental results over sequential in all the test cases.

Keywords: CUDA, GPU, Parallel Count Sort, Sorting, Sequential Count Sort

Introduction

Nowadays multi core CPUs[1] are easily available in the market. The multi core CPUs are not sufficient to solve the high data computation task. So, recently GPU[2,3] introduced to solve these problems. The GPU is having the multi core processors thousands of threads running concurrently[4]. 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[5,6]. 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[7,8]. This algorithm works according to keys that are integer for sorting a collection of objects[9]. Count sort is an integer sorting algorithm. It is simple and efficient sorting algorithm[10] with linear time complexity O(n + k), where 'n' is the input elements and 'k' is the range of elements from 1 to k. When k=O(n) then count sort runs in O(n) time. It is stable sorting algorithm, i.e. if the same element occurs twice in the data, then it is maintain the order of duplicate keys.The ordering relation in countsort is derived from the set, i.e. to be sorted say 'A'.Suppose the set to be sorted is called 'A'. Then define the auxiliary array with equal size of A, say B. The algorithm stores the number of items in 'A' which are smaller than or equal to 'e' in B(e) for each element in 'A', say 'e'. Now we will sort the elements of 'C' based on the index of array 'B' for every element of the array 'A' and value of array 'B' is updated after each update in 'C'. The algorithm makes two passes over 'A' and

**Author for correspondence*

one pass over 'B'. The time complexity for 'n' input with range 'k' is O(n), where 'k' is less than 'n'. The count sort is cost efficient, stable and easy sorting algorithm, but it is also having the disadvantage over it. It is not recommended on large sets of data.This is the main drawback of count sort. In this paper the drawback has been taken as research concern and we have parallelized the count sort using GPU computing with CUDA. We have used the sorting benchmark to test both theversions of count sort (parallel and sequential). The sorting benchmark having the six types of test cases which are uniform, Gaussian, staggered, sorted, zero, and bucket test cases. The detailed information about sorting benchmark has been given in section 2. We have also measured the speedup of parallel count sort over sequential count sort. The contribution of the paper is as follows:

- The main content of the paper 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 has been taken as research concern.
- The parallel and sequential count sort is tested on sorting benchmark.
- The speedup is also calculated in this paper.

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[11].

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. The duplicate removal is allowed to have a natural functional and data parallel implementation which makes it for GPUs. The suggested algorithms are implemented on the GPU in Obsidian. The Obsidian is a high-level do-main specific language for GPU programming. The result shows the implementations in many cases outperforms sorting algorithm provided by the library Thrust. The occurrence sort is two faster than the ordinary counting sort. When we consider the sorting algorithms for GPU counting sort is an important contender. The occurrence sort is highly preferable only when applicable. The author has also shown that Obsidian can produce very competitive code. The contribution of the paper as follows[12]:

- The author showed that counting sort is a competitive algorithm for sorting keys on the GPU and outperformed the sorting implementation in the library Thrust.
- The author showed the occurrence sort suitable for implementing on the GPU.
- The Obsidian implementation of two sorting algorithm is detailed with CUDA.

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[13].

2. Sorting Benchmark

We have tested the both the versions (sequential and parallel) of the count sort algorithm on six types of test cases which are Uniform, Sorted, Zero, Bucket, Gaussian, and Staggered[14-16]. We have varied the data from 100 to 10000000 and the thread in the multiple of 2 from 1 to 1024.

1. Uniform test case: In this test case values are picked randomly from 0 to 2.
2. Gaussian test case: In this test case the distribution of data is created by taking the average of four randomly values picked from the uniform distribution.
3. Zero test case: In this test case a constant value is used.
4. Bucket test case: For p∈ N, the input of size 'N' is split into 'p' blocks, such that the first n/p^2elements in each of them 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∈ N, the input of size 'N' is split into 'p' blocks such that if the block index is i≤ 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 value has been taken.

3. Hardware

We ran the new version of the sequential count sort algorithm on Window 7 64-bit operating system Intel® core™ i5 processor 3230M @ 2.60 GHz machine[17]. The new version of the parallel count sort algorithm ran on Window 7 32-bit operating system Intel® core™ i3 processor 530@ 2.93 GHz machine. The system has the GeForce GTX 460 graphic processor with (7 multiprocessors X (48) CUDA cores\MP) = 336 CUDA cores. There are maximum 1536 threads per multiprocessor and 1024 threads per block. System having the CUDA runtime version 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. System having the total number of registers available per block is 32768 and warp size is 32. Maximum sizes of each dimension of a block are 1024 x 1024 x 64 and maximum size of each dimension of a grid is 65535 x 65535 x 65535.

4. Implementation of Sequential Count Sort Algorithm

In this section the implementation results of the sequential countsort 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 1. In Table 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 1. We can see that zero test case is more efficient compare to other test cases.

5. Implementation of Parallel Count Sort Algorithm

In this section we have implemented the parallel count sort algorithm using GPU computing with CUDA. We have tested the parallel count sort using sorting benchmarks. The benchmarks having the six types of test cases. The Table 2 shows the execution time in milliseconds using uniform test case. By analysing the Tables 1 and 2, we can see that the parallel count sort is much more efficient than sequential count sort. We can see this effect in Figure 1 and both the count sort (sequential, parallel) is recommended for large number of data.

In the Tables 2 to 7 we have shown the parallel execution time using six types of test cases with varying data and thread size. 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 1 to 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 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 3 and in Figure 2.

The Table 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

Table 1. Execution time in milliseconds of sequential count sort

Execution time of sequential count sort using six types of test cases						
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

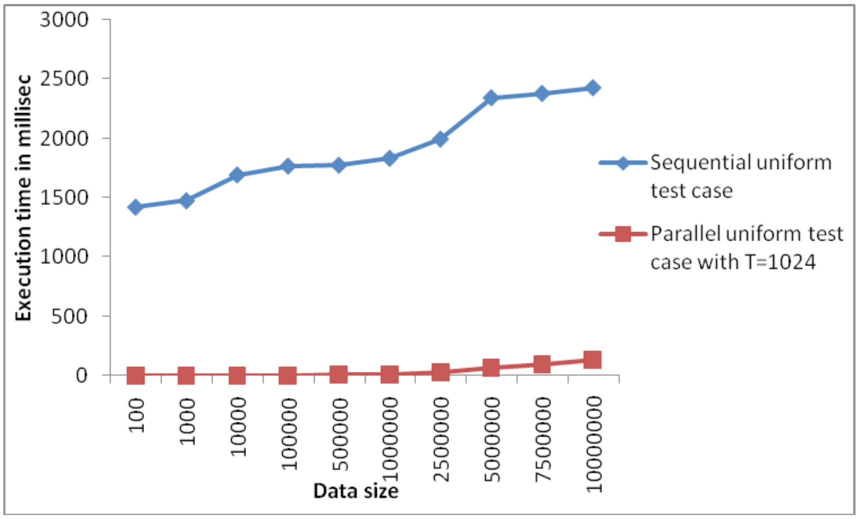

Figure 1. Execution time comparison between parallel and sequential count sort using uniform test case.

Table 2. Execution timein milliseconds of parallel count sort using uniform test case

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

Table 3. Execution timein milliseconds of parallel count sort using sorted test case

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

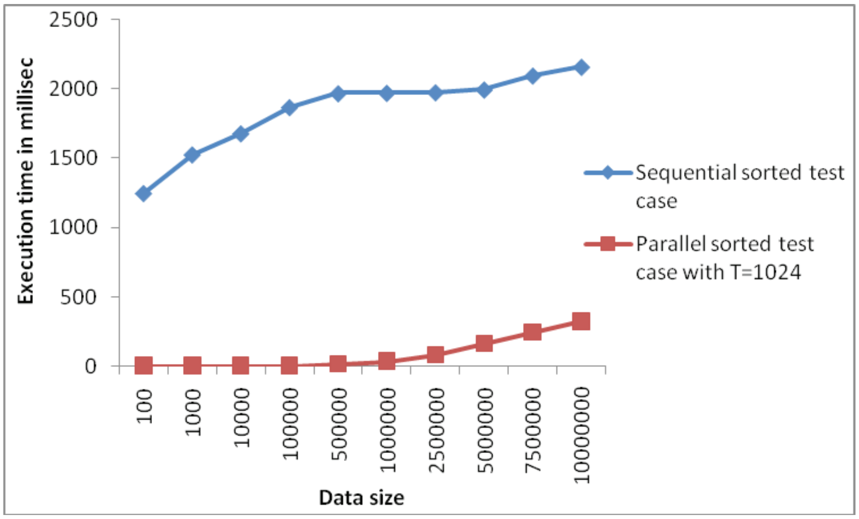

Figure 2. Execution time comparison between parallel and sequential count sort using sorted test case.

Table 4. Execution timein milliseconds of parallel count sort using zero test case

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

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 4 and Figure 3 we can see that sequential count is more efficient than parallel when the test case is zero.

The Table 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 5 and in the Figure 4. The Figure 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.

The Table 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 6 and in Figure 5. The Figure 5 tells us that parallel Gaussian test case is having the very much less execution time in comparison to the sequential Gaussian test case.

The Table 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

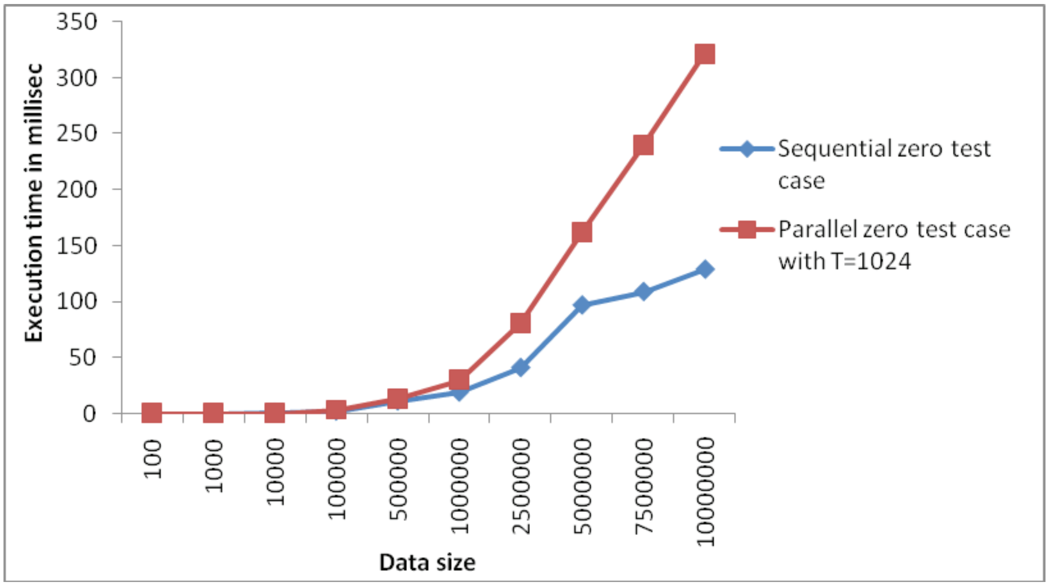

Figure 3. Execution time comparison between parallel and sequential count sort using zero test case.

Table 5. Execution timein milliseconds of parallel count sort using bucket test case

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

than sequential. We can see this effect in Table 7 and in Figure 6.

6. 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[18]. 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. In the Tables 8, 9, 10, 11 and 12 we have measured the speedup achieved by the parallel count sort using the different types of test cases. In all the Tables 8, 9, 10, 11 and 12 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

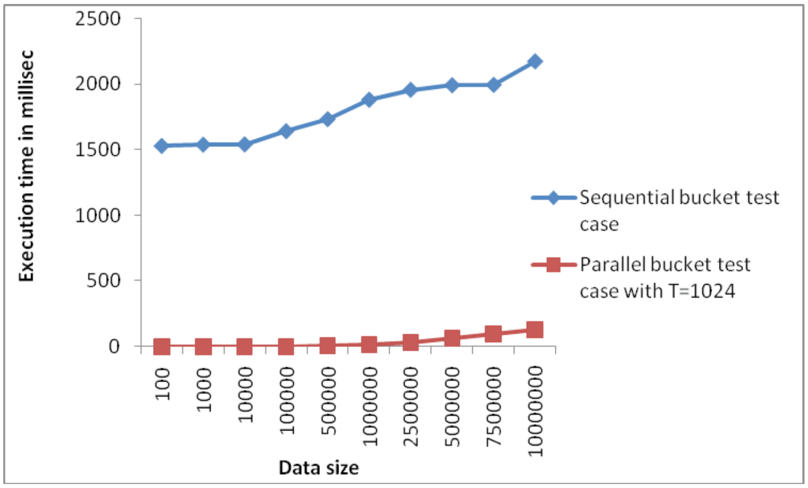

Figure 4. Execution time comparison between parallel and sequential count sort using bucket test case.

Table 6. Execution timein milliseconds of parallel count sort using Gaussian test case

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

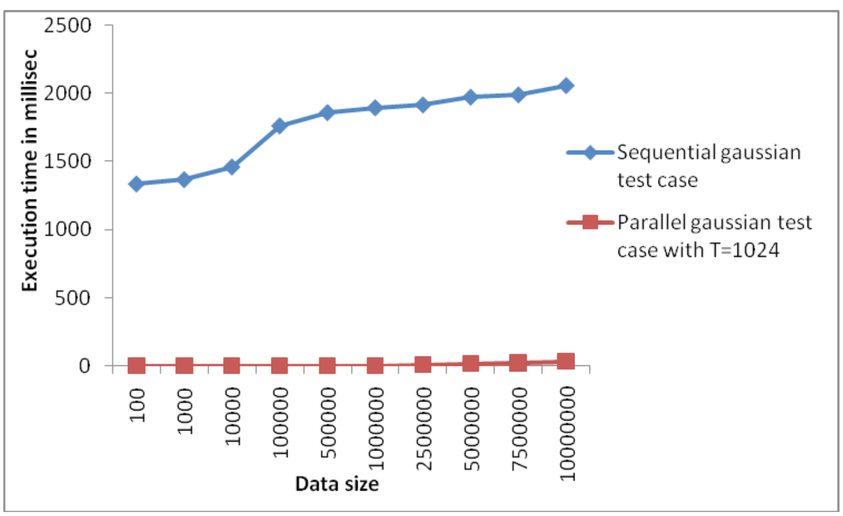

Figure 5. Execution time comparison between parallel and sequential count sort using Gaussian test case.

Table 7. Execution timein milliseconds of parallel count sort using staggered test case

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

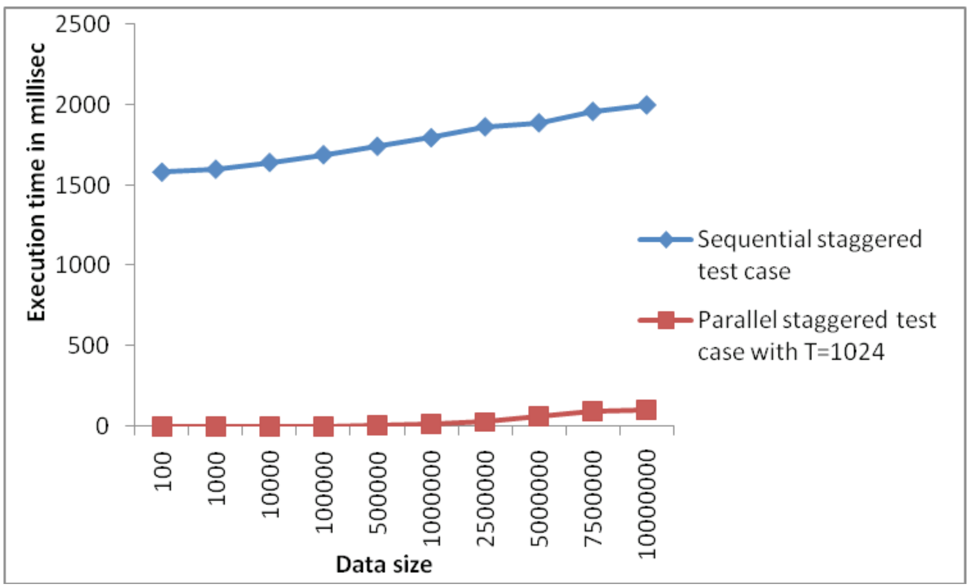

Figure 6. Execution time comparison between parallel and sequential count sort using staggered test case.

Table 8. Speedup achieved by parallel count sort using different types of test cases with N=7500000

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

is explained earlier. The Figures 7, 8, 9, 10 and 11 have been drawn using the Tables 8, 9, 10,11 and 12. In all the FiguresX-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 isalso increases. And in all the Figures Gaussian test case has achieved more speedup compared to other test cases.

Table 9. Speedup achieved by parallel count sort using different types of test cases with N=10000000

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

Table 10. Speedup achieved by parallel count sort using different types of test cases with N=5000000

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

Table 11. Speedup achieved by parallel count sort using different types of test cases with N=2500000

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

Table 12. Speedup achieved by parallel count sort using different types of test cases with N=1000000

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

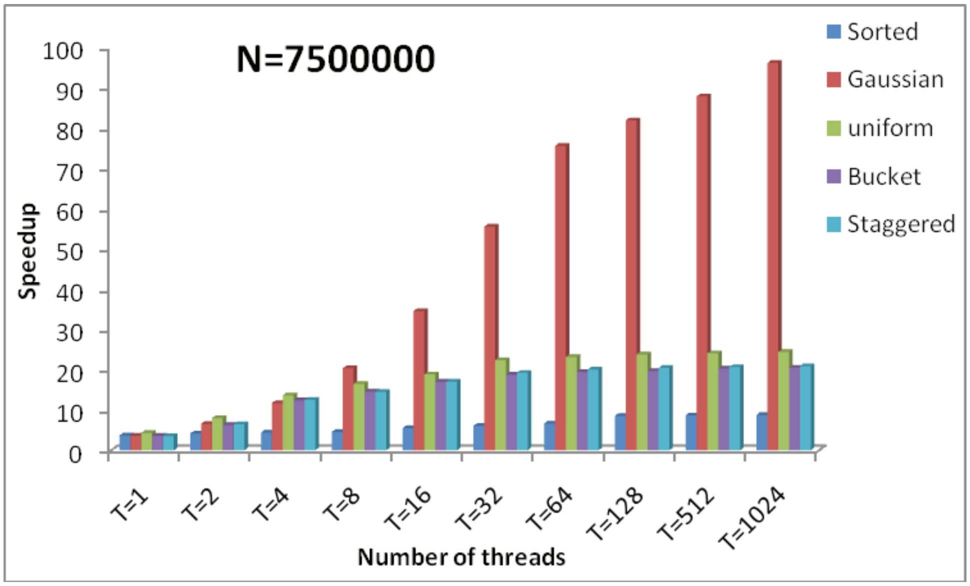

Figure 7. Speedup achieved by parallel count sort using different types of test cases with N=7500000.

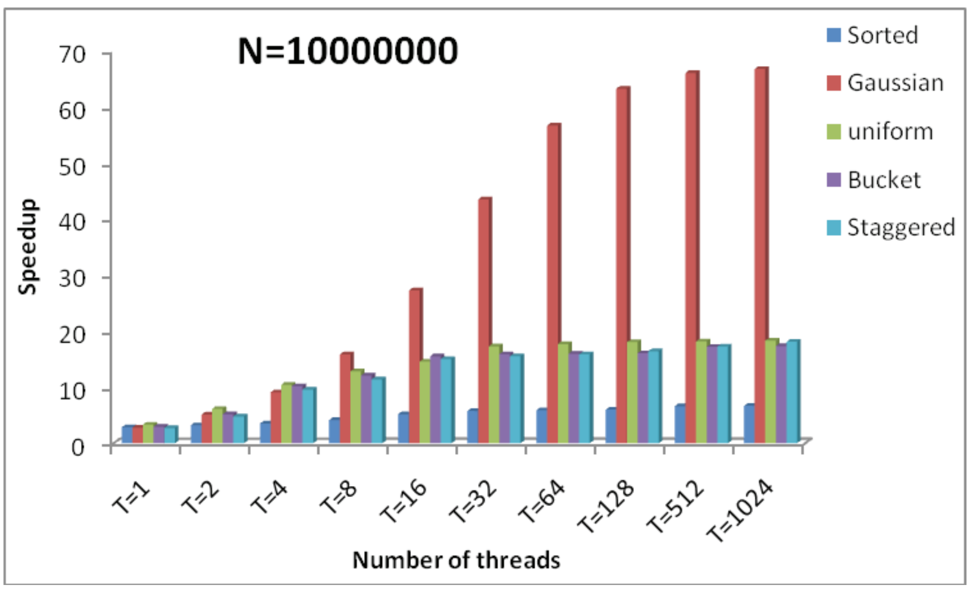

Figure 8. Speedup achieved by parallel count sort using different types of test cases with N=10000000.

7. Conclusion

The count sort is recommended for large sets of data as shown by implementation results. We have done the testing on the six types of test cases. We have varied the data from 100 to 10000000 and the thread in the multiple of 2 from 1 to 1024. We have used the GPU computing using CUDA hardware having the compute capability 2.1 to test the algorithms. But, if the same algorithm has been used on the hardware having the compute capability 3.0, then it will give an added advantage of unified memory architecture. We have also measured the speedup achieved by the

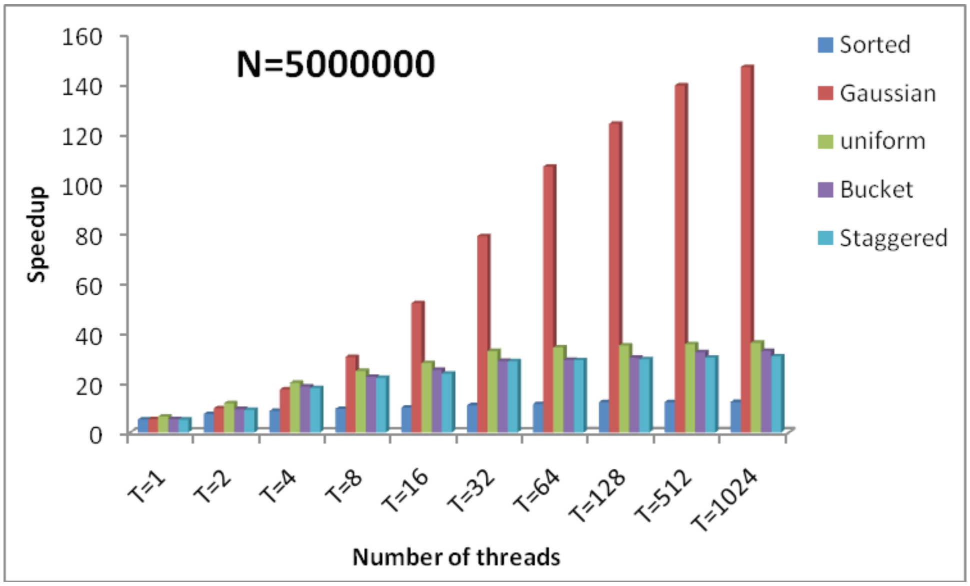

Figure 9. Speedup achieved by parallel count sort using different types of test cases with N=5000000.

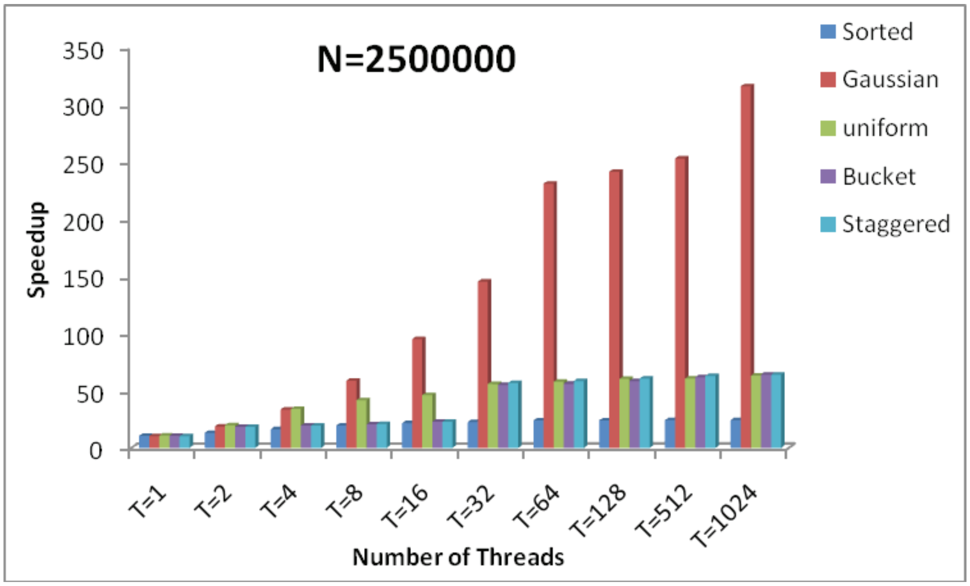

Figure 10. Speedup achieved by parallel count sort using different types of test cases with N=2500000.

parallel count sort over sequential. The main conclusion 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. Andthe parallel count sort algorithm has done using GPU computing with CUDA hardware.

8. Acknowledgment

This work has been done only for research concern. All experimental results are done in the research lab of Jaypee University of Information Technology, WaknaghatSolan, India.

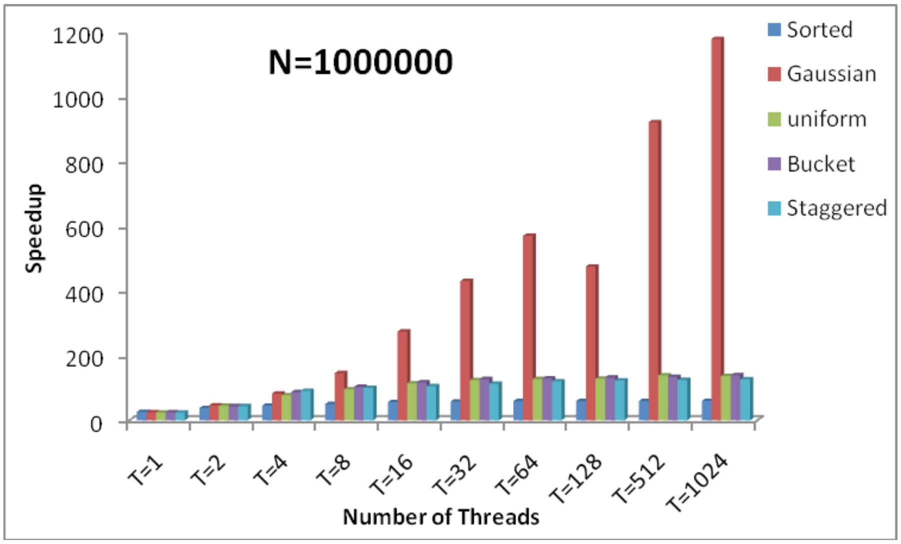

Figure 11. Speedup achieved by parallel count sort using different types of test cases with N=1000000.

References

1. Creeger M. Multicore CPUs for the masses. ACM Queue. 2005; 3(7):64-5.
2. Hacker H, Trinitis C, et al. Considering GPGPU for HPC centers: Is it worth the effort? In Facing the Multicore-Challenge. Lecture Notes in Computer Science. 2010; 63(10): 118–30.
3. Nickolls J, Dally WJ. The GPU computing era. IEEE Micro. 2010; 30(2):56–69.
4. Zhang Y, Owens JD. A quantitative performance analysis model for GPU architectures. IEEE 17th International Symposium on High Performance Computer Architecture; San Antonio, TX. 2011. p. 382–93.
5. Garland M. Parallel computing with CUDA. IEEE Symposium on Parallel and Distributed Processing IPDPS; Atlanta, GA. 2010. p. 1-10.
6. KindratenkoVV, Enos J, Shi G, et al. GPU clusters for high-performance computing. IEEE International Conference on Cluster Computing Workshops, CLUSTER; 2009. p. 1–8.
7. Faujdar N, Ghrera SP. Analysis and testing of sorting algorithms on a standard dataset. IEEE 5th International Conference on Communication Systems and Network Technologies (CSNT); Gwalior, India. 2015. p. 962-7.
8. Faujdar N, Ghrera SP. Performance Evaluation of merge and quick sort using GPU Computing with CUDA. International Journal of Applied Engineering Research (IJAER). 2015; 10(18):39315-9.
9. Joshi R, Panwar GS, Pathak P. Analysis of non-comparison based sorting algorithms: A review. International Journal of Emerging Research in Management and Technology. 2013; 2(12):61-5.
10. Mishra A D, Garg D. Selection of best sorting algorithm. International Journal of Intelligent Information Processing. 2008; 2(2):363-8.
11. Keshav B, Kots A. Implementing and analyzing an efficient version of counting sort (E-counting sort). International Journal of Computer Applications. 2014; 98(9):1-2 .
12. Svenningsson, David J, et al. Counting andoccurrence sort for GPUs using an embedded language. ACM Proceedings of the 2nd ACM SIGPLAN workshop on Functional High-Performance Computing; Boston, MA, USA. 2013. p. 37-46.
13. Weidong S, Ma Z. Count sort for gpu computing. IEEE Parallelanded Distributed Systems (ICPADS) 15th International Conference; Shenzhen. 2009. p. 919-24.
14. Cederman D, Tsigas P. Gpu-quicksort: A practical quicksort algorithm for graphics processors. Journal of Experimental Algorithmics (JEA). 2009. 14(4):1-24.
15. Leischner N, Osipov V, Sanders P. GPU sample sort. IEEE International Symposium on Parallel and Distributed Processing (IPDPS); Atlanta, GA. 2010. p. 1-10.
16. Matsumoto M, Nishimura T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS). 1998; 8(1):3-30.
17. Krishnam Raju IR, et al. Deadline aware two stages scheduling algorithm in cloud computing. Indian Journal of Science and Technology. 2016; 9(4):1-10.
18. Chitra E. Vigneswaran T. An Efficient low power and high speed distributed arithmetic design for FIR filter. Indian Journal of Science and Technology. 2016; 9(4):1-5.