Shahid Beheshti University

Faculty of Mathematical Science

Department of Computer Science

Concurrent Programming – Spring 2022 – Final Project

Parallel Sorting

By:

Arman Davoodi

Contents

Introduction

Sorting is one of the most fundamental problems in computer science. In fact, various problems can be reduced to sorting; hence performance is a matter of utmost importance for sorting and many algorithms have been presented for this purpose.

One way of increasing the speed of algorithms is to use parallelization. In this project, I have implemented 7 different sorting algorithms in C++ and parallelized 5 of them by using OpenMP and CUDA libraries. I have also compared the performances of these algorithms on different vectors.

The results given in this report are based on several runs on a system with Intel Core I7-8550U CPU, GeForce GTX 1050, CUDA version 10.1 and Linux mint version 20 operating system.

Simple Sorting Algorithms

There are numerous algorithms for sorting arrays. However, the easiest sorting methods that come into mind generally have a time complexity of $O(n^2)$. This section is about introducing bubble sort and insertion sort which are two most known sorting algorithms of this order.

Bubble Sort

The idea behind this algorithm is to swap each element of the array with its right neighbor if the neighbor has a lower value than the stated element. By doing so, it can be guaranteed that at the end of each iteration, the maximum remaining element will be at the end of the array. Below, a pseudocode for bubble sort is presented which sorts an array of size n in ascending order.

```
1. bubble_sort(array, n):
2.     sorted = false
3.     for i=0 to n-2:
4.         sorted = true
5.         for j=0 to n – i – 2:
6.             if array[j] > array[j+1]:
7.                 sorted = false
8.                 swap(array[j], array[j+1])
9.         if not sorted: break
```

Algorithm 1

In the second line of *Algorithm 1*, a sorted flag has been defined. Whenever the algorithm starts an iteration in line 4, it is assumed that the array is already sorted. If no swaps happen in this iteration it means that the array is already sorted and the algorithm can be ended. Also, the second loop in line 5 will only be repeated n – i – 1 times as it is already known that the last i^{th} elements are in their correct position.

The best case of the algorithm happens when the array is already sorted. In this case the sorted flag is never set to false and therefor after a single iteration the algorithm will be finished. The time complexity in best case for this algorithm is hence O(n).

On the other hand, the worst case of this algorithm happens when the array is sorted in reverse. In this case, at each iteration, the algorithm will swap the largest element with its neighbor until it reaches the end of the array doing 1 + 2 + … + n-1 swaps which makes the time complexity of this algorithm in worst case $O(n^2)$.

We can also note that the space complexity of this algorithm is O(1) and that it is also an stable sorting algorithm, meaning that the order of elements with the same value but different indexes will not change.

Insertion Sort

The idea behind this algorithm is to remove each element from the array and then shift all of its left neighbors with higher values than itself right and then put the removed element in the emptied space. Below, a pseudocode for insertion sort is presented which sorts an array of size n in ascending order.

```
1. insertion_sort(array, n):
2.     for i=1 to n-1:
3.         temp = array[i]
4.         for j=i downto 1:
5.             if array[j-1] > temp:
6.                 array[j] = array[j-1]
7.             else break;
8.         array[j] = temp
```

Algorithm 2

The loop in the 4th line of the algorithm, will shift all elements higher than the one stored in temp, right. By doing this process n – 1 times we can guarantee that the array is sorted.

The best case of the algorithm happens when the array is already sorted. In this case the condition in line 5 will never be true and therefor the algorithm will just visit n-1 elements. The time complexity in best case for this algorithm is hence O(n).

Whilst, the worst case of this algorithm happens when the array is sorted in reverse. In this case, at each iteration, the algorithm will have to shift all left-side elements which makes the time complexity of this algorithm in worst case $O(n^2)$.

We can also note that the space complexity of this algorithm is O(1) and that it is also an stable sorting algorithm.

Faster Sorting Algorithms

In the previous section, two simple sorting algorithms were introduced. Both of those algorithms have time complexity of $O(n^2)$. However, there are more efficient algorithms that can sort big enough arrays in shorter times. A few famous examples of such algorithms are merge sort, heap sort and quick sort.

Merge Sort

Assuming we have two sorted arrays, it is quite easy to construct an algorithm which merges these two arrays and returns a single sorted array consisting of all elements of the two sorted arrays. Algorithm 3 bellow is one of such algorithms.

```
merge(arr_left, arr_right):
    n = len(arr_left), m = len(arr_right)
    arr := initialize array of size n + m
    i = 0, j = 0
    while(i < n and j < m):
        if arr_left[i] > arr_right[j]:
            arr[i+j] = arr_right[j]
            j++
        else
            arr[i+j] = arr_left[i]
            i++
    while(i < n)
        arr[i+j] = arr_left[i]
        i++
    while(j < m)
        arr[i+j] = arr_right[j]
        j++
    return arr
```

Algorithm 3

If both input arrays are of size n/2, then this algorithm will have a time complexity and space complexity of O(n).

The merge sort algorithm uses this idea and sorts the input array by splitting it into two halves, then sorts those two halves recursively and at last merges the two sorted halves by using *Algorithm 3*. *Algorithm 4* represents a simple pseudocode for this algorithm.

```
merge_sort(array, startIdx, end):
    // end is the index after the last index
    if end – startIdx < ad_hoc:
        insertion_sort(array, startIdx, end)
    else:
        left = startIdx, mid = (startIdx + end) / 2, right = end
        merge_sort(array, left, mid)
        merge_sort(array, mid, right)
        if array[mid – 1] > array[mid]:
            merge(array, left, mid, right)
```

Algorithm 4

In both best and worst cases, this algorithm has a time complexity of O(n*lg(n)) if the input array is of size n. This can be proven by writing the recursive formula for time complexity of this algorithm and then using the master theorem. The memory complexity of this algorithm is of O(n) because of the merge function. We can also note that this algorithm is a stable sorting algorithm.

Since insertion sort, is very efficient for small arrays, we can define an ad_hoc and whenever the size of array was lower than this ad_hoc, we switch to insertion sort. This can boost the performance of the algorithm.

Because merging is most efficient when the number of runs is equal to, or slightly less than, a power of two, and notably less efficient when the number of runs us slightly more than a power of two, we can choose an ad_hoc that tries to ensure the former condition. Such algorithm is implemented in the merge.h file and is referred to as getMinRunSize.

The condition on the 9th line ensures that the merge function is called only when the array is not already sorted.

A bottom-up implementation of this algorithm has a better performance than the recursive implementation.

Sorting Networks

What is important in the designing process of sequential algorithms, is doing as few operations as possible which is achieved by having sophisticated control over the algorithm's flow. This method have absolutely no problem for designing sequential algorithms as CPUs are great at flow control. However, due to the fact that GPUs are quite weak at flow control whilst they can do far more parallel computations, this approach will most likely yield unsatisfying results if used to design SIMD parallel algorithms. Therefore, when designing parallel algorithms for GPUs, we can tolerate more independent computations and have much less flow control.

Most famous sorting algorithms that we know such as insertion sort, merge sort, quick sort and heap sort are data dependent algorithms. This means that their input will affect the order and quantity of their operations. For instance, in merge function of merge sort, we have two pointers and depending on the input, either the first or the second pointer will go onward. This is a good example of having a sophisticated flow control to reduce total number of operations.

In contrast, we have sorting algorithms that are not data dependent, meaning that no matter the input given to them, they will always follow the same flow. A good example of such algorithms are sorting networks.

In sorting networks, in addition to a constant number of wires, which will carry our input signals, we have a number of fixed connectors in specific places, connecting two wires together.

At the points of connection, the connected wires will exchange their signals if the upper wire's signal is stronger than the lower wire's signal. The goal is to connect wires in a way that at the end, our signals are sorted.

Figure 1, is a demonstration of such networks.

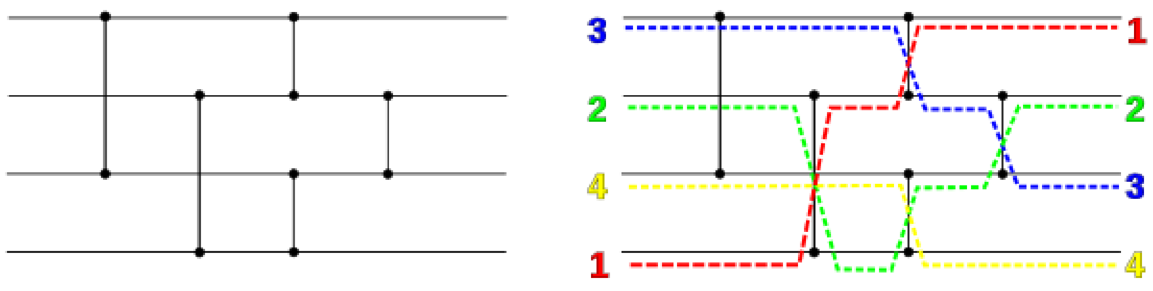


Figure 1

Another positive aspect of sorting networks is that, even though they are constructed for fixed number of inputs or signals, sorting networks of larger size can be easily constructed from smaller sorting networks.

Assume a sequence is bitonic if, the sequence is descending up to a point and from there it is ascending or the sequence is ascending up to a point and from there it is descending. For instance sequences {1 2 5 1 0}, {7 6 2 3 4} are bitonic but {1 2 3 0 5} is not a bitonic sequence.

If we divide a bitonic sequence into two sorted sequences (one ascending and one descending), then starting from the first element of each sequence, we compare the elements and swap them if the element of the first sequence is higher. By doing this we will acquire two bitonic sequences where the elements of one of the sequences is higher than the elements of the other sequence. By doing this recursively we can acquire a sorted array.

This example is the algorithm used in Bitonic sort to merge two sorted sequences.

Odd-Even/Brick Sort

Odd-Even sort which is also known as brick sort, is a sorting network as well as a variant of bubble sort algorithm. This algorithm does the same comparisons a bubble sort implementation without a sorted flag does but in different order. The idea behind the algorithm is the same as bubble sort, however it first compare and swaps odd elements and then it will do so for elements at even index. The algorithm repeats this routine n/2 times until the array is sorted.

In sequential form, just like bubble sort, a sorted flag can also be added to increase the performance of the algorithm.

Algorithm 5 represents a simple pseudocode for this algorithm.

```
1. brick_sort(array, n):
2.     for phase = 0 to n – 1:
3.         for i = phase mod 2 to n – 2:
4.             if array[i] > array[i+1]: swap(array[i], array[i+1])
```

Algorithm 5

The network of size 8 for this algorithm is shown in *Figure 2* bellow.

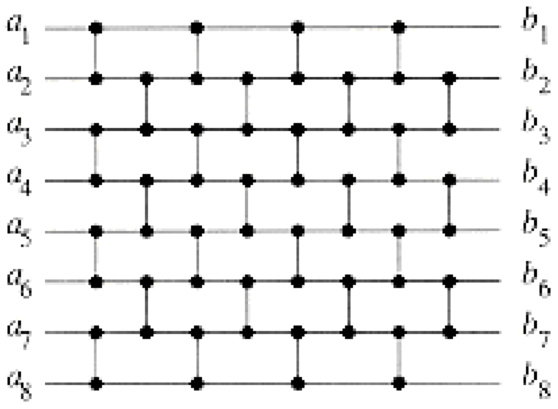


Figure 2

Unlike bubble sort, in this algorithm we have a lot of independent operations that we can do in parallel. The boxes in *Figure 3* represent the phases. The compare and swaps operations for each phase can be done in parallel.

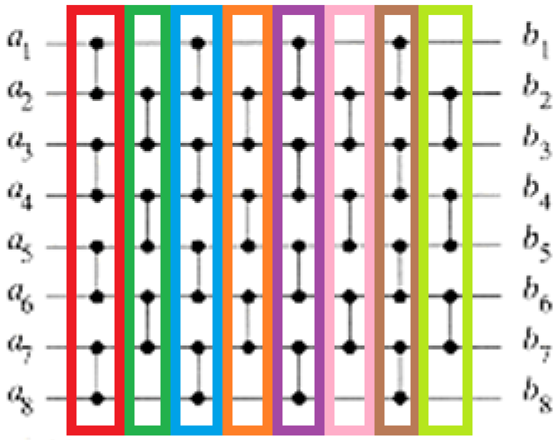


Figure 3

Considering these facts, in general this algorithm, is a stable sorting algorithm, with time complexity of O(n^2) sequentially and O(n) when run parallel on a PRAM. The memory complexity of the algorithm is O(1).

Bitonic Sort

The idea behind bitonic sort is pretty similar to that of the merge sort. The only difference between these two algorithms is that the merge algorithm used for bitonic sort is not tha merge algorithm used for the merge sort. The merge algorithm for bitonic sort, reverses the second half of the array hence making a bitonic sequence. Then it sorts the bitonic sequence with the method mentioned before. *Figure 4* and *Figure 5* demonstrate the bitonic sorting network of size 16.

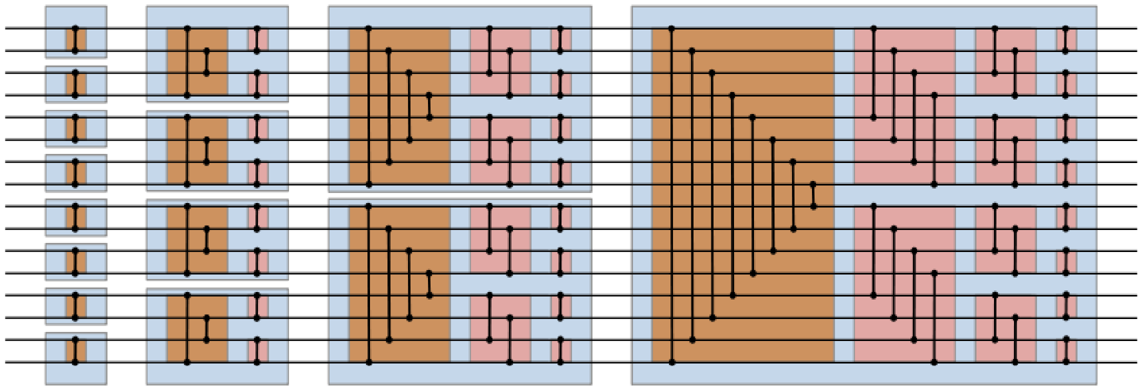


Figure 4

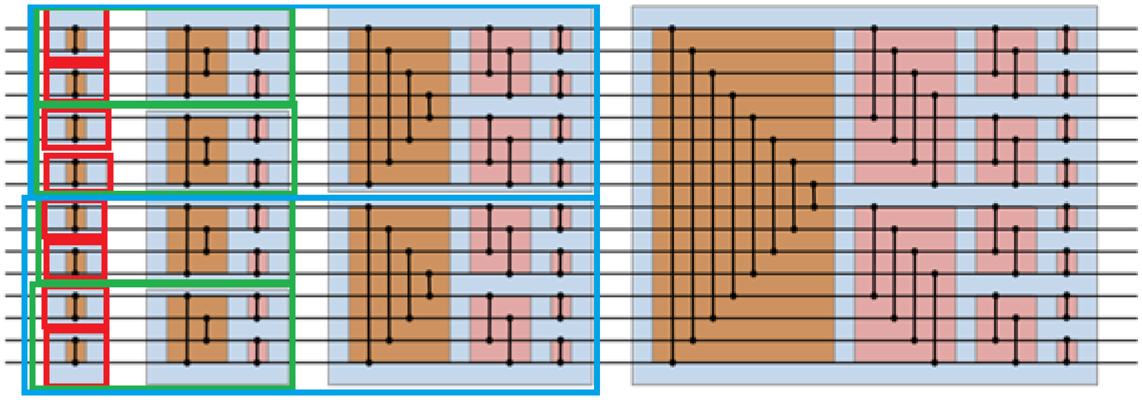


Figure 5

The recursive approach of the algorithm is quite evident in *Figure 5* as each red, green and blue box represent a sorted array of size 2, 4 and 8 respectively.

Figure 6 is another illustration of the same network.

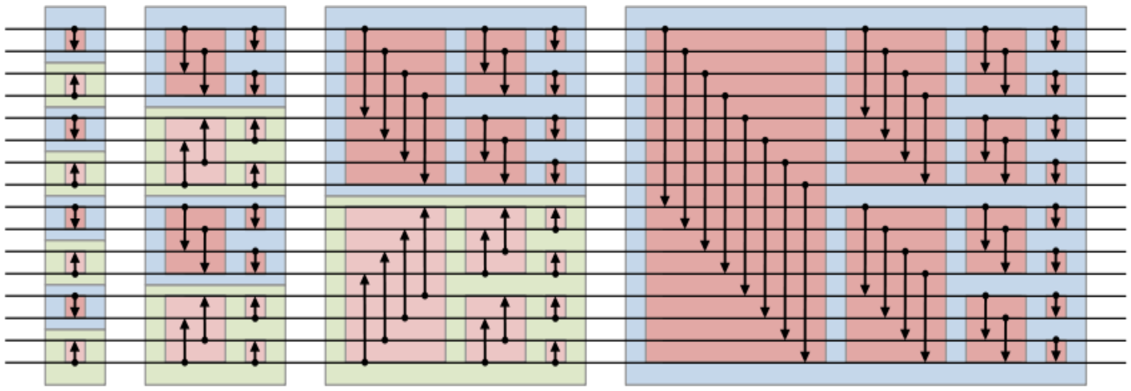


Figure 6

Both upward and downward arrows in *Figure 6* are connectors. However, whilst the downward arrows sort the signals in ascending order, the upward arrows sort the signals in descending order. At the end, the input will be sorted in ascending order.

This algorithm can be implemented in a way to have a space complexity of O(1). It is also considered a stable sorting algorithm.

The time complexity of this algorithm when run sequentially is O(n*lg^2(n)) as the bitonic-merge algorithm has a time complexity of O((n/2)*lg(n)).

Although this algorithm does more comparisons compared to algorithms such as merge sort, it can be parallelized quite efficiently. Each red box in *Figure 7* represent a stage in which all comparisons can be done in parallel.

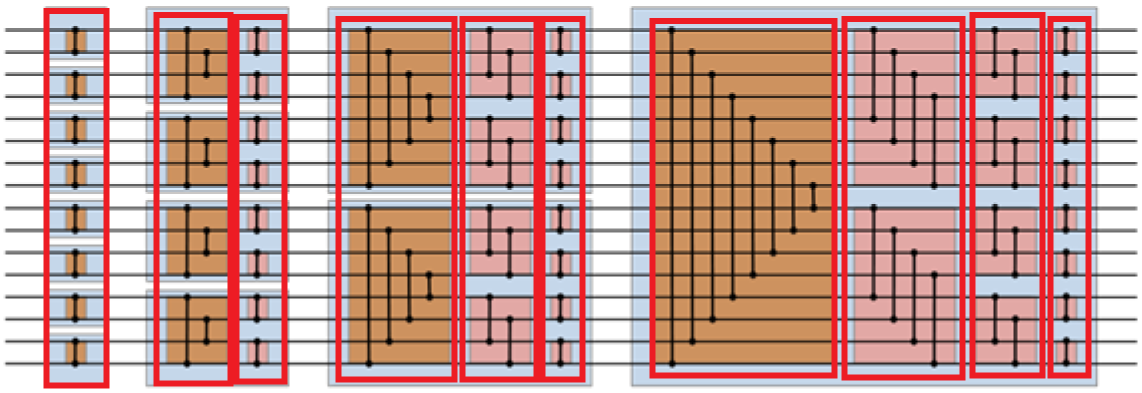


Figure 7

Therefore if this algorithm is run on a PRAM in parallel, its time complexity will be of O(lg^2(n)) as all n/2 comparisons in each stage can be done simultaneously.

Batcher Odd-Even Merge Sort

The principle idea behind this algorithm is the same idea used for merge and bitonic sorts. The odd-even merge algorithm will first run itself recursively on odd and then even elements and then it compares each odd element with its next even element. If the size of array is two, it just compares the first and second element. Figure 8 represents the batcher odd-even merge sort network of size 16.

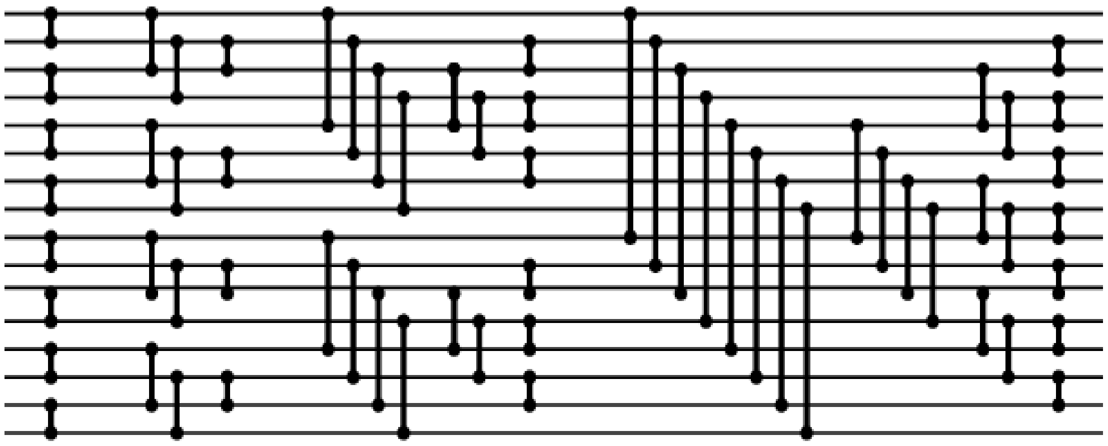


Figure 8

This algorithm can be implemented in a way to have a space complexity of O(1). It is also considered a stable sorting algorithm.

The time complexity of this algorithm when run sequentially is O(n*lg^2(n)) as the odd-even merge algorithm has a time complexity of O(n*lg(n)) and just like bitonic sort the time complexity of this algorithm when run in parallel on a PRAM is O(lg^2(n)). However, even though the time complexities of this algorithm and bitonic sort are the same, odd-even mergesort does fewer number of comparisons.

The comparisons in each red box in *Figure 9* can be done in parallel.

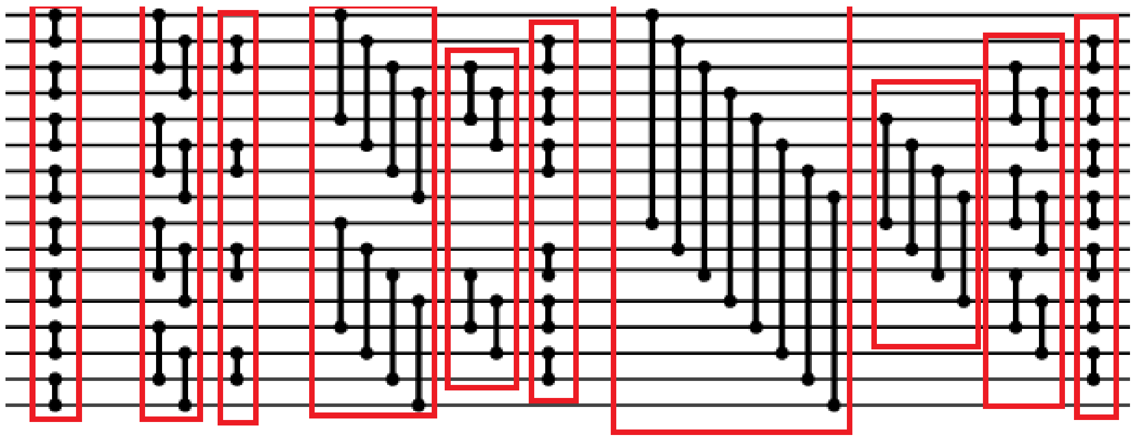


Figure 9

Non-Comparison Sorting Algorithms

All of the algorithms introduced in previous sections, have no assumptions about the data and put no limits on them. In other word, these algorithms only read the list elements through a single abstract comparison operation that determines which of the two elements should occur first in the sorted list. The only requirement is that the operator forms a total preorder over the data with transitivity (if a ≤ b and b ≤ c then a ≤ c) and connexity (for all a and b, a ≤ b or b ≤ c). These algorithms are known as comparison sorting algorithms and it can be proven that no comparison algorithm can have a lower time complexity than O(n*lg(n)) on a random access machine.

In contrast we also have non-comparison algorithms. These algorithms make other assumptions on their input which makes it possible to design significantly efficient algorithms with linear time complexity. One such algorithm is counting sort.

Counting Sort

This algorithm assumes that the input data are integer numbers whose maximum and minimum values are max and min respectively. Since the number of possible values for each element of the input is finite and these values are known in advance, this algorithm operates by counting the occurrence of each possible value in the input. The count array is then used to produce the output. *Algorithm 6* is a pseudocode expressing this algorithm.

```
counting_sort(arr, min, max):
    // initialization
    n = len(arr)
    k = max – min + 1
    count := initialize an all zero array of size k
    input := a copy of arr
    // counting
    for i = 0 to n – 1:
        count[input[i] – min]++
    // computing prefix sum of the count array
    for i = 0 to k-1:
        count[i] += count[i – 1]
    // sorting the array
    for i = n – 1 downto 0:
        count[input[i] – min]--
        arr[count[input[i] – min]] = input[i]
```

Algorithm 6

This algorithm, has a time and space complexity of O(n + k) where k is the number of possible values for a single input element. Moreover, this implementation of the algorithm makes it a stable sorting algorithm.

Implementations

The sequential implementation of each algorithm is almost the same as the pseudocodes presented in previous sections. The only difference is that for achieving better efficiency, none of the algorithms were implemented recursively and in fact all recursive algorithms were implemented using a bottom-up approach.

All of the mentioned algorithms were parallelized using both OpenMP and CUDA except bubble sort and insertion sort as these algorithms are not efficiently parallelizable.

CPU-Parallel Implementations

Due to the straight forwardness of batcher odd-even merge, bitonic and brick sort, parallelizing them using OpenMP library is quite simple. All that is needed is to use an omp parallel for directive before the loops that are responsible for comparing and swapping inputs of the same stage.

For simple merge sort algorithm, both insertion sort and merge algorithms are done sequentially as they cannot be efficiently parallelized. However, since at each time we may be able to perform these algorithms on different parts of the input, we can use this to our advantage to parallelize these different runs.

As counting sort has multiple stages, each stage of the algorithm has been parallelized independently. The initialization of count and input arrays can be easily done simultaneously by adding an omp parallel for directive before their loops. For counting the number of occurrences we can also use an omp parallel directive; however, to avoid race condition, the increment operation for count array shod be done atomically.

Parallelizing prefix sum can be a bit tricky as there is a clear data dependency in the flow of the algorithm. Therefore, the count array is divided into P equal sized sections where P is the number of processors. Then each processor will compute the prefix sum of its own section.

Then the master thread will construct a temp array of size P and stores the sum of each section in its respective index. The prefix sum of this temp array is then computed by the master thread. As there are few CPU cores, this computation will not impose a noticeable overhead.

At the end, each core will add its respective value in temp array, to all of its section's elements.

At last, the final stage is also run in parallel. To make sure race condition will not happen, k locks are initialized at the start of the algorithm and then these locks are used to protect the critical section that we have in this stage.

GPU-Parallel Implementations

The parallelization method for all of the algorithms in this section is the same as the last section except the counting sort. The reason for this is that in the prefix sum computation stage, we assumed that the number of cores are few hence this method is not useful when using GPUs for parallelization.

The first two stages of the algorithm is just like the CPU-parallelized implementation. The only difference is that in this implementation there are no input arrays.

Two compute the prefix sum of the count array, the count array is passed to a kernel for computing the prefix sum in each block. In this implementation, each block has 128 threads and assumes that it has an input array of size 128. For computing the prefix sum of these 128 elements, each block will move all of its respective elements into a shared array of size 128. If the number of elements are less than 128, the rest of the shared array will be filled with 0.

Then a reduction algorithm is used and in this process, some of the output elements can be computed. Then by using the already computed elements, the algorithm can compute the rest of the output elements.

At the end of the algorithm, one thread from each block will store the sum of all elements of that block in a respective index of an array of size ceil(k/128) called sum array.

This algorithm is illustrated for an array of size 16 in *Figure 10*.

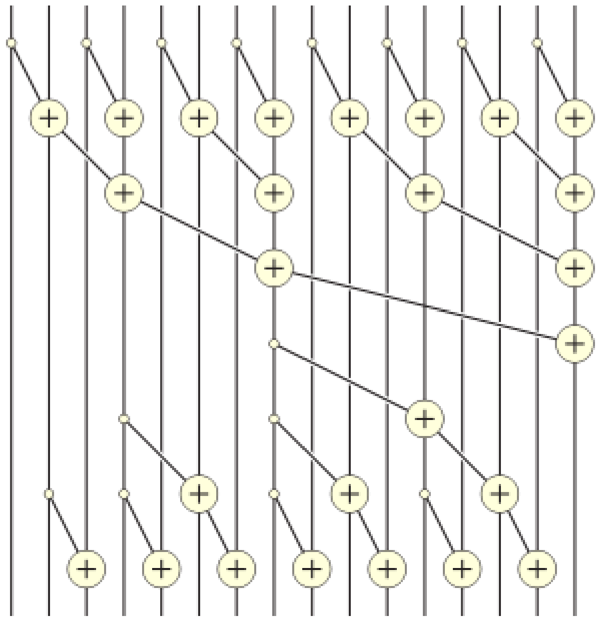


Figure 10

Just like sorting networks, each vertical line represents a wire which carries a signal. The result of each operation is stored on the wire which has the operator's sign on it. The input signals are entered from the upper part of the network while the output signals are outputted from the bottom of the network.

After this the prefix sum of the sum array is computed recursively and then another kernel adds the sum of each block to the elements of the next blocks. This specific algorithm has a time complexity of O((k/P)*lg(k)) where P is the number of processors and k is the size of the input array.

After the prefix sum of the count array is computed, each index of the output array is assigned to a GPU thread. Each thread then uses binary search on the count array to find the value which should be putted in its respective index of the output array. This algorithm has a time complexity of O((n/P) * lg(k)).

Overall this implementation of counting sort has a time complexity of
O(n/P + ((n + k)/p)*lg(k)) = O((n+k)/P * lg(k)).

Comparison

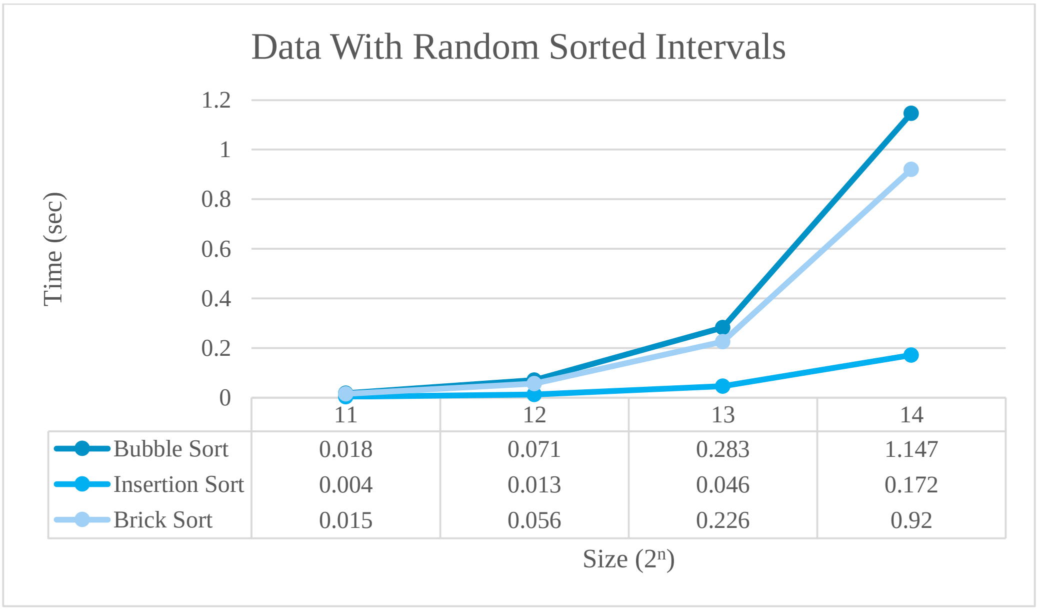
For parallelizing these algorithms using OpenMP, 8 threads have been used. Also, for evaluating sequential and CPU-parallelized algorithms' runtimes, omp_get_wtime api has been used inside each sorting function as the first and last instructions. For evaluating GPU-parallel algorithms' runtimes, cudaEvent_t has been used.

O(n^2) Complexity Class Algorithms

To evaluate the performance of bubble, insertion and brick sorts when run sequentially, these algorithms were run on three different types of float arrays. The first type of arrays may have small sorted parts which can be either sorted in ascending or descending order. The second type arrays are nearly sorted in ascending order and the third type are nearly sorted in descending order. By nearly sorted it means that the arrays were sorted after being generated and then 10 percent of their elements were randomly chosen and swapped with some other random elements.

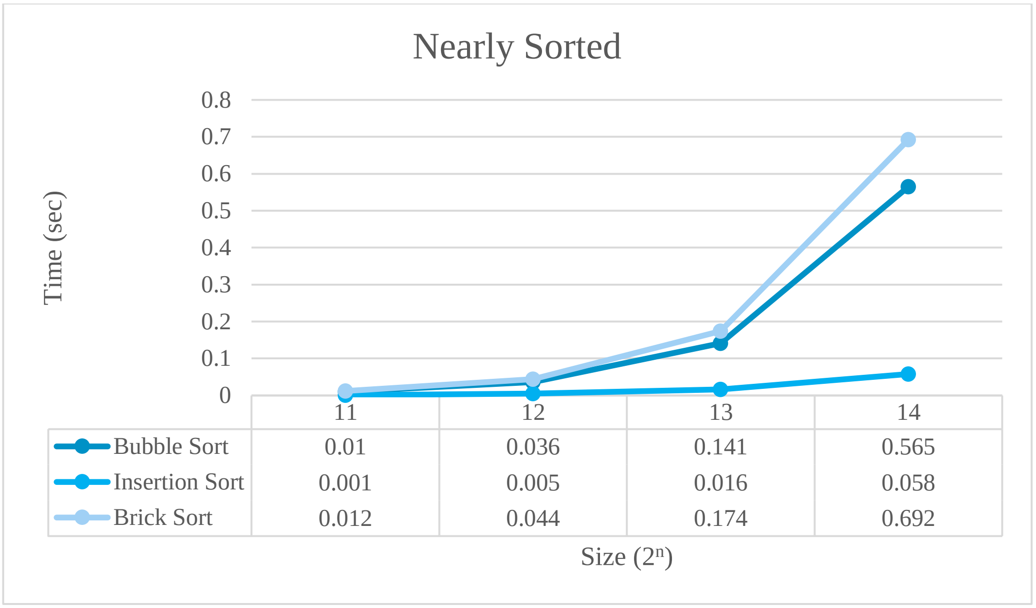
Figure 11, *Figure 12* and *Figure 13* demonstrate the tests run on these types of arrays respectively.

The horizontal axis represents the size of each array as for each number like n, the size of the corresponding array would be 2^n. The vertical axis represents the average run time of the algorithms in seconds.



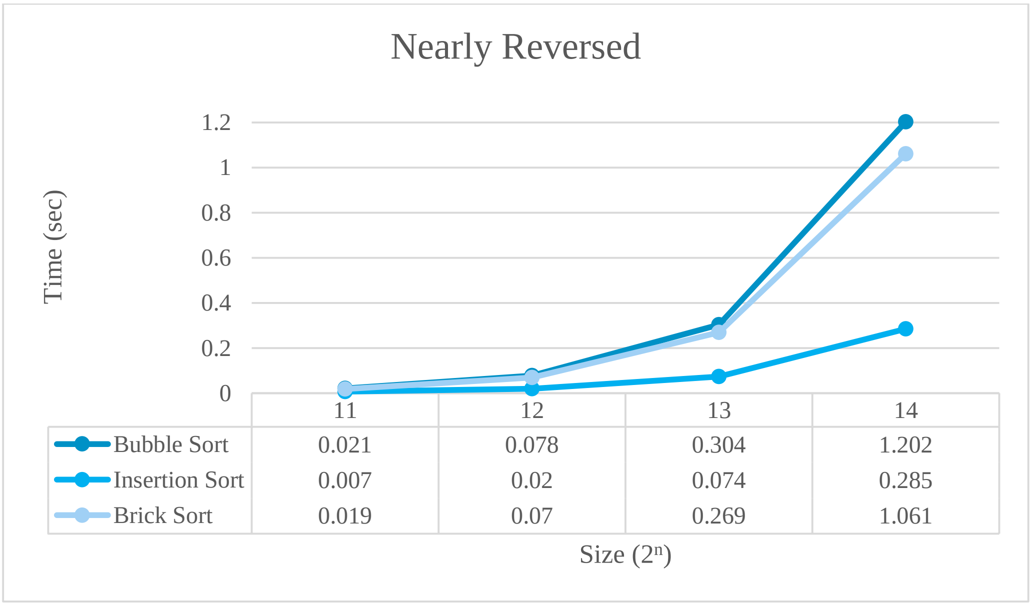
	11	12	13	14
Bubble Sort	0.018	0.071	0.283	1.147
Insertion Sort	0.004	0.013	0.046	0.172
Brick Sort	0.015	0.056	0.226	0.92

Figure 11



	11	12	13	14
Bubble Sort	0.01	0.036	0.141	0.565
Insertion Sort	0.001	0.005	0.016	0.058
Brick Sort	0.012	0.044	0.174	0.692

Figure 12



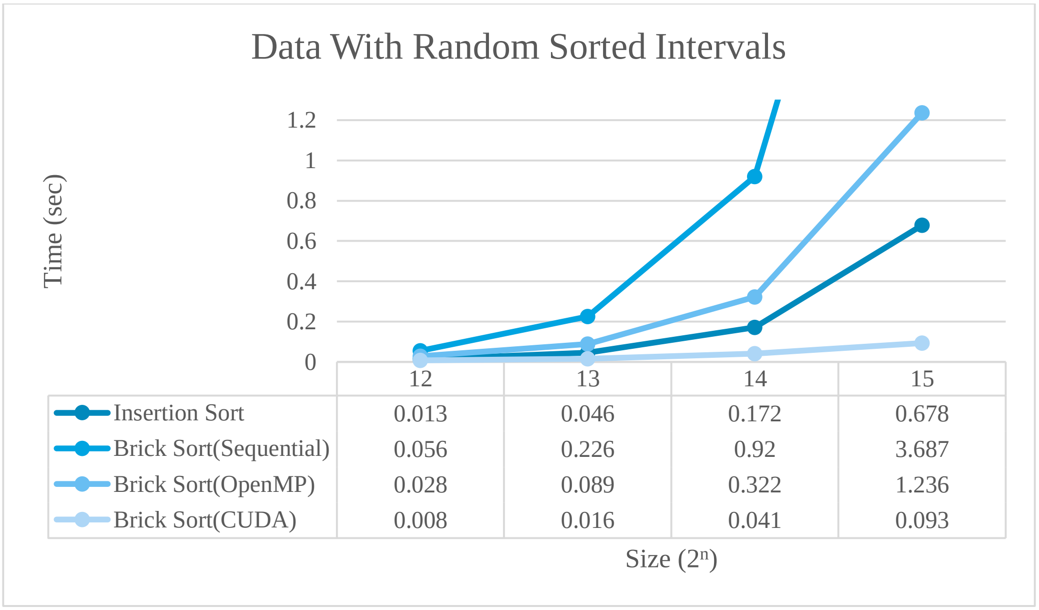
	11	12	13	14
Bubble Sort	0.021	0.078	0.304	1.202
Insertion Sort	0.007	0.02	0.074	0.285
Brick Sort	0.019	0.07	0.269	1.061

Figure 13

It can be seen from these figures that insertion sort totally outperforms the other two algorithms when run sequentially. Moreover, generally the odd-even sort algorithm is more efficient than bubble sort but depending on our data, sometimes bubble sort may outperform sequential odd-even sort.

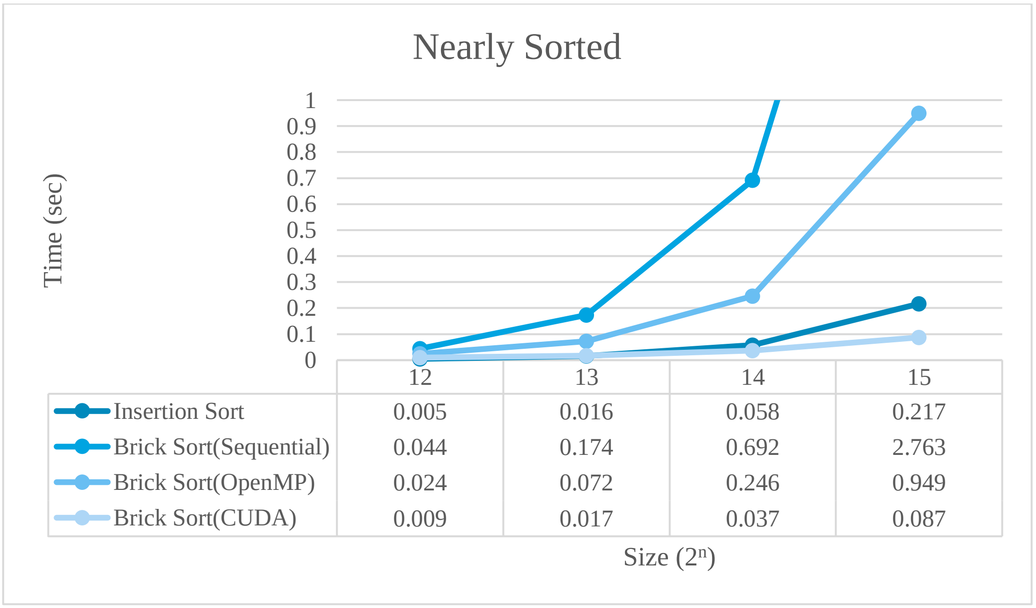
From these experiments we can conclude that we should compare the runtime of parallel odd-even sort with insertion sort to see whether our parallelization is useful or not.

Figure 14, *Figure 15* and *Figure 16* compare insertion sort with sequential, CPU-parallel and GPU-parallel versions of odd-even sort.



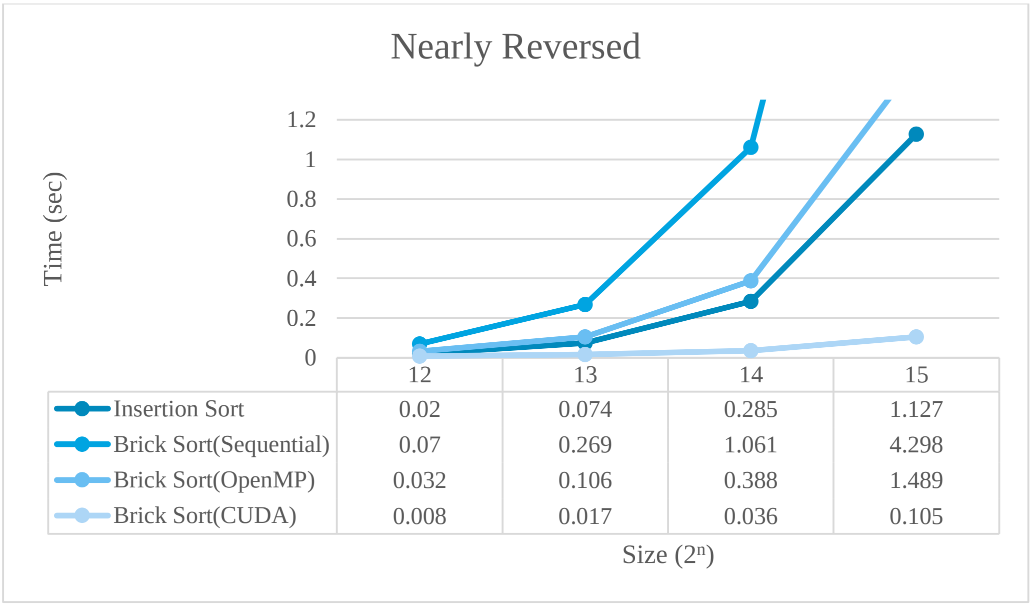
	12	13	14	15
Insertion Sort	0.013	0.046	0.172	0.678
Brick Sort(Sequential)	0.056	0.226	0.92	3.687
Brick Sort(OpenMP)	0.028	0.089	0.322	1.236
Brick Sort(CUDA)	0.008	0.016	0.041	0.093

Figure 14



	12	13	14	15
Insertion Sort	0.005	0.016	0.058	0.217
Brick Sort(Sequential)	0.044	0.174	0.692	2.763
Brick Sort(OpenMP)	0.024	0.072	0.246	0.949
Brick Sort(CUDA)	0.009	0.017	0.037	0.087

Figure 15



	12	13	14	15
Insertion Sort	0.02	0.074	0.285	1.127
Brick Sort(Sequential)	0.07	0.269	1.061	4.298
Brick Sort(OpenMP)	0.032	0.106	0.388	1.489
Brick Sort(CUDA)	0.008	0.017	0.036	0.105

Figure 16

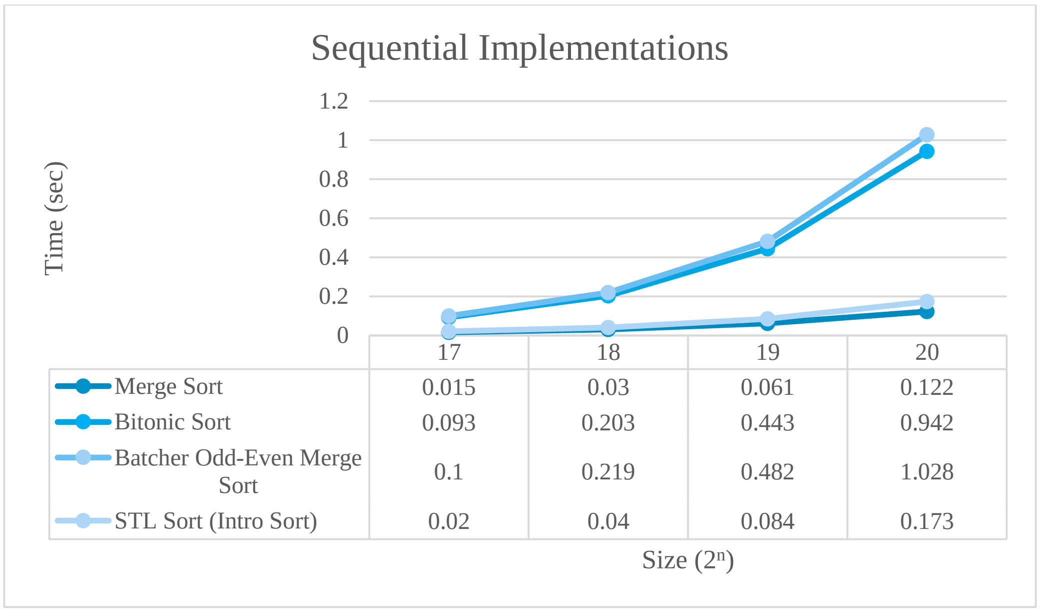
It can be understood from these figures that even CPU-Parallelized odd-even sort has lower performance compared to the sequential insertion sort. However GPU-Parallelized odd-even sort preforms better than insertion sort even if the array is nearly sorted. The speed-up of the represented parallel implementation for arrays of size 32768 are given in *Table 1*.

Table 1

Array Type	Data With Random Sorted Intervals		Nearly Sorted		Nearly Sorted In Reverse	
Parallelization Tool (Brick Sort)	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
Sequential Brick Sort	2.98	39.65	2.91	31.76	2.87	40.93
Insertion Sort	-	7.29	-	2.49	-	10.73

Optimal Complexity Class Algorithms

As mentioned before, O(n*lg(n)) is the best time complexity a comparison sorting algorithm may have. However, algorithms such as bitonic or batcher odd-even merge sort pay the price of being highly parallelizable on GPU by having an additional lg(n) multiplied in their complexity order, making them O(n*lg^2(n)). *Figure 17*, compares the runtime of bottom-up hybrid version of merge sort with bottom-up bitonic sort, batcher odd-even merge sort and at last the C++ standard sort.



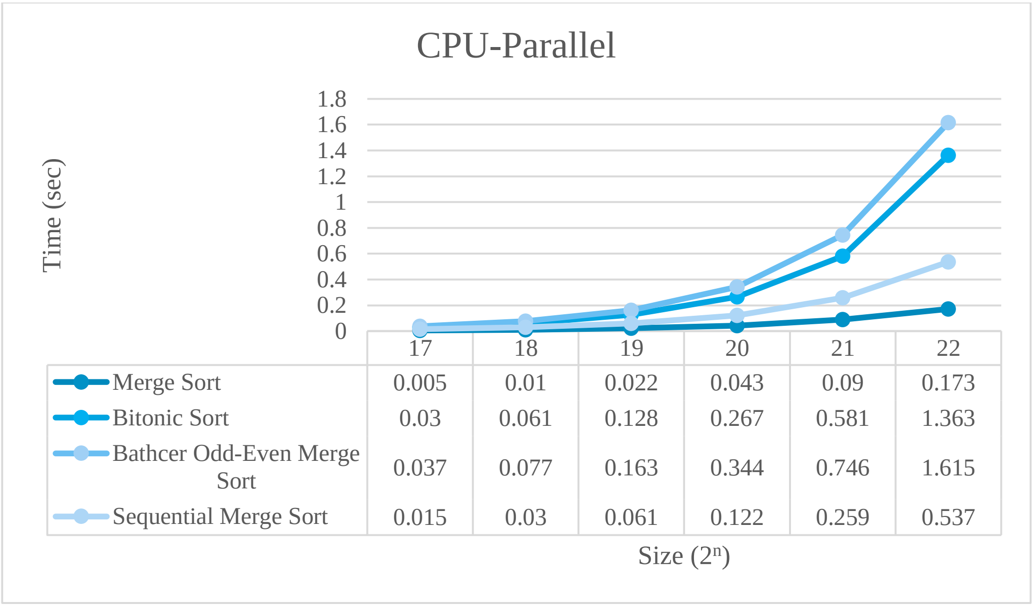
	17	18	19	20
Merge Sort	0.015	0.03	0.061	0.122
Bitonic Sort	0.093	0.203	0.443	0.942
Batcher Odd-Even Merge Sort	0.1	0.219	0.482	1.028
STL Sort (Intro Sort)	0.02	0.04	0.084	0.173

Figure 17

As it can be seen from *Figure 17*, bitonic sort and batcher odd-even merge sort have almost the same runtime with bitonic sort being surprisingly more efficient, and this implementation of merge sort has performed just a bit better than the C++ standard library sort making it the best sequential sorting algorithm in this project.

It can also be noted that sequential hybrid bottom-up merge sort is more than 6 times faster than bottom-up bitonic and batcher odd-even merge sort.

Figure 18, compares parallelized version of the same algorithms with sequential merge sort which was the best sequential algorithm in this project.



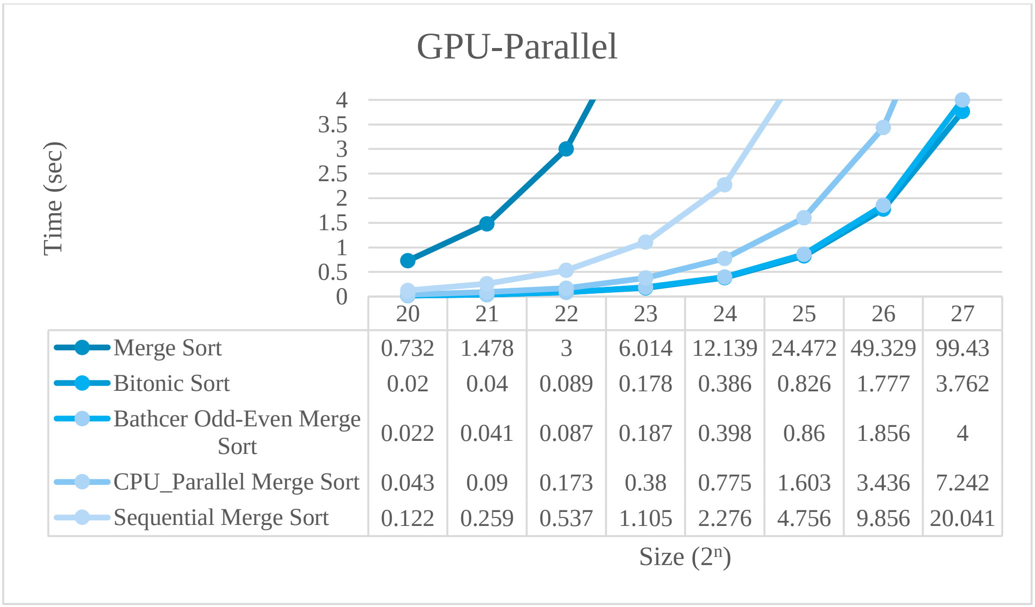
	17	18	19	20	21	22
Merge Sort	0.005	0.01	0.022	0.043	0.09	0.173
Bitonic Sort	0.03	0.061	0.128	0.267	0.581	1.363
Bathcer Odd-Even Merge Sort	0.037	0.077	0.163	0.344	0.746	1.615
Sequential Merge Sort	0.015	0.03	0.061	0.122	0.259	0.537

Figure 18

It can be understood from *Figure 18* that this implementation of sequential merge sort is more than two times faster than CPU-parallel bitonic and batcher odd-even merge sorts. We can also see that this parallel merge sort implementation is more than 6 times faster than the other two parallelized algorithms.

Overall, CPU parallelized version of this merge sort is the best CPU-parallelized sort between the mentioned sorts. However, *Figure 19*, which compares GPU-parallelized version of these algorithms with sequential and CPU-parallel implementations of this merge sort, shows that this is not the case for GPU-parallel implementation of merge sort.

The reason why GPU-parallelized merge sort performs even worse than sequential merge sort, lies in the parallelization approached discussed in the GPU-parallelized Implementations section of this report. The mentioned approach works quite fine on CPU as proved by *Figure 18*, because that it still has a complex control flow which is fine when run on a CPU but since GPUs are weak at control flow, the overhead of this parallelization approach on GPU far surpasses the speed up that it gives us.



	20	21	22	23	24	25	26	27
Merge Sort	0.732	1.478	3	6.014	12.139	24.472	49.329	99.43
Bitonic Sort	0.02	0.04	0.089	0.178	0.386	0.826	1.777	3.762
Bathcer Odd-Even Merge Sort	0.022	0.041	0.087	0.187	0.398	0.86	1.856	4
CPU_Parallel Merge Sort	0.043	0.09	0.173	0.38	0.775	1.603	3.436	7.242
Sequential Merge Sort	0.122	0.259	0.537	1.105	2.276	4.756	9.856	20.041

Figure 19

In contrast, we can see that GPU-parallelized bitonic sort and odd-even merge sot have been quite efficient.

Table 2, demonstrates the speed up of each parallelized implementation against the sequential implementation of merge sort and the algorithm itself for the array with length 2^{20}.

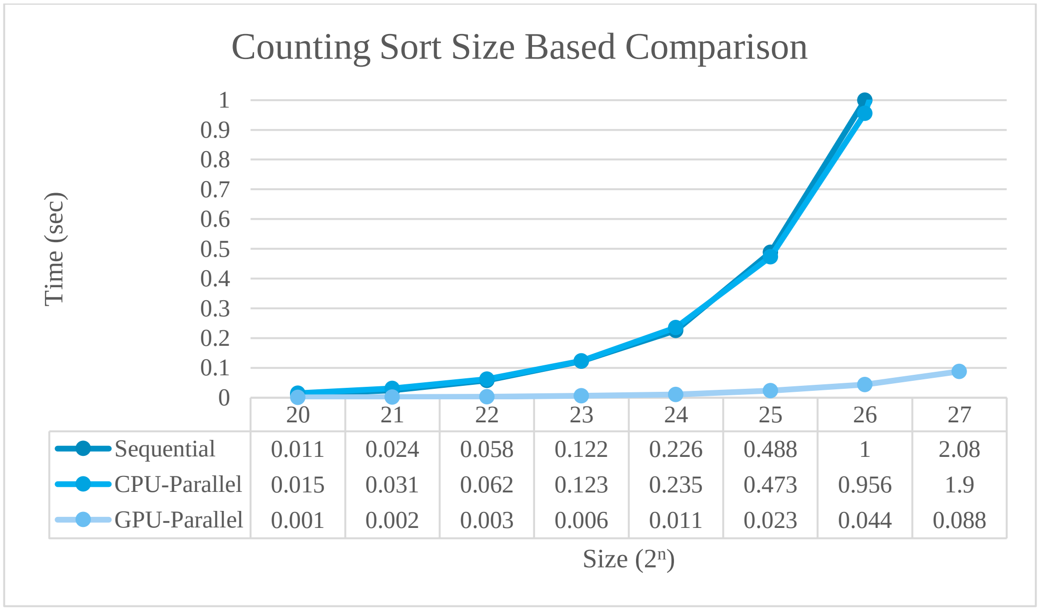
Table 2

Parallel Implementations	Merge Sort		Bitonic Sort		Batcher Odd-Even Merge Sort	
Parallelization Tool	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
Merge Sort	2.84	-	-	6.1	-	5.55
Bitonic Sort	-	-	3.53	47.1	-	-
Batcher Odd-Even Merge Sort	-	-	-	-	2.99	46.73

Non-Comparison Algorithms (Counting Sort)

When the boundaries for an integer array values are known beforehand, counting sort may perform great for sorting the array hence parallelization of this algorithm can lead to an efficient sorter.

Figure 20 compares the runtimes of sequential and parallel implementations of counting sort on arrays with different length. The arrays elements are between -50000 and 50000.

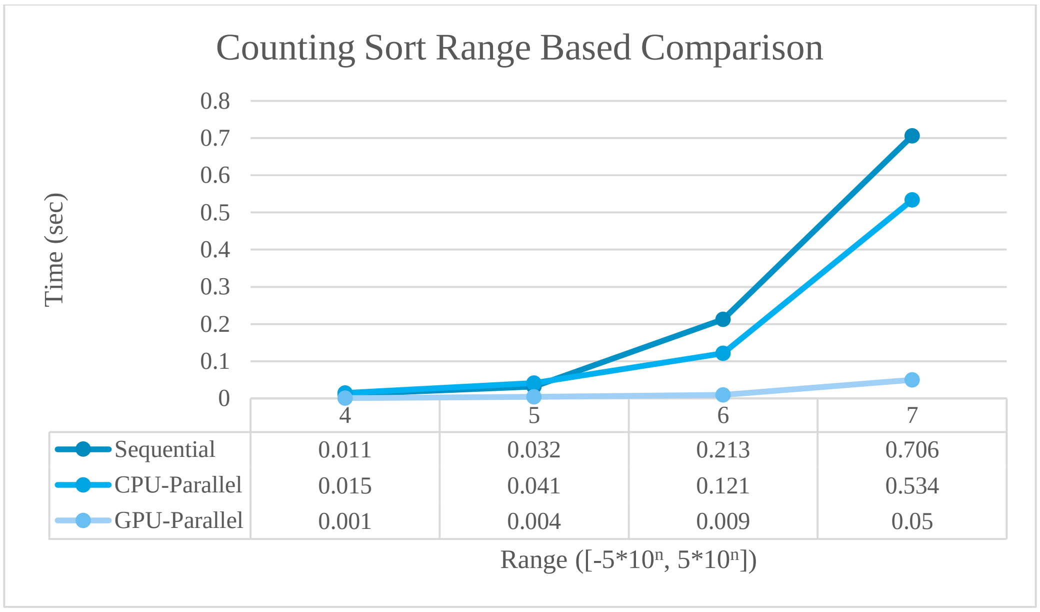


	20	21	22	23	24	25	26	27
Sequential	0.011	0.024	0.058	0.122	0.226	0.488	1	2.08
CPU-Parallel	0.015	0.031	0.062	0.123	0.235	0.473	0.956	1.9
GPU-Parallel	0.001	0.002	0.003	0.006	0.011	0.023	0.044	0.088

Figure 20

As it can be seen, despite the poor performance of this CPU-parallel implementation of counting sort, the GPU-parallel implementation is quite efficient.

Figure 21 compares the same runtimes on arrays of length 2^{20} and with elements in different ranges. It can be concluded from this figure that the CPU-parallel implementation mostly parallelizes operations that depend on the range of data. It can also be noted that the GPU-parallelized version has worked quite efficiently in this case as well.



	4	5	6	7
Sequential	0.011	0.032	0.213	0.706
CPU-Parallel	0.015	0.041	0.121	0.534
GPU-Parallel	0.001	0.004	0.009	0.05

Figure 21

The speed-ups of these implementations on different arrays are shown in *Table 3*.

Table 3

Parallelization Tool	OpenMP	CUDA
Size(2^{27})	1.09	23.64
Range([-5e7, 5e7])	1.32	14.12

Conclusion

Overall, we can conclude that using a good sequential algorithm is more efficient than parallelizing a slower but highly parallelizable algorithm on 8 cores of the CPU used for these tests. Moreover, CUDA parallelization of slower but highly parallelizable algorithms can give us better performance and can be useful.