CS 700B Master's Project Report

Comprehensive Performance Comparison of Sequential and different Parallel Sorting techniques on CPU & GPU

Submitted to
the Department of Computer Science
Ying Wu College of Computing
New Jersey Institute of Technology

in partial fulfillment of
the requirements for the degree of
Master of Science in Computer Science

Submitted by
Aarjavi Dharaiya
ard22@njit.edu
December 17, 2021

Project Advisor: ____________________

Signature: ________________

Date: __/__/____

Table of Contents

Chapter 1 Introduction and background

Sorting is a fundamental operation that is performed by most computers [1]. It is a computational building block of basic importance and is one of the most widely studied algorithmic problems [2]. Sorted data is easier to manipulate than randomly-ordered data, so many algorithms require sorted data. It is used frequently in a large variety of useful applications.[3]. In this modern era of technology where fields like Internet of Things, Big Data, Machine Learning, AI, Medical Research, etc are becoming more and more widespread, the amount of data that is to be managed and processed is increasing at a humongous rate. In such times, the traditional sequential sorting algorithms would not be performant in processing the gigantic data. Also as we approach the end of Moore's law, no further improvement in a single core performance will soon be possible. With these restrictions, parallelization of sorting algorithms will soon become paramount.

The hardware limitation of a single core can be overcomed by using multiple CPU cores in parallel. Furthermore, the entire world of parallel computing endured a change when GPUs were gradually embraced in today's high-performance computing cluster[4]. GPUs offer massive computing capacity with thousands of cores working in parallel. Parallelization of Sorting algorithms is possible by mainly 3 different methods

1) MultiThreading - single process, multiple threads running on multiple CPU cores
2) Multiprocessing - multiple processes running on multiple CPU cores
3) Parallelization over multiple GPU cores

This project will mainly focus on the design and performance comparison of parallel sorting algorithms running on multiple GPU cores. Different techniques to achieve better parallelization over GPU, like cuda streams will also be explored. The project will cover implementation and analysis of mainly 2 sorting algorithms:-

1. Odd-even transposition sort
2. Bitonic sort

Chapter 2 Project Description

2.1 Project Definition

The project aims to implement 2 sorting algorithms - odd-even transposition sort and bitonic sort in below 3 different modes and perform a comprehensive performance comparison of all these 3 techniques

1) Traditional sequential implementation on single core CPU
2) Multiple GPU cores
3) Repeat *'2'* not with direct algorithm implementation but with a modified version for each algorithm where it runs a bucket of keys per cuda stream.

The modified bucket of keys algorithm is explained in chapter : 2.3

2.2 Sorting algorithms

Let us first discuss a basic building block which will be used in all sorting networks - Comparator Circuit.

2.2.0 Comparator Circuit

It takes two elements as input (x,y) and returns them in sorted order., i.e (x,y) if x<=y otherwise it returns (y,x).

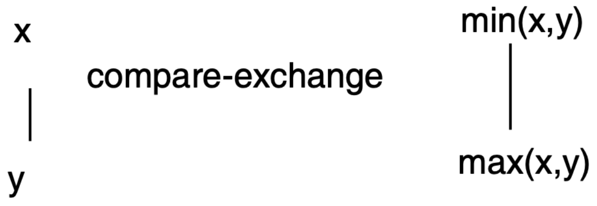


Fig 1 - Compare and Exchange Circuit [9]

Below sections explains in detail the sorting algorithms used in this project.

2.2.1 Odd - even Transposition Sort

Odd - even transposition sort is a parallel sorting technique. It is based on the sequential Bubble sort. It comprises two phases :- odd phase and even phase. In the odd phase, the odd indexed numbers are compared and exchanged with their adjacent numbers. Similarly in the even phase, the even indexed numbers are compared and exchanged with their adjacent elements. For an input of *n* elements there are approximately *n/2* comparisons in each round. This sorting algorithm takes *n* rounds to give the sorted output. Below is the example demonstrating this algorithm.

Unsorted Input :- 10,5,6,8,4,1

Step	Phase						
1	Odd	10	5	6	8	4	1
2	Even	5	10	6	8	1	4
3	Odd	5	6	10	1	8	4
4	Even	5	6	1	10	4	8
5	Odd	5	1	6	4	10	8
6	Even	1	5	4	6	8	10

Sorted Output :- 1,4,5,6,8,10

The same algorithm can be applied to sort in decreasing order with the comparison operator reversed.

In a unicore processor or a completely sequential execution, the algorithm will have time complexity of Theta(n^2). But in a multicore processor or in GPU implementation, the comparisons in a single round can be parallelized to achieve a time complexity of Theta(n).

2.2.2 Bitonic Based Sort

Bitonic Based Sort is a sorting network first devised by Ken Batcher [6]. A sorting network is a sorting algorithm where the sequence of comparisons (i.e. the order, direction, and number of comparisons) is data-independent and therefore the algorithm can be described through a static network of comparators, connected to each other with data lanes. The number of compare/exchange operations needed by bitonic based sort for sorting a sequence of length *n*, and thereby the (sequential) complexity, is $O(n \cdot \log^2 n)$, which is not optimal. The strength of bitonic sort is that it is a sorting network, which can be implemented directly in hardware, but which is also adequate for generic parallel architectures since it works in-place, requires less inter-process communication and can be naturally implemented in SIMD architectures as NVIDIA's GPUs. [13]

Definition:-

A bitonic sequence is a sequence of numbers $X0, X1 \ldots Xn-1$ with the following properties[14]:

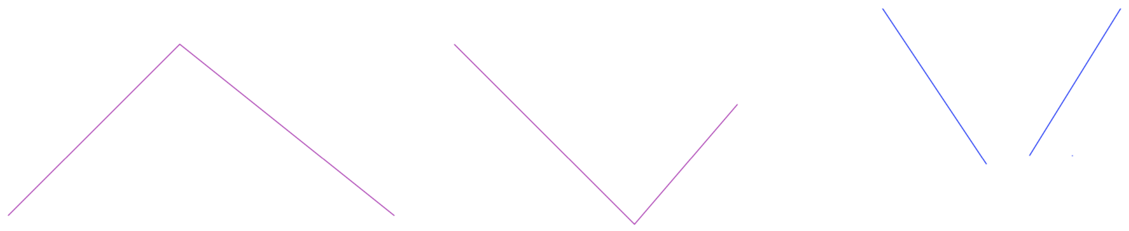
1. There exists an index i where $0 \le i \le n - 1$ such that,
   $$X0 \le X1 \le \ldots \le Xi \text{ and } Xi \ge Xi + 1 \ge \ldots \ge Xn-1$$
2. We may cyclically shift the Xi such that (1) is true.

Explanation[15] :-

In other words, the sequence is bitonic if after placing it on a circle we can partition the circle into a nondecreasing sequence and the following nonincreasing sequence.

Bitonic sequences are categorized into following two categories:-

Simple Bitonic Sequences:

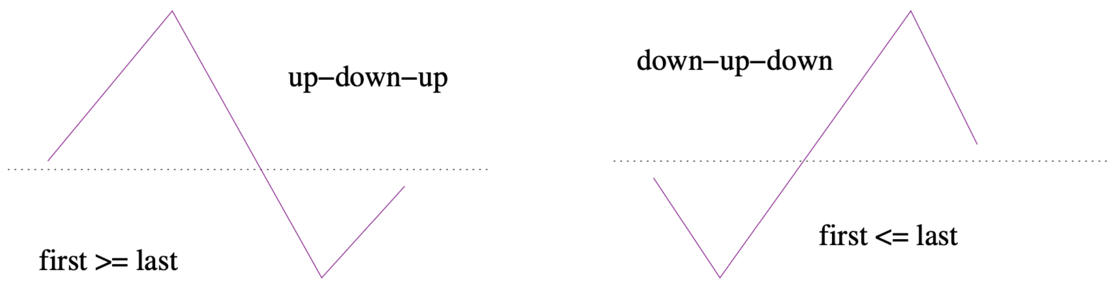


Examples :

1 2 3 4 5 6 3 2 1

8 7 6 3 9 10 15

General bitonic sequences



Examples :

4 5 6 3 2 1 1 2 3

6 3 9 10 15 8 7

<u>**Bitonic Based Sort Architecture Modules**</u>[20]

For better understanding of the Bitonic Based Sort Algorithm, let us first discuss the basic modules that are building blocks of the overall architecture.

<u>**Bitonic Sequence Sorting**</u>

Input:- Bitonic sequence of size *n*
Output:- Sorted sequence of size *n* (in non decreasing order)

The Bitonic Sequence Sorting Network takes as an input - a Bitonic sequence and in the first step divides it into two subsequences- an upper half and a lower half, such that it satisfies below two properties:

1. Each element in the upper half is less than or equal to every element in the lower half.
2. Both the sub sequences are themselves Bitonic sequences.

Recursively repeating the above procedure, we will get a sorted sequence as the output.

The Bitonic Sequence Sorting network consists of two main blocks:-

- Half Cleaner - HC(n)
- Full Cleaner - FC(n)

<u>Half Cleaner:-</u>

We construct a comparison network which connects every input line to its diagonally opposite line, i.e line i is connected to line $i + n/2$, $0 \leq i < n/2$. Such a network turns a bitonic sequence of 0’s and 1’s into two bitonic sequences of half the original size, one of which is clean (all output lines contain one kind of input either all 0 or all 1). We call such a network a half-cleaner and we denote it with HC(n). An HC(n) uses $n/2$ comparators and its depth is one.

<u>Full Cleaner:-</u>

We use the idea of half-cleaning recursively until we “clean” all the lines/wires. The first application of half-cleaning cleans half the input (which becomes a trivial bitonic sequence) and turns the other half into a bitonic sequence whose elements are smaller than (or equal to) the cleaned ones if it is the top-half and at least the cleaned elements if it is the bottom half.

A full cleaner results by using half cleaners for smaller and smaller line size networks. We start with HC(n) a half-cleaner for *n* wires. It is then followed by two HC(n/2), followed by four HC(n/4) and so on. This becomes a FC(n). An FC(n) uses log(n) levels of HC(.) Each level consists of *n/2* comparators.

Network:-

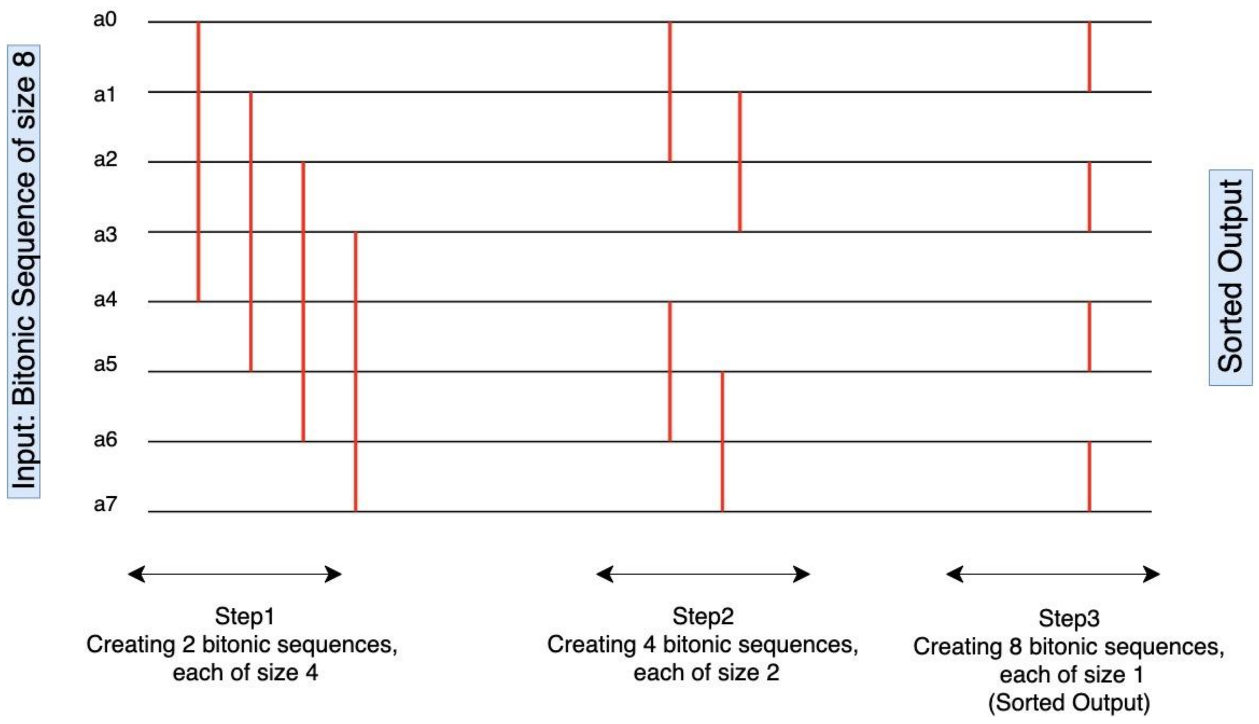

Input: Bitonic Sequence of size 8
a0
a1
a2
a3
a4
a5
a6
a7
Sorted Output
Step1
Creating 2 bitonic sequences, each of size 4
Step2
Creating 4 bitonic sequences, each of size 2
Step3
Creating 8 bitonic sequences, each of size 1
(Sorted Output)

Fig.2 Bitonic Sequence Sorting Network using FC(8)

As discussed in the algorithm description, in Bitonic Sequence Sorting, in the first step it runs HC(n), where it will compare the diagonally opposite points.Thus for an Bitonic sequence of size 8, we have:-

S = <a0,a1,a2,a3,a4,a5,a6,a7>

n = 8

n/2 = 4

HC(8):-

The diagonally opposite points are :

(a0 & a4); (a1 & a5); (a2 & a6); (a3 & a7)

As shown in the above figure, in step1 these diagonally opposite points are compared and exchanged, so as to form two bitonic sequences each of size 4.

Recursively, in step 2, it runs 2 HC(4), where these 2 bitonic sequences are further divided into 4 bitonic sequences, each of size 2. And in step 3, in the input there are 4 bitonic sequences, such that all elements in the sequence 1st < 2nd < 3rd < 4rth (Property 1 of Bitonic Sequence Sorting). Thus finally, it runs 4 HC(2), where just by one comparison step across all these 4 bitonic sequences, it will give the final sorted sequence in the output.

Merge Algorithm

Input :- Two sorted sequences, each of size *n/2*
Output:- Sorted sequence of size *n* (in non decreasing order)

The Merge Network takes as input - two sorted sequences of size *n/2* each, it merges them into one Bitonic sequence of size *n* and then produces as an output, a sorted sequence of size *n.*

Algorithm :-

To merge two sorted sequences into one Bitonic sequence, it just needs to reverse the second sequence and append it at the end of the first sequence. Then, sort the bitonic sequence using a Bitonic Sequence Sorting Network discussed in the previous section. The flow can be described as:

1. Take as input two sorted sequences, S1 & S2, each of size *n/2*.
2. Reverse the second sequence to form SR2 = Reverse{S2}
3. Append S1 and SR2 to form a bitonic sequence BS = <S1,SR2>
4. Sort BS using Bitonic Sequence Sorting Network to get the sorted output sequence.

For example, consider 2 sorted sequences:
S1 - 1,3,5,7,9
S2 - 2,4,6,8,10
Reversed S2 = SR2 = 10,8,6,4,2

By appending S1 and SR2, we get BS - 1,3,5,7,9,10,8,6,4,2.
It is clear that BS formed by the above method is a bitonic sequence. After this it just has to use the Bitonic Sequence Sorting Network from the previous section to achieve a sorted sequence in the output.

This algorithm can be realised using the below network:-

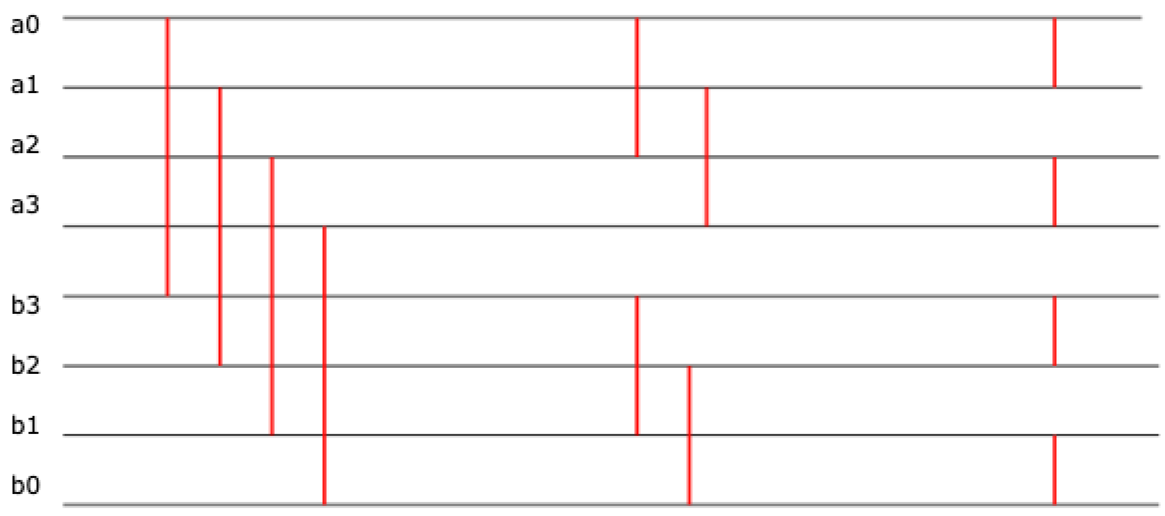


Fig.3 - Merging two Sorted Sequences [15]

Consider below two sorted sequences in the input:
S1 :- a0,a1, a2, a3
S2 :- b0,b1, b2 ,b3
As per the discussed algorithm we reverse S2 to get SR2 :- b3,b2,b1,b0 and then apply Bitonic Sequence Sorting Network which was described by the figure 2. Combining these two, we will get the Network as shown in fig. 3

Merge Network , FC’(n) :-

The network in Figure 3 takes as input a sorted sequence and another reverse sorted sequence. But the Merge Network takes as input, two sorted sequences.

We discussed the full cleaner circuit - FC(n) in chapter 2.2.2. In the first level of FC(n), it uses HC(n) where network line *i* is compared to line *i + n/2*. Projecting S1 and S2 as inputs to FC(n), the key of S2 in line *i+n/2* is the i-th largest key of S2. If S2 was to be reversed then the key would move to line *n−i+1*.

Thus in order to build a merging-network out of FC(n) we need to make sure that the input is a bitonic sequence rather than two sorted sequences! This means that we need to have HC’(n) at the first level where wire *i* and *n − i + 1* are fed into a comparator rather than *i* and *i* + *n/2*.

Thus, FC’(n) is an FC(n) where the first level HC(n) has been replaced with an HC’(n). The figure below gives the network for FC’(n).

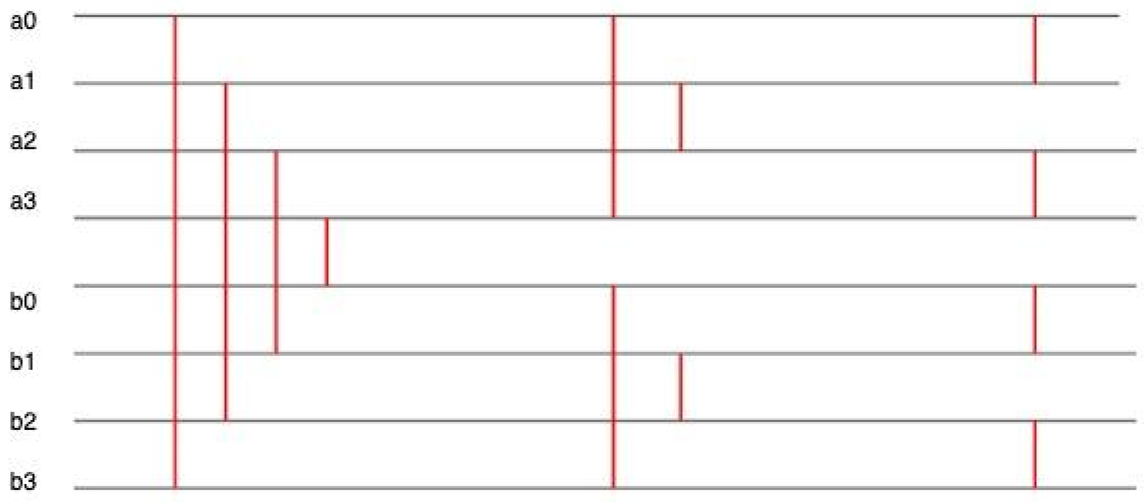


Fig.4 - Merge Network - FC’(n) [15]

Bitonic Based Sorter:-

Denoted as :- BS(n)
Input:- Unsorted sequence of size *n*
Output:- Sorted sequence of size *n* (in non decreasing order)

Algorithm :-

The Algorithm/architecture for Bitonic Based Sorting can be defined in below 2 main steps:-

1) **SORT** - Divide the *n* size unsorted input sequence into two subsequences, each of size *n/2*. Recursively sort these two subsequences, each by a Bitonic Based Sort Network of size *n/2 X n/2* in parallel.
2) **MERGE** - Use the Merge Network, FC'(n) as described in the previous section, which takes as input, two sorted sequences of size *n/2* each, and produces the output of a sorted sequence of size *n* using a Bitonic Sequence Sorting Network.

Thus, a bitonic-based sorter BS(n) consists of two BS(n/2) that sort the first *n/2* and the last *n/2* keys respectively. At the output we have two sorted sequences. This structure is then followed by an FC'(n) that merges the two *n/2*-long sorted sequences. Building block wise

BS(n) = BS(n/2) + BS(n/2) + FC'(n).

Network :-

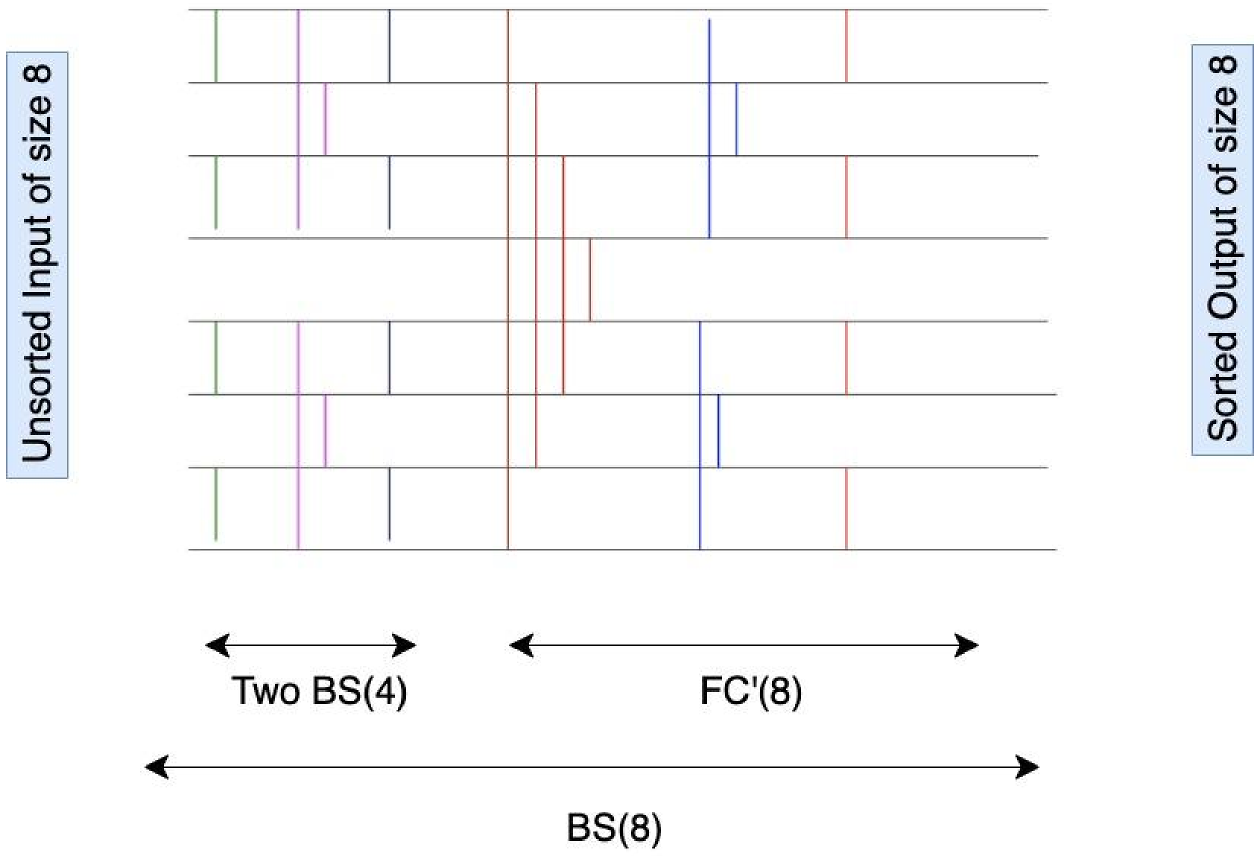


Fig.5 Bitonic Based Sorting Network

The above figure represents a 8X8 Bitonic Based Sorting Network. As explained in the above algorithm, we divide input of size 8 into two sequences of size 4, sort them individually and recursively using a 4x4 Bitonic Based Sort Network and finally apply a 8x8 Merge Network from figure 4 to give the final sorted output.

This network runs in 6 parallel steps(6 time units) and performs a total of 24 comparisons.

2.3 ‘Bucket of Keys’ Algorithm for parallelization

We will implement both the algorithms discussed above, but instead of the direct algorithm implementations it will use a 'bucket of keys' per processor/gpu-stream. For input of size *n*, normal implementation of odd-even transposition sort will take *n* rounds to produce sorted output. But, in this ‘bucket of keys’ algorithm, we divide *n* keys into *b* buckets(processors or gpu-streams). First we sort *b* buckets, each of size *n/b* keys and then merge the sorted buckets to get a final sorted output of size *n*. It needs to run *b* rounds of merge network to get the sorted output.

Algorithmic Flow for Odd-even Transposition Sort:-

1. Divide *n* input keys into *b* buckets, each having *n/b* keys.
2. **Sort:-** Sort the *b* buckets, each with *n/b* rounds of odd-even transposition sort. All the buckets can be sorted simultaneously or parallely by using gpu-streams in cuda.
3. **Merge:-** Previous step will produce an output of *b* sorted buckets, each of size *n/b* keys. Now, run *b* rounds of the merge circuit to get the final sorted output. The merge network takes in two buckets, each of size *n/b* keys and produces a sorted output of *2*n/b* keys. The merge network comprises two phases :- odd phase and even phase. In the odd phase, the odd indexed buckets are merged with their adjacent buckets. Similarly in the even phase, the even indexed buckets are merged with their adjacent buckets. To merge two buckets, we are using the Merge network discussed in Bitonic Based Sorting in chapter 2.2.2.

Example:-
Now, suppose we have these 16 keys to sort:-
60 30 40 50 35 45 65 42 14 24 34 44 13 53 63 23.

Normal implementation of odd-even transposition sort will take 16 rounds to sort these keys. But in this ‘bucket of keys’ algorithm, we divide these 16 keys in 4 buckets(processors or gpu-streams). In the first step, it sorts 4 buckets which are represented by 4 columns in the figure below. Then it runs 4 rounds of Merge network. In the first merge round, it will merge buckets 1 & 2 and buckets 3 & 4. In the second merge round, it merges buckets 2 & 3. Similarly, in the 3rd merge round, it again merges buckets 1 & 2 and buckets 3 & 4. And finally in the 4rth merge round, it will again merge buckets 2 & 3. This will produce the final sorted output as shown in the 5th step of the figure below.

```
 1   2    3    4    --->      1  2   3   4   ---> 1   2   3    4        -->        1   2    3    4        -->      1   2    3    4
30  35  14  13               30 45 13 34      30 13 45  34                  13  30   34   53             13   30   42  53
40  42  24  23               35 50 14 44      35 14 50  44                  14  35   44   60             14   34   44  60
50  45  34  53               40 60 23 53      40  23 60  53                 23  40   45   63             23   35   45  63
60  65  44  63               42 65 24 63      42  24 65  63                 24  42   50   65             24   40   50  65
```

This way parallelization can be implemented for the bitonic based sorting too. The code can launch multiple experiments to find an optimum value for the number of buckets which gives the best performance for a given sorting algorithm and a given *n* value.

Chapter 3 Introduction to GPU and CUDA

3.1 GPU Architecture

At the hardware level a GPU (device) is attached as a PCI3 card in the chassis of a traditional desktop PC that has a CPU (host). A GPU consists of a number of streaming multiprocessors known in NVIDIA terminology as SM, SMX (Kepler), or SMM (Maxwell) in different architecture generations (Kepler, Maxwell of NVIDIA). Generally we shall be using the term SM unless we describe a specific architecture and then we might use the specific name for an SM (e.g. SMX for Kepler). Multiple GPUs may be on the same card and can communicate directly with each other (no host utilization).

An SM is similar to a traditional ’CPU core’ but has hardware support for multiple streaming processors (’SM cores’ or functional units) and thus it is a highly multithreaded coprocessor to the accompanying CPU (host). It executes many threads in parallel on several multiprocessor cores. Thus we may have multiple GPUs containing multiple SMs that contain multiple SM cores that execute multiple threads each. Multiprocessors execute in parallel and asynchronously. Thus threads residing in one multiprocessor cannot send data into threads of another multiprocessor.

Each SM multiprocessor has registers (register file), a data cache (to be called L1 even if it differs from a CPU’s L1 cache) and shared memory that is limited in size (tens of kilobytes) and that it is shared by all the cores of the SM.

An L2 memory might be (and currently is) available and it is shared among all SM multiprocessors and their cores (and their threads). No cache coherence is available. The shared memory is organized in 32, 4B banks. Successive words are accessed through different banks of the shared memory

If multiple threads use the same bank to access different words, serialization takes place If those multiple threads access the same word however, a multicast takes place and the access is completed in one fetch. In such an architecture, L1 serves as a victim cache/spill memory for the registers. If a register does not contain the information but it is in L1, a 128B transfer is initiated; if data is not in L1 but in L2, then a 32B transfer is initiated. It is not a very good practice to use the L1 cache of a GPU the same way an L1 is being used in a CPU i.e. to cache (to move into faster memory) a block of memory that is to be used in an imminent computation. This is because 100s or 1000s of threads spread over all cores of an SM would be competing for the L1 cache of that SM. In fact it would be better to use the shared memory as the latter is shared among all the threads of an SM; moreover no eviction will ever take place from shared memory!

Global memory is faster than the host CPU memory (2-3 times on the average) but it is 100-300 times slower than the registers of the SM multiprocessors and their cores, and at least 5-30 times slower than L1 cache and the shared memory and 2-3 times slower than L2 cache . Communication between host and device is through PCI3 at more than 10GB/s and up to 85GB/s with close to 800ns latency or so.

3.2 CUDA

When a program is written for a CUDA-enabled device (GPU) it is usually written at the host and compiled by the NVIDIA framework/compiler (nvcc) that determines what part is going to be executed at the host and thus invokes the host's compiler infrastructure or what part is going to be executed by the CUDA device and thus the device's compiler infrastructure is to be used.

The host code executed at the host can include sequential (serial) and parallel code. The device code is a parallel function written in CUDA. A program in CUDA operates as follows: (1) data are copied from host (CPU) memory into device (GPU) memory, (2) the device code is then executed, and while this is being done, device memory is being used and cached on (device) chip, and (3) when the computation is completed results are then transferred from the device (GPU) memory back to the CPU memory. CUDA programs are SIMT (an instance of SIMD), where SIMT stands for Single Instruction Multiple Thread. A single instruction in CUDA is issued for a group of a fixed number of threads at a time. This group of threads is called a warp.

Threads in CUDA are organized hierarchically and have their own program counters (PC) and registers.

A thread belongs to a block of threads and blocks of threads are organized into grids. Thus, threads are arranged into a one, two, or three dimensional grid of identically shaped blocks. The geometry of a block is fixed in one, two or three-dimensions. All blocks must have the same dimensions. The number of threads in a ”kernel” is thus the number of blocks times the number of threads per block. When a GPU computation is launched one needs to specify the dimensions of the grid, and the dimensions of the block defining the grid. In NVIDIA CUDA this information is provided by a special execution configuration syntax (≪< ≫>). The (maximum) sizes/dimension values of blocks and grids depend on the particular architecture. All threads share the same (”global” or ”main” aka ”DRAM”) memory. Threads within the same block share a very fast ”shared memory” that is small in size, and within the SM an L1 (data) cache. Within a given block threads share an instruction stream. When there is divergence the different branches are run sequentially (aka serially) while they are divergent. When they converge, parallelism is restored.

*This chapter is taken from [17]

3.3 Concurrency in CUDA by Streams

Kernels in CUDA can be launched sequentially or in parallel. To launch multiple kernels in parallel, it needs to make use of CUDA streams.

CUDA applications manage concurrency by executing asynchronous commands in streams.

- Each stream is a sequence of commands that execute in order.
- Different streams may execute their commands concurrently or out of order with respect to each other.

What is a Stream?

In CUDA, stream refers to a single operation sequence on a GPU device. We can run multiple kernels on different streams concurrently. Typically, we can improve performance by increasing the number of concurrent streams by setting a higher degree of parallelism.

To achieve concurrency by CUDA streams, following points have to be taken care of:-

- Kernels need to be assigned to different streams and invoked asynchronously.
- Kernel launch and dependent memory transfer functions need to be assigned to the same stream.
- If only one kernel is invoked, the default stream, stream0, is used.
- When you execute asynchronous CUDA commands without specifying a stream, the runtime uses the default stream.
- If *n* kernels are invoked in parallel, *n* streams need to be used.
- Pinned memory must be used.
- Asynchronous memory transfer API functions must be used.

The concurrent streams are independent, which means that streams neither communicate nor have any inter-stream dependency. However, different streams may have different execution times (asynchronous), and we cannot guarantee that all streams will complete at the same time. To ensure all streams are synchronized, use a synchronization barrier.

Synchronizing Streams

There are two types of stream synchronization in CUDA - Explicit & Implicit. A programmer can place the synchronization barrier explicitly, to synchronize tasks such as memory operations. Some functions are implicitly synchronized, which means one or all streams must complete before proceeding to the next section. In our code we use explicit synchronization using cudaDeviceSynchronize().

*This chapter is written with reference from [18] & [19].

Chapter 4 Performance Evaluation & Comparison

This chapter presents the experimental setup and the results for all the experiments performed for both the algorithms implemented in this project. It will also discuss performance comparison across different algorithmic implementations and input types. The chapter also gives an idea about how every phase of this project is developed to give a better performance as compared to its previous phase.

4.1 Experimental Setup

The entire experimental setup is developed with automated scripts that makes input generation, launching experiments and analysing results a very quick and easy process. It is described briefly below.

Input Sequence :- Input sequence is generated from a script : GenerateData.c which takes in 2 arguments :

1. *n* - number of keys
2. Input_Type - type of input sequence needed, which are:
    a. RS - Reverse Sorted
    b. RND - Random numbers in range of 1… *n*
    c. Natural Random numbers
    d. Sorted Sequence
    e. All keys with a constant same value.

The script GenerateData.c saves the generated input sequence in a file. All the experiments read this same file for getting its input sequence, so as to have a standard input for a proper algorithmic comparison across all the experiments.

Launching Experiments :- This project tests the two algorithms : odd even transposition sort and bitonic based sort for a variety of input combinations like different values of *n* and different types of input sequences. It is difficult to manually launch these experiments every time. Thus, I use an automated script - LaunchExperim.sh to launch all desired combinations of experiments. The script uses knobs for selecting types of input sequence and different values of *n*.

Analysing Experiments:- The last step of experimental setup is to analyse the generated results for performance comparison. I have an automated python script -

AnalyseResults.ipynb , which will read the output generated by LaunchExperim.sh and provide .csv files with tabular representation of data as illustrated by the tables below.

Compiler details :- All the C codes have been compiled using gcc/5.3.0 and cuda codes are ran using cuda/9.0.176

4.2 Experimental Results

For most of the experiments, we have used values of 32768, 1048576 and 33554432 for *n*. This is because *n* values smaller than 32768 gives running time in microseconds, which does not give a proper idea for performance comparison. And *n* values greater than 33554432, gives running time in hours for odd even transposition sort which increases the results collection time so much that it is not in the scope of this project span to record them.

All the tables uses below notation :-

- RND - Random Input sequence with values in the range 1 … *n* .
- RS - Reverse Sorted Input sequence with values starting from *n,* and going down till 1.
- oets - odd even transposition sort
- btn - bitonic based sort
- *n* - number of input keys to sort

4.2.1 Sequential Performance

The table below gives running time in seconds for input types: RND and RS, for the odd even transposition sort and bitonic based sort, with values of 32768, 1048576 and 33554432 for *n*. The code was compiled using gcc/5.3.0 with no compiler optimizations.

n	Input_type	Sorting Algo	Running Time(secs)
32768	RND	oets	1.9
32768	RS	oets	1.66
32768	RND	btns	0.03
32768	RS	btns	0.01
1048576	RND	oets	1991.01
1048576	RS	oets	1689.05
1048576	RND	btns	0.74
1048576	RS	btns	0.59
33554432	RND	btns	36.48
33554432	RS	btns	30.86

Table 1 - Sequential Running Times

With different experiments, it was observed that by using gcc compiler optimizations, it gives almost 3x-6x better sequential performance as compared to the above results. The table below shows results with following optimization options:-

1. O2All := -O2 -march=native -funroll-loops
2. O3All := -O3 -march=native -funroll-loops
3. O3unroll := -O3 -funroll-loops
4. O3only := -O3

The below explanation helps to understand some of the column names of the table

- min_seq = minimum of all the sequential running times obtained with 4 compiler optimization options - O2All, O3All, O3unroll & O3only.
- Seq without optimization = running time for sequential code that was compiled without any compiler optimizations (same as presented in table 1).
- Speedup with optimization = Seq without optimization / min_seq => gives how much speedup achieved with compiler optimization w.r.t to one without it.

n	Input_type	Sorting Algo	Running Time(secs)					Seq without optimization	Speedup with optimization
			O2All	O3All	O3unroll	O3only	min_seq		
32768	RND	oets	0.67	0.66	0.67	0.75	0.66	1.9	2.88
32768	RS	oets	0.28	0.28	0.28	0.41	0.28	1.66	5.93
32768	RND	btns	0	0	0	0	0	0.03	--
32768	RS	btns	0	0	0	0	0	0.01	--
1048576	RND	oets	692.62	692.69	695.86	849.53	692.62	1991.01	2.87
1048576	RS	oets	307.81	307.8	307.74	426.76	307.74	1689.05	5.49
1048576	RND	btns	0.25	0.25	0.25	0.23	0.23	0.74	3.22
1048576	RS	btns	0.16	0.16	0.16	0.14	0.14	0.59	4.21
33554432	RND	btns	12.41	12.41	12.42	11.43	11.43	36.48	3.19
33554432	RS	btns	7.93	7.94	7.93	7.35	7.35	30.86	4.2

Table 2 - Sequential Running Times with compiler optimizations.

4.2.2 Performance comparison of Sequential and Parallel Implementations

n	Input_type	SortingAlgo	seq	cuda	speedup
32768	RND	oets	0.66	0.13	5.077
32768	RS	oets	0.28	0.14	2
1048576	RND	oets	692.62	155.2	4.463
1048576	RS	oets	307.74	180.67	1.703
1048576	RND	btns	0.23	0.03	7.667
1048576	RS	btns	0.14	0.04	3.5
33554432	RND	btns	11.43	1.53	7.471
33554432	RS	btns	7.35	1.46	5.034

Table 3 - Performance Comparison of Parallel and Sequential Running Times

As indicated in the above table, the parallel cuda code shows on an average ~5x times better performance than the sequential code. This performance is still improved in the 'bucket of keys' implementation discussed next.

4.2.3 Improved Parallel performance achieved by 'bucket of keys' approach

In chapter 2.3, an alternative parallel implementation approach, 'bucket of keys' was discussed. I implemented algorithms: odd even transposition sort and bitonic based sort with this approach in cuda, which helped gain better parallel performance w.r.t to the one achieved with regular parallel cuda implementation.

n	Algo	Cuda regular	cuda_buckets						
			2	4	8	16	32	64	128
1048576	oets	155.2	78.42	39.81	15.62	7.43	4.4	3.24	3.8
1048576	btn	0.03	0.04	0.03	0.05	0.06	0.13	-	-
33554432	btn	1.53	1.39	1.44	1.72	2.48	-	-	-

Table 4 - Parallel Running Times for 'bucket of keys' Algorithm

*These numbers are only for RND input source.

The below explanation helps to understand some of the column names of the table

- cuda_regular => Running time obtained with regular cuda code with no buckets (running the entire oets/btn algo over *n* keys in cuda)
- cuda_buckets => Running time obtained with cuda code for 'bucket of keys' algorithm.
- In cuda_buckets, *n* keys are divided in b buckets, where each bucket has *n/b* keys. Results are reported for *b*=2,4,8,16,32,64 & 128.
- Code is written such that *b* is taken as a dynamic input. So that we can test the code for any *b* value.
- As we can observe from the above results, minimum running time is achieved by b=64 in oets and *b*=2 & b=4 in btn.

4.2.4 Final Performance comparison of All Implementations

The table below shows results for all the 3 major implementations of this project :-

1. Sequential - Sequential code running on 1 CPU core.
2. Cuda regular - Regular parallel cuda code running parallely across various GPU cores.
3. Cuda_buckets - Parallelization achieved by combining GPU core parallelization + multiple buckets of keys running in parallel with help of GPU streams.

The below explanation helps to understand some of the column names of the table

- optimal_num_buckets - it is value of *b* which gave best performance (minimum value for running time in cuda_buckets) from the table 4.
- Cuda_buckets - minimum running time out of the cuda_buckets running times from table 4.
- Speedup_from_cuda_reg = cuda_buckets / cuda-regular => shows performance improvement from the regular cuda code.
- Speedup_from_seq = minimum_cuda_buckets / seq => shows performance improvement from the sequential implementation.

		Running Time(secs)					
n	Algo	seq	Cuda regular	cuda_buckets	optimal_num_buckets	Speedup from cuda_reg	Speedup from seq
1048576	oets	692.62	155.2	3.24	64	47.901	213.772
1048576	btn	0.23	0.03	0.03	4	1	7.667
33554432	btn	11.43	1.53	1.39	2	1.101	8.223

Table 5 - Performance Comparison of All Implementations

As shown in the above table, with the 'bucket of keys' cuda implementation, for odd even transposition sort, we can achieve almost ~48x times better performance w.r.t regular cuda code and ~214x times better performance w.r.t sequential code. And for bitonic sort, cuda buckets gives slightly improved performance than cuda regular but almost ~8x times better performance than sequential code.

Chapter 5 Code Links

For a detailed understanding of the different implementations, this chapter provides the link to the github repository which has files of source code for all the algorithms implemented in this project.

Github link : https://github.com/aarjavid/Parallel-Sorting-Algorithms

Chapter 6 Conclusion

The project successfully implemented odd even transposition sort and bitonic based sort in 3 different implementation modes :- Sequential, GPU Core Parallelization; and Combined GPU Core + Multistream Parallelization named ‘bucket of keys’ method. During the course of this project, I developed various automated scripts which could easily and quickly launch different experiments and analyse results. The project statistically compared all different code implementations for various different input types and number of input keys and was able to provide proper performance numbers which clearly demonstrated how each implementation is better than the other. As discussed in chapter 4.2.4, with the ‘bucket of keys’ cuda implementation, for odd even transposition sort, I could achieve almost ~48x times better performance w.r.t regular cuda code and ~214x times better performance w.r.t sequential code. And for bitonic based sort, cuda ‘bucket of keys’ gives slightly improved performance than cuda regular but almost ~8x times better performance than sequential code. Thus, I was very well able to achieve the target with which this project was started - to learn and implement different parallel approaches of sorting algorithms and achieve a good performance improvement and provide a statistical and comprehensive performance comparison of sequential and different parallel sorting techniques on CPU & GPU.

Chapter 7 Future Scope

Although the project achieved a good performance improvement in parallel implementation of sorting algorithms as compared to their sequential ones, much work can be done on this in future which could not be achieved due to the limited time span of this project. To name a few of the the top priority future updates on the current version of this project are:-

1. Improve performance of cuda regular code, by advanced programming of grid dimensions.
2. Although the ‘bucket of keys’ implementation uses the multistreaming feature of GPU, due to some technical challenges, it seems this feature is not able to provide proper parallelization of streams. Debug this with nvprof - virtual profiling tool of nvidia and fix this feature which will drastically improve the cuda ‘bucket of keys’ performance even further.
3. Add caching in cuda code, to check if it gives better performance.
4. Modify the pattern of input keys range which goes to every CUDA core to check if it can give a better cuda performance.
5. Add multi-threading in cuda code, to get a wholesome parallelization of CPU and GPU.

Chapter 8 References

[1] Madhavi Desai, Viral Kapadiya, Performance Study of Efficient Quick Sort and Other Sorting Algorithms for Repeated Data, National Conference on Recent Trends in Engineering & Technology, 13-14 May 2011.

[2] D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Second ed. Boston, MA: Addison-Wesley, 1998.

[3] Ishwari Singh Rajput ,Bhawnesh Kumar ,Tinku Singh ,Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms ,International Journal of Computer Applications (0975 – 8887) Volume 57– No.9, November 2012

[4] Performance Analysis of Sequential and Parallel Programming Paradigms on CPU-GPUs Cluster by B N Chandrashekhar , H A Sanjay ,Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV 2021).

[5] X. Qian and J. Xu, "Optimization and implementation of sorting algorithm based on multi-core and multi-thread," *2011 IEEE 3rd International Conference on Communication Software and Networks*, 2011, pp. 29-32, doi: 10.1109/ICCSN.2011.6014668.

[6] F. G. Khan, O. U. Khan, B. Montrucchio and P. Giaccone, "Analysis of Fast Parallel Sorting Algorithms for GPU Architectures'," *2011 Frontiers of Information Technology*, 2011, pp. 173-178, doi: 10.1109/FIT.2011.39.

[7] Parallelization of Sorting Algorithms by Narek Abroyan, Robert Hakobyan presented at the Conference: Computer Science and Information Technologies (CSIT) At: Yerevan, Armenia in September 2015.

[8]https://en.wikipedia.org/wiki/Batcher_odd%E2%80%93even_mergesort

[9] Efficient Parallel Algorithms by Wojciech Rytter Alan Gibbons

[10]http://math.mit.edu/~shor/18.310/batcher.pdf

[11] Lec 10: Odd Even Merge Sort (OEMS) by professor Sajith Gopalan, Dept. Of Computer Science and Engineering, IIT Guwahati - https://www.youtube.com/watch?v=20rAiIjncg0

[12] Lec 11:: OEMS, Bitonic-Sort-Merge Sort (BSMS) by professor Sajith Gopalan, Dept. Of Computer Science and Engineering, IIT Guwahati - https://www.youtube.com/watch?v=OpTf4xT9Cpc&list=RDCMUCCDzHkpuIuD1ZC0wsCXUuPQ&start_radio=1&t=3419s

[13] Peters H., Schulz-Hildebrandt O., Luttenberger N. (2010) Fast In-Place Sorting with CUDA Based on Bitonic Sort. In: Wyrzykowski R., Dongarra J., Karczewski K., Wasniewski J. (eds) Parallel Processing and Applied Mathematics. PPAM 2009. Lecture Notes in Computer Science, vol 6067. Springer, Berlin, Heidelberg.

[14] https://cseweb.ucsd.edu//classes/fa06/cse160/Lectures/Readings/bitonic.pdf

[15] https://www.mimuw.edu.pl/~rytter/TEACHING/2CS68/sorting2_net.ps

[16]https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s12-lec28-slides-JMC.pdf?version=1&modificationDate=1333163955158

[17] CUDA programming Version 1.1(November 17, 2017) by Prof. Alexandros Gerbessiotis, CS Department, NJIT, Newark

[18] https://nichijou.co/cuda8-Stream/

[19] https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[20] Comparison Networks (Fall-2021 Notes) by Prof. Alexandros Gerbessiotis, CS Department, NJIT, Newark