

CONCURRENT MATRIX COMPUTATION-CSE 557

FINAL PROJECT REPORT

AMOGH BEDARAKOTA

akb6975@psu.edu

Pary Search

Introduction

P-ary search is a novel, scalable parallel search algorithm optimized for Single Instruction Multiple Data (SIMD) architectures such as GPUs. It substantially outperforms traditional search methods like binary search by leveraging parallelism to improve both throughput and response time. Specifically, P-ary search:

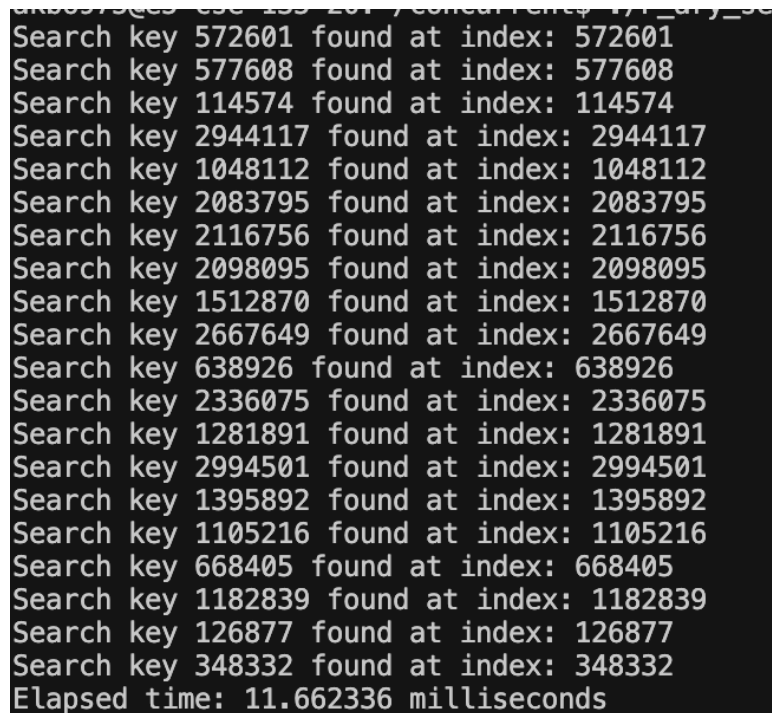
- Utilizes a Divide-and-Conquer Approach: All threads in a block search for the same key but in different segments of the sorted data, allowing for concurrent memory access and efficient use of the GPU's memory bandwidth.
- Scales with the Number of Threads: It can manage individual search queries collaboratively across multiple threads, effectively maintaining nearly constant query response times regardless of workload size.
- Efficient Memory Access: P-ary search supports coalesced memory access, particularly suited for data structures like B-trees used in database indexes, which optimizes memory usage and decreases latency.
- Algorithmic Efficiency: The search operates with a time complexity of $\log_p(n)$, where p is the parallel thread count, showing significant improvements in response times compared to conventional algorithms for large workloads.

In practice, P-ary search enables rapid and efficient querying, making it especially powerful for applications that require searching through large datasets in environments where high parallelism can be exploited, such as with GPUs in database search operations.

Experimental Setup

Experiments were conducted on NVIDIA RTX A4500 GPUs installed in Westgate 135 machines. The GPUs feature a clock speed of 1.5 GHz and are equipped with 20 GB of GDDR6 VRAM. The host machines are powered by Intel Core i7 processors with quad-core architecture, running at 2.6 GHz, and are configured with 32 GB of DDR4 RAM. The operating system used is CentOS Linux, version 8.4, with Kernel 5.10. The NVIDIA graphics drivers version 550.54.15 and CUDA version 12.4 were employed for GPU computation.

I create a sorted array whose values start from 0 and end at arraySize-1. Pary Search finds keys in the array and returns the indices at which the keys are present. Here is the screenshot of the result

A screenshot of a terminal window with a black background and white text. It displays the results of a search operation on a sorted array. The output consists of 17 lines, each showing a search key and its corresponding index, followed by a final line showing the elapsed time. The keys and indices are identical, ranging from 572601 to 348332 in descending order. The elapsed time is 11.662336 milliseconds.

```
Search key 572601 found at index: 572601
Search key 577608 found at index: 577608
Search key 114574 found at index: 114574
Search key 2944117 found at index: 2944117
Search key 1048112 found at index: 1048112
Search key 2083795 found at index: 2083795
Search key 2116756 found at index: 2116756
Search key 2098095 found at index: 2098095
Search key 1512870 found at index: 1512870
Search key 2667649 found at index: 2667649
Search key 638926 found at index: 638926
Search key 2336075 found at index: 2336075
Search key 1281891 found at index: 1281891
Search key 2994501 found at index: 2994501
Search key 1395892 found at index: 1395892
Search key 1105216 found at index: 1105216
Search key 668405 found at index: 668405
Search key 1182839 found at index: 1182839
Search key 126877 found at index: 126877
Search key 348332 found at index: 348332
Elapsed time: 11.662336 milliseconds
```

Steps to run the Cuda code

```
nvcc P_ary_search.cu -o P_ary_search
./P_ary_search
```

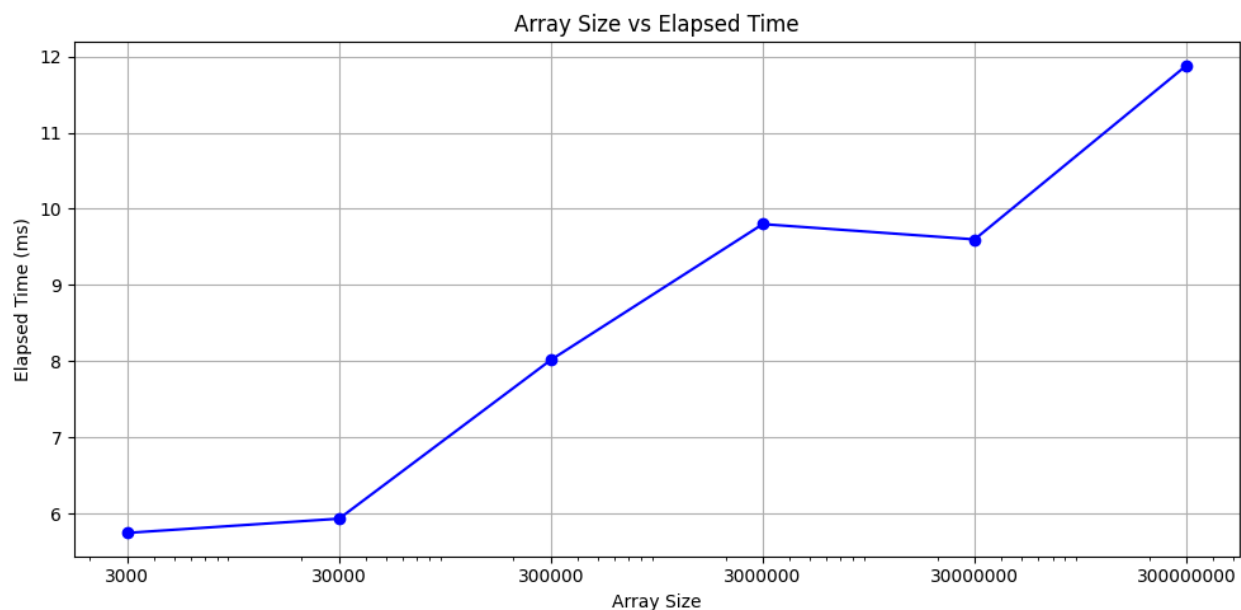
Steps to run the sequential code

```
gcc Binary_search.c -o Binary_search
./Binary_search
```

Results

I kept my number of search keys as 20 and threads per block to be 32 and increased the size of the array from 3×10^3 to 3×10^8 and calculated the elapsed time. If I increase the size of the array further CUDA runtime is unable to allocate the requested amount of memory on the GPU

arraySize	average elapsed time(millisecond)
3000	5.749760 milliseconds
30000	5.936128 milliseconds
300000	8.018944 milliseconds
3000000	9.799776 milliseconds
30000000	9.598976 milliseconds
300000000	11.880288 milliseconds

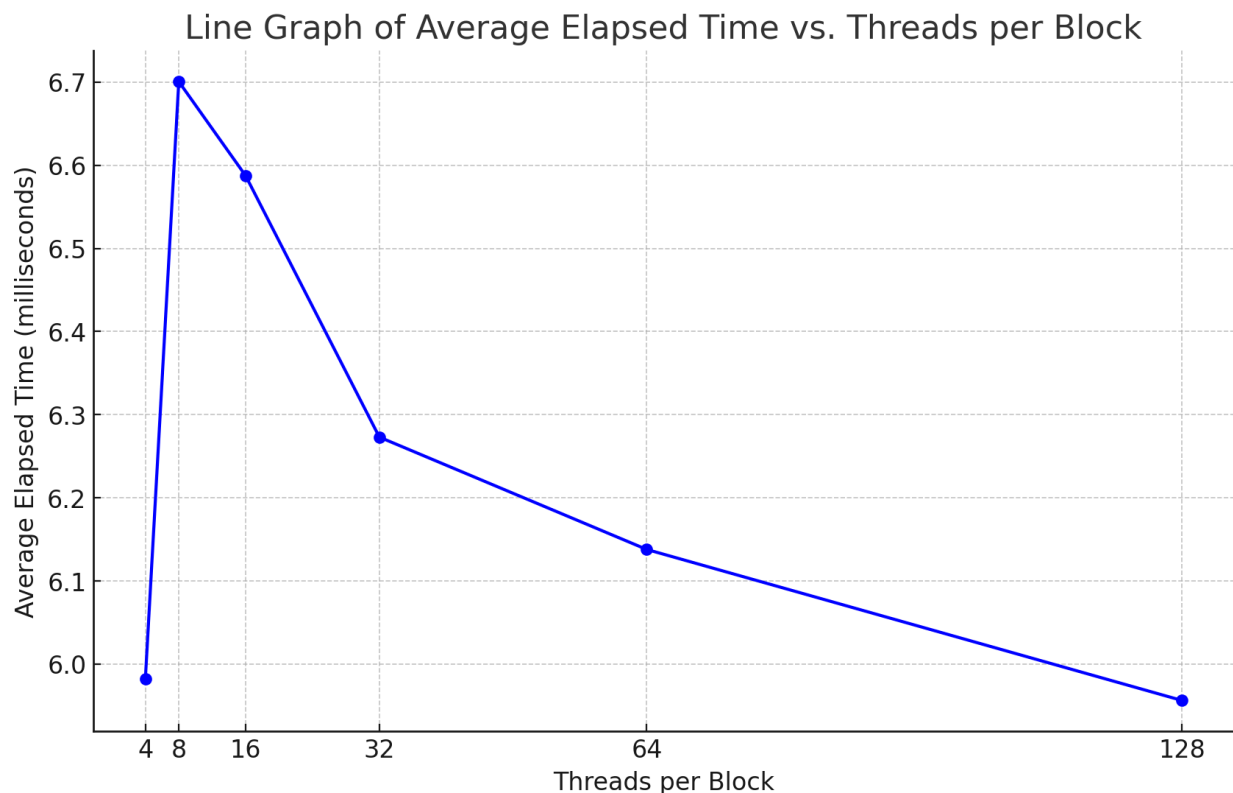


For smaller array sizes (from 3000 to 3000000), we see a gradual increase in the elapsed time. The curve is sub-linear, which suggests that the algorithm scales well with the increasing size. The small increases in time despite tenfold increases in array size could be due to the parallel nature of the GPU efficiently handling larger data sets with more threads. As the array size increases to 300,000,000, there is a more pronounced increase in the elapsed time. This indicates that as we approach the limits of the GPU's memory or processing capabilities, the P-ary search starts

to scale less efficiently. It's possible that with larger data sets, memory bandwidth becomes a limiting factor, leading to increased times.

In the second scenario, I increase the number of threads per Block from 4 to 128 keeping the size of array as 300000000 and number of search keys(number of Blocks) to be 20 and calculate the elapsed time

Threads per Block	average elapsed time(milliseconds)
4	5.982208
8	6.70064
16	6.58739
32	6.273024
64	6.137856
128	5.956352

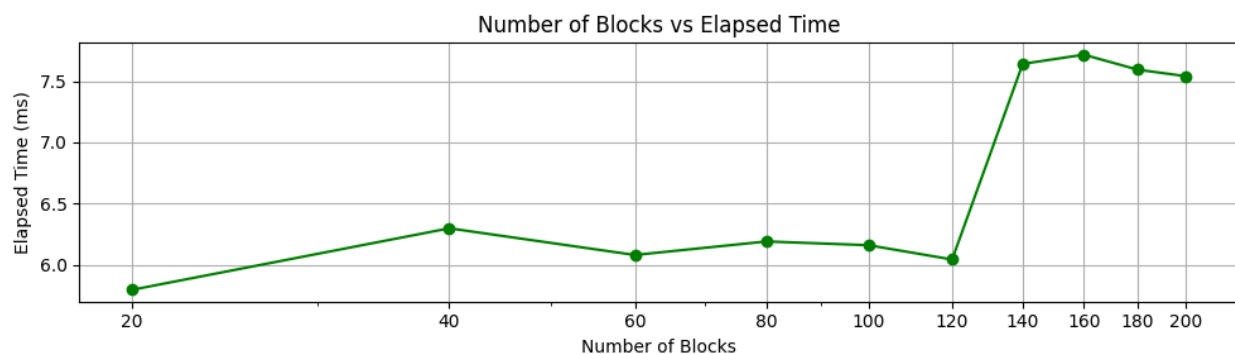


The graph shows that as the number of threads per block increases from 4 to 8, there's an initial increase in average elapsed time, which might be due to overhead from managing more threads. However, as the thread count continues to increase beyond 8, the elapsed time generally decreases, hitting the lowest at 128 threads

per block. This decrease in time with more threads can be attributed to better utilization of the GPU's parallel processing capabilities. More threads per block can effectively reduce the time it takes to process the data by distributing the workload more efficiently across the GPU's cores, minimizing idle time and maximizing computational throughput.

In the third scenario, I increase the number of Blocks (number of search keys) from 20 to 200 in steps of 20 and calculate the elapsed time keeping the array size as 300000000 and keeping the number of threads per Block as 32

Number of Blocks	average elapsed time(milliseconds)
20	5.794816
40	6.297600
60	6.0795529
80	6.190080
100	6.159360
120	6.042624
140	7.641312
160	7.716672
180	7.594080
200	7.540736



The graph shows a trend of initially increasing average elapsed time with an increasing number of blocks, followed by a fluctuation around a certain level. This pattern can be attributed to how the workload is distributed among the available computational resources. Initially, as the number of blocks increases, more computational resources are utilized, resulting in a decrease in elapsed time. However, beyond a certain point, the overhead of managing a larger number of

blocks starts to outweigh the benefits gained from increased parallelism, leading to fluctuations in performance. Additionally, factors such as memory access patterns and hardware limitations may also contribute to this behavior.

Graph Betweenness Centrality

Introduction

Betweenness centrality is a fundamental measure used in network analysis to identify the importance of nodes (or vertices) in a graph. It quantifies the number of times a node acts as a bridge along the shortest path between two other nodes. The concept is critical in understanding the flow and control of information in various types of networks, such as social networks, communication networks, and transportation networks.

The calculation of betweenness centrality for a node v involves counting the number of shortest paths from all pairs of nodes that pass through v . The formula for betweenness centrality (BC) of a node v can be expressed as follows:

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

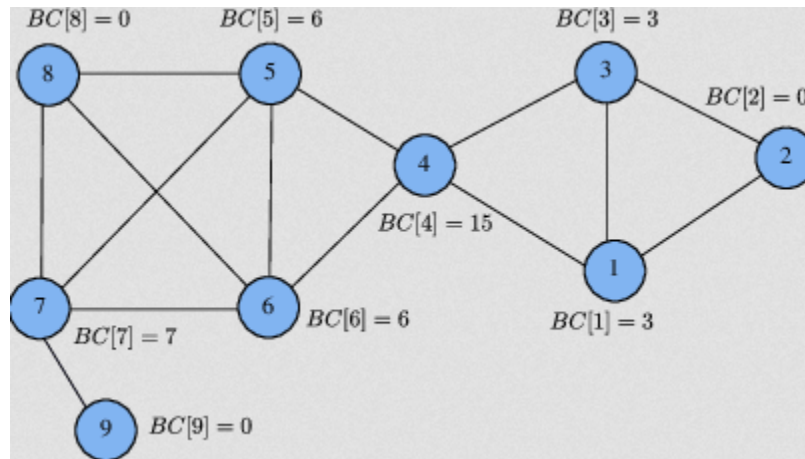
where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v .

The significance of betweenness centrality lies in its ability to reveal the potential influence of a node within a network. Nodes with high betweenness centrality scores are often critical for maintaining the connectivity of the network, and removing these nodes can significantly disrupt communication within the network. Therefore, betweenness centrality is not only a theoretical measure but also has practical applications in network design, analysis, and intervention strategies.

The chapter 2 of the GPU GEMS Jade Edition provides sequential and CUDA pseudocodes for computing Betweenness Centrality values of a graph. I have implemented them. I faced some issues while implementing the CUDA version.

Experimental Setup

Experimental setup is the same as the one for P_ary search. I created a random graph(Adjacency List form). I converted this random graph into cuGraph which consists of 'froms' and 'nhbrs' arrays . I pass this data structure and initialised BC array(along with other necessary arrays) as parameters to computeBetweennessCentrality which further invoke forwardPropagation and backwardPropagation. Here is a small toy example graph for which the program generated Betweenness Centrality values.



```
froms: 0 0 0 1 1 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 8
nhbrs: 1 2 3 0 2 0 1 3 0 2 4 5 3 5 6 7 3 4 6 7 4 5 7 8 4 5 6 6
Vertex 0: Betweenness Centrality: 3.000000
Vertex 1: Betweenness Centrality: 0.000000
Vertex 2: Betweenness Centrality: 3.000000
Vertex 3: Betweenness Centrality: 15.000000
Vertex 4: Betweenness Centrality: 6.000000
Vertex 5: Betweenness Centrality: 6.000000
Vertex 6: Betweenness Centrality: 7.000000
Vertex 7: Betweenness Centrality: 0.000000
Vertex 8: Betweenness Centrality: 0.000000
akb6975@e5-cse-135-20:~/concurrent$
```

The Betweenness Centrality values are matching according to the toy example graph (Note that node's index start from 0 in my implementation)

Steps to run the Cuda code (which creates random graph)

```
nvcc finalised_BC.cu -o finalised_BC
./finalised_BC
```

Steps to run the sequential code which prints the example provided in the picture

```
gcc small_graph_example_BC.c -o small_graph_example_BC
./small_graph_example_BC
```

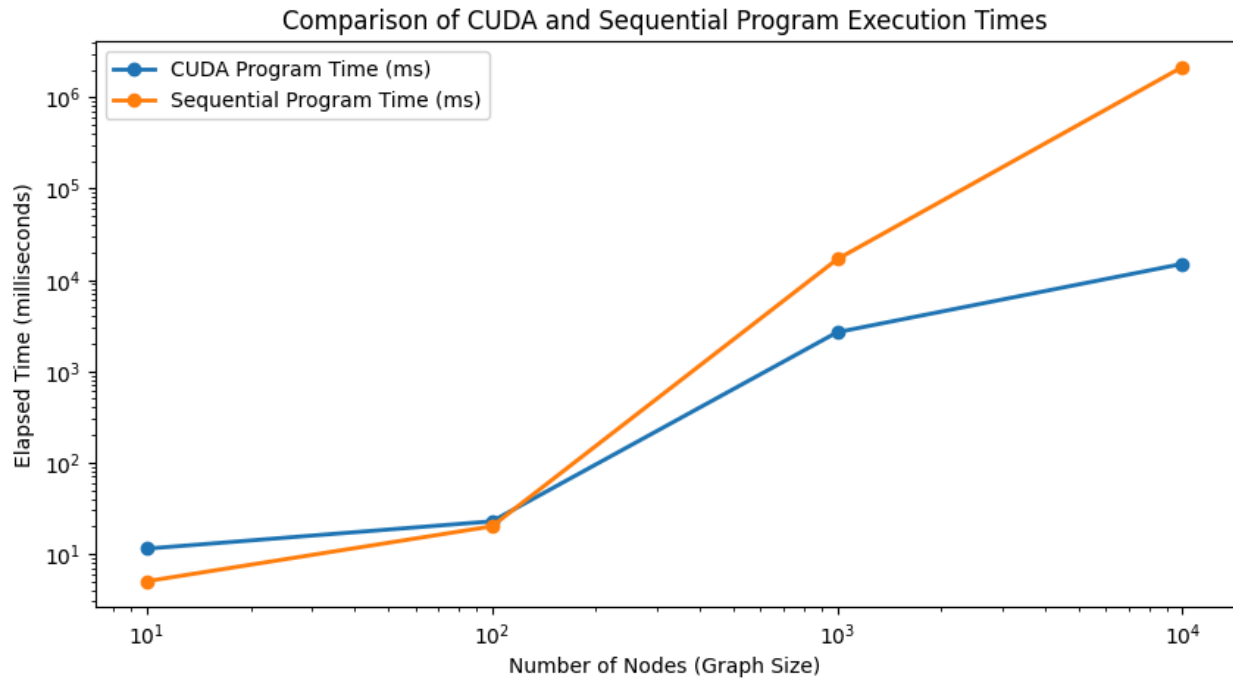
Steps to run the sequential code (which creates random graph)

```
gcc BC_seq.c -o BC_seq
./BC_seq
```

Results

I kept my number of Blocks to be 128 and number of threads per block to be 512 and increased the graph size from 10 nodes to 10000 nodes and calculated the execution times. Here are the results

Number of Nodes (graph size)	CUDA program time elapsed time (milli sec)	Seq Program time elapsed time (milli sec)
10	11.374496	0.000000
100	22.660864	20.000000
1000	2673.655029	16870.000000
10000	15004.89789	2156180.0000



This graph presents a comparison of execution times for a CUDA program versus a sequential program across different graph sizes, ranging from 10 to 10,000 nodes. On a logarithmic scale, both the number of nodes and the elapsed time are plotted to show the performance relationship as graph complexity increases. Initially, for smaller graphs (10 nodes), the CUDA program time starts higher than the sequential, possibly due to overheads in setting up parallel computation. However, as the number of nodes increases, the CUDA program shows better performance relative to the sequential one, with significantly lower execution times. This trend continues, and the gap widens dramatically for 10,000 nodes, where the parallelization benefits of CUDA yield a substantial decrease in computation time compared to the much slower sequential approach, which increases sharply and surpasses the CUDA time significantly at the highest node count.

Bitonic Sort

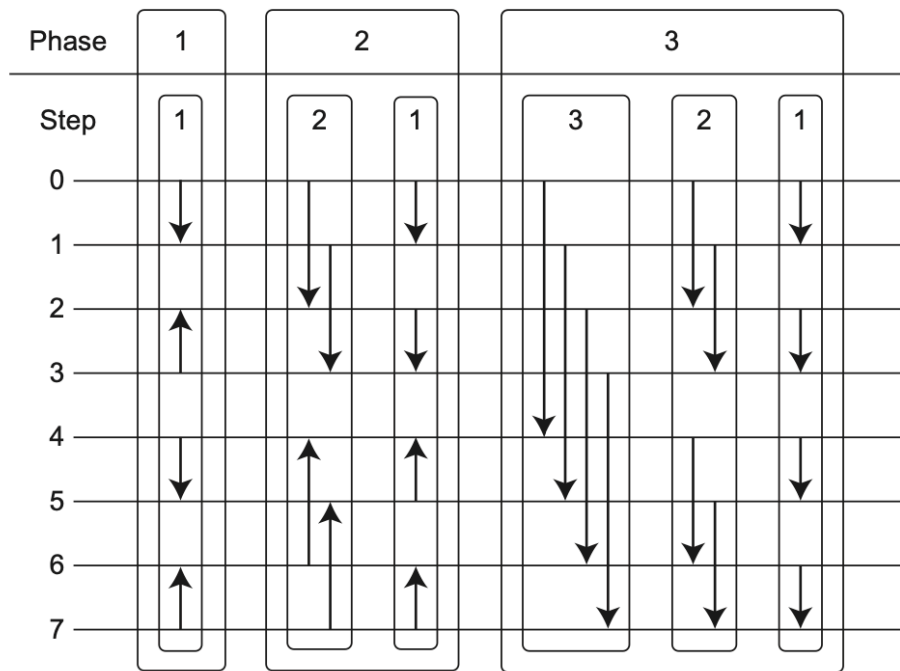
Introduction

Bitonic sort, introduced by Ken Batcher in 1967, is a parallel sorting algorithm recognized for its predictable performance and suitability for hardware implementations, especially on GPUs like NVIDIA's. This sorting method employs a comparison-based sorting network that enables efficient parallelization, crucial for performance gains on modern computing architectures. The structure of bitonic sort is such that it requires sequences of data to pass through a series of phases, each consisting of multiple steps involving compare-and-exchange operations between data elements. This method's parallel nature is highly compatible with CUDA's model, where each thread can independently execute part of the network, leveraging the GPU's capabilities to handle numerous operations concurrently.

In practical implementations on NVIDIA GPUs, the challenge is optimizing the sequence of operations to minimize the expensive memory accesses and synchronization requirements. The algorithm's in-place characteristic helps limit memory usage but requires clever management of data flow and thread synchronization to maintain efficiency. Optimizations include minimizing global memory access by intelligently grouping operations within threads and kernel launches to reduce the overhead of data communication. Such approaches significantly improve upon the basic implementation by reducing the number of required kernel launches and memory accesses, making the bitonic sort competitive with more complex sorting algorithms, particularly in environments where data distribution is not a limiting factor.

Experimental Setup

Experimental setup is the same as the one for P_ary search. I create a random array of size 2^k and pass this array into the BitonicSort function. I print the sorted array afterwards. I am implementing the basic Bitonic sort algorithm.



Here are the results

```
akb6975@e5-cse-135-20:~/concurrent$ nvcc bitonicSort.cu -o bitonicSort
akb6975@e5-cse-135-20:~/concurrent$ ./bitonicSort

The initial unsorted array values are
1822 6111 9744 8382 3079 7253 7289 381 113 43 7196 852 7874 4953 4969 7834 2804 6056 6269 4165 7948 957
20 7296 4291 5851 4550 1580 2585 1015 1623 9781 8219 5850 1086 3188 36 3891 9244 6305 4408 3545 2236 38

The final sorted array values are
36 43 113 308 381 471 779 852 889 979 1015 1086 1129 1232 1382 1580 1623 1822 2236 2301 2585 2804 2989
0 4953 4969 5070 5850 5851 6056 6111 6269 6305 6420 7183 7196 7253 7289 7296 7335 7552 7834 7874 7948 8
akb6975@e5-cse-135-20:~/concurrent$
```

Steps to run the Cuda code

```
nvcc bitonicSort.cu -o bitonicSort
./bitonicSort
```

Steps to run the sequential code

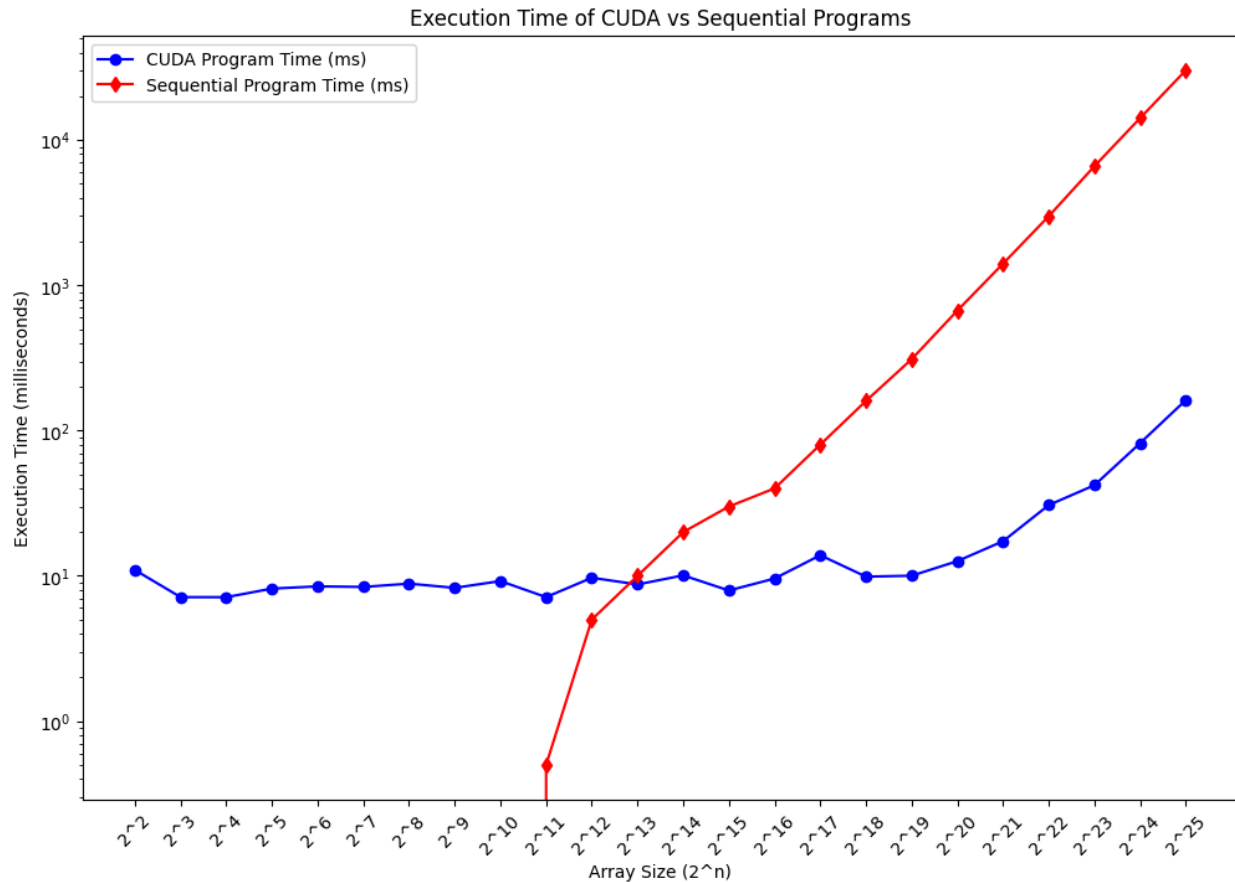
```
gcc bitonicSort_seq.c -o bitonicSort_seq -lm
./bitonicSort_seq
```

Adding -lm tells the linker to link against the math library (libm), which includes the implementation for the pow function and other math-related functions.

Results

I kept my number of threads per block to be 512 and increased the array size from 2^2 to 2^{25} nodes to 10000 nodes and calculated the execution times. Here are the results

Size of Array	CUDA program time elapsed time (milli sec)	Seq Program time elapsed time (milli sec)
2^2	10.969824	0.000000
2^3	7.137120	0.000000
2^4	7.129824	0.000000
2^5	8.170080	0.000000
2^6	8.462432	0.000000
2^7	8.386592	0.000000
2^8	8.839552	0.000000
2^9	8.266112	0.000000
2^{10}	9.205248	0.000000
2^{11}	7.135648	0.000000
2^{12}	9.703168	0.000000
2^{13}	8.724640	10.000
2^{14}	10.074944	20.000
2^{15}	7.926208	30.000
2^{16}	9.582944	40.000
2^{17}	13.865568	80.000
2^{18}	9.874944	160.000
2^{19}	10.019616	310.000
2^{20}	12.623232	670.000
2^{21}	17.313313	1410.000
2^{22}	30.700480	2990.000
2^{23}	42.058399	6610.000
2^{24}	81.909439	14160.000
2^{25}	162.234818	30240.000



The graph illustrates the execution times for both CUDA (parallel) and Sequential (non-parallel) implementations of the Bitonic Sort algorithm across varying array sizes, scaled as 2^n . Initially, the CUDA program maintains a consistent execution time across smaller array sizes, showcasing the efficiency and overhead management of parallel computing. However, as the array size surpasses 2^{12} , CUDA times begin to gradually increase, yet they remain substantially lower than the sequential times until about 2^{23} . In contrast, the sequential execution time remains negligible until 2^{13} , beyond which it increases exponentially with array size. This steep increase highlights the limitations of sequential processing for larger data sets, where the lack of parallelism leads to significantly longer execution times compared to CUDA, emphasizing the advantage of parallel processing for handling large-scale computations more effectively.