N-Puzzle using AI Algorithms on GPU

KAUSTAV VATS, ANUBHAV JAISWAL and ARSHDEEP SINGH, Indraprastha Institute of Information Technology

1. INTRODUCTION

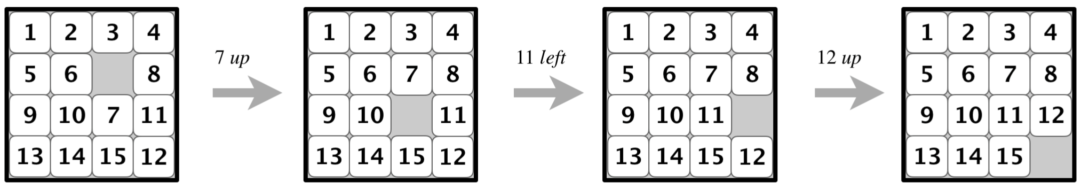

Fig. 1. Example of a 15-Puzzle Problem

The N-Puzzle problem consist of $\sqrt[2]{N+1}$ size square tiles numbered from 1 to N-1 with one tile missing. The objective of the problem is to arrange tile in some numerical order.

Intuitively solving this problem requires the algorithm to visit all the possible states and select the final solution state. Visiting states requires recursion and backtracking which is easy to implement in a serial code(Non threading implementation), Thus implementing these graph algorithms on the GPU is a challenging task that we wish to accomplish.

There are 4 algorithms in this field **BFS**, **DFS**, **A*** and **IDA*** algorithm. For the parallel implementation we will be implementing BFS and A* but the serial implementation would be there for all the 4 algorithms

2. LITERATURE REVIEW

2.1 Parallelizing the BFS algorithm

The paper "Accelerating Large Graph Algorithms on the GPU Using CUDA"[2] has been referred to understand the parallel approach for BFS algorithm. Graph algorithms require recursion and implementation of queues and stacks that might lead to an hindrance in the performance hence this paper describes how to implement a parallel version of the BFS and other graph algorithms. It defines a new way to store the adjacency list of the graph making parallel execution easier and causing less overheads.

2.2 Parallelizing the A* algorithm

"Massively Parallel A* Search on a GPU [3]", This paper explains three parallel things required to make the A* algorithm work in parallel. It describes the parallel Computation of the Heuristic function, parallel implementation of priority queue and detecting Node duplication on GPU.

3. THE 4 ALGORITHMS USED FOR N PUZZLE PROBLEM

N-Puzzle is a graph traversal problem. Where each state contains a structure of a representation of the puzzle in a form of 2D Array. Each state can have at max 4 neighbouring state, depending upon the position of the blank tile. If blank tile is on the corners there will be 2 possible state. When blank tile is on the edges there are 3 possible state and 4 for rest of the positions.
Big issue with N-Puzzle problem is that, search space is not defined. It's an exploratory search problem, which means that the algorithm can be massively parallelized.

3.1 Breadth First Search Algorithm

Breadth first search is the most commonly used algorithm for solving graph traversal search space problems. Bfs start traversing from a start node and traverse the graph layerwise "Fig. **??**" thus exploring the neighbour nodes. Bfs also handles already visited nodes and visits each node only once.
Serial implementation of bfs uses queue to store neighbours. Then it pops the neighbour from the queue and add it's neighbour if not visited to the queue again. In the breadth first fashion, it first visits all neighbours of the current node and so on.

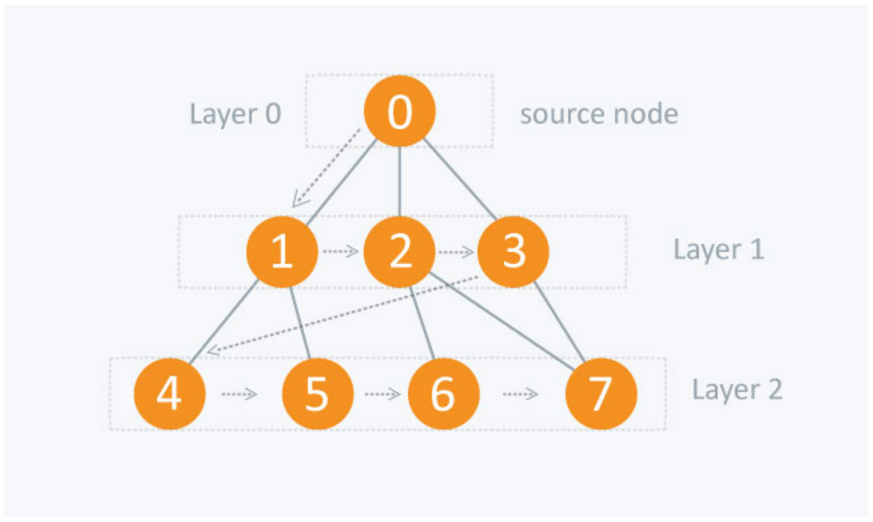

Fig. 2. Example of a bfs approach

3.2 Depth First Search Algorithm

Depth first search algorithm is a recursive algorithm, which basically uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
Here backtrack means when you are moving forward there's no state available to explore further, so you move backwards on the same path to find nodes to traverse.

3.3 A* Algorithm

A* algorithm is very similar to Breadth first search algorithm. A* uses a heuristic function to decide which state need to be explored first and which is closer to final solution. Whereas Bfs explore each and every node, A* only explore those node which have a less total cost. Heuristic tries to minimize the sum of distances between the initial state and the current state and the final state and the current state. Ref “Fig. 3”

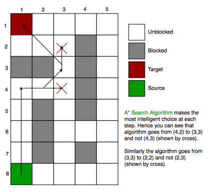
1
2
3
4
5
6
7
8
Unblocked
Blocked
Target
Source
A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).
Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Fig. 3. A* algorithm approach

3.4 Iterative Deepening A* Algorithm

On similar lines, IDA* limits some nodes from visiting, if they have a cost of visiting greater than a threshold. It follows a Dfs approach however, only visits to a threshold of depth. This depth is calculated as the heuristic distance between the initial state and the final state. Ref “Fig. 4”

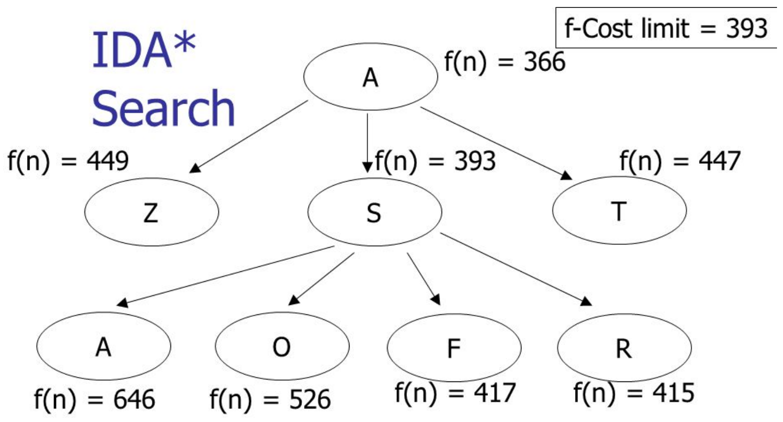

Fig. 4. IDA* algorithm approach

4. ANALYSIS OF BFS FOR PARALLELIZATION

Trying to parallelize the serial approach is has two difficulties: A Recursive BFS will affect the performance and stack overflow is likely to occur cause our state space is of (N)! Where N is the size of the puzzle. Maintaining a queue would require atomic operations on the queue which again would lead to communication overhead among the threads Hence the authors of the paper Accelerating Large Graph Algorithms on the GPU Using CUDA have suggested a different approach to store as well as query the graph.

We will query the graph level by level and use the adjacency list to store the representation of the graph.

We will achieve parallelization on each level where the nodes in a level are independent of each other. Each thread will be given the task of node on a level. The kernel would be called the same number of times as are the total number of levels to visit

The task dependency graph is given in figure 5. For the task interaction graph also the same graph is valid as the data requirements for task are satisfied using the same dependency.

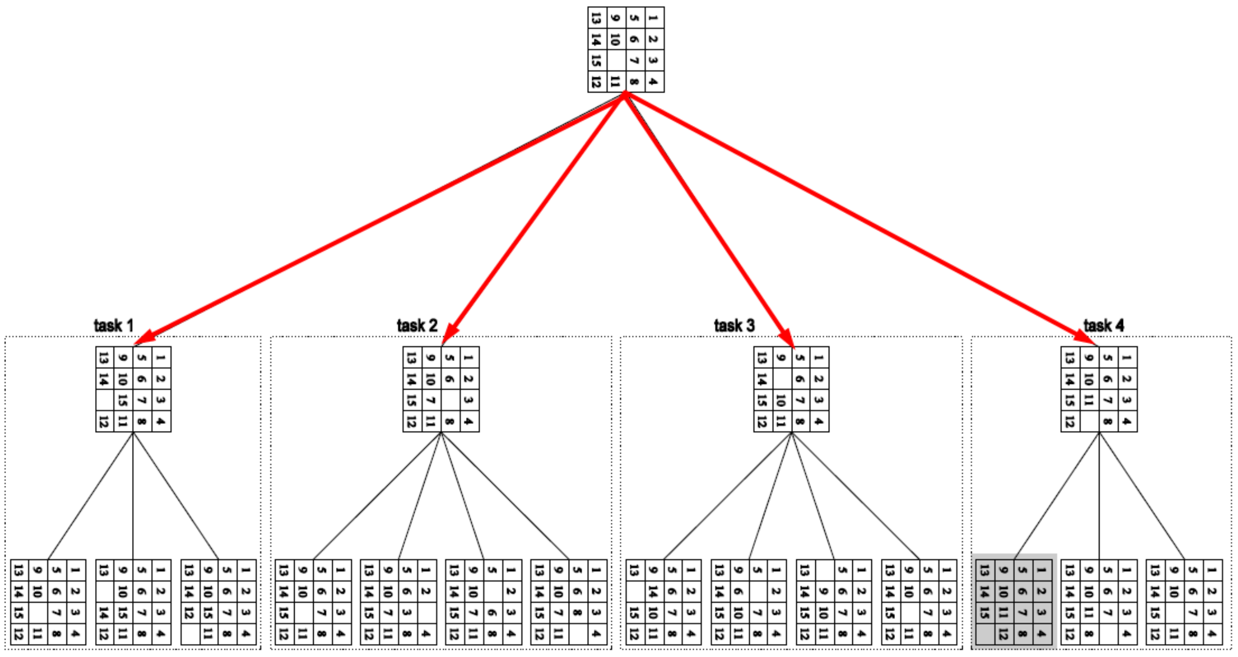

Fig. 5. Task Dependency Graph

5. ANALYSIS OF A* FOR PARALLELIZATION

The A* algorithm, serially, is implemented using priority queues. Initially the root element or the starting state of the puzzle is added to the first priority queue. In a parallel ensemble each thread has to access the same priority queue thus causing bank conflicts. This was averted by using a different priority queue for each thread. To sum up the running procedure uptil now, each thread is taking an element from its respective priority queue and comparing it with the final destination for finding the solution state. Each thread after comparison, generates a maximum of 4 new neighbours, which are added to different threads. This can be either random or a algebraic function (threadID*4 + neighbourIndex), where the latter is preferred to control flow divergence and idle threads in a warp. This implementation is done with a single kernel call where each thread is running in a loop. However, for many steps, most of the threads remain idle as the queues are empty. This was avoided via the Multiple Kernel Call implementation. Here, a thread doesn't have a loop, rather the kernel call is iterated. Each kernel call analyses a level of nodes and produces further child nodes (which are a maximum of 4). At each kernel call the number of threads is increased 4 times. This produces divergence till 2 steps, after which each warp has all the threads working. However, new overheads of kernel calls are introduced here.

Neighbours can be generated in series as well as parallel. For the serial one, each thread produces its own 4 neighbours, however, for the latter part 4 different threads do the given task.

The task dependency graph is given in figure 5. For the task interaction graph also the same graph is valid as the data requirements for task are satisfied using the same dependency.

6. IMPLEMENTATION OF PARALLEL BFS ALGORITHM

To take advantage of the level wise approach of the BFS, we modify the adjacency list. So we have an array of Edges and we will have a structure of vertices that stores the starting position in the Edges array from where its edges start and the number of nodes it is attached to.

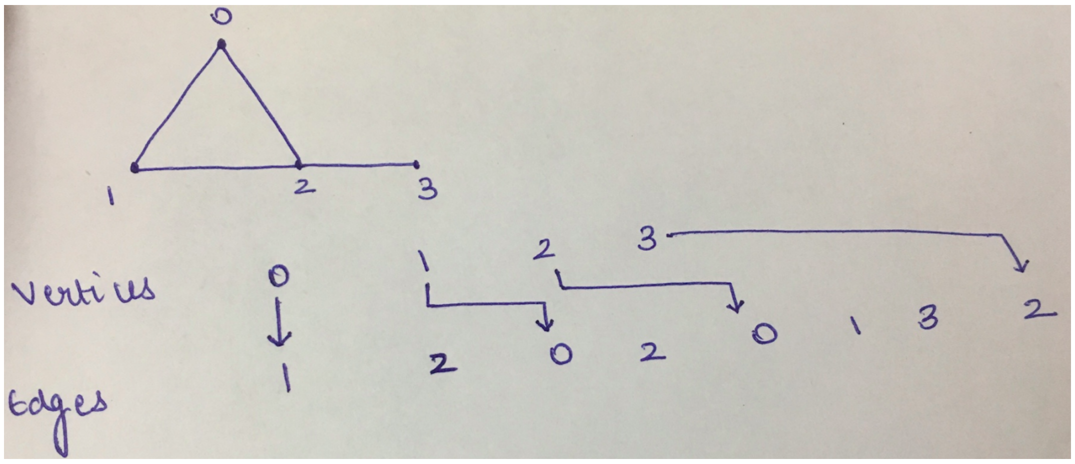

Fig. 6. Parallel Bfs approach

In “Fig. 6” edges of vertex 0 starts from 0th index and has length 2, vertex 1 starts from 2nd index and has length 2 and so on.

Along with the above array and structure, we maintain another visited array, cost array, and level array. Visited array prevents from visiting the node again, cost array stores the minimum cost path and level array tells whether this vertex lies on the level we are currently exploring.

We have implemented the parallel BFS for N-Puzzle and checked whether we are able to reach the solution and if we are then what is the cost. The main part of the implementation was to make the modified adjacency list of the graph. We have assumed each node to be a permutation of the puzzle and thus each node has a maximum of 4 neighbors. We have first given a unique id to each permutation and then proceeded with creating neighbors and storing them in the adjacency list. For the 4-puzzle problem since there are 24 states that we have to store, it is easily done on GPU but for N more than 4, the memory limit on GPU does not allow to use this form of storing the graph. Hence the results are listed for 4-puzzle problem.

The implementation for parallelly executing bfs in cuda has been referenced from[3].

7. IMPLEMENTATION OF PARALLEL A* ALGORITHM

To implement the parallel A* algorithm, We have first implemented the priority queue on CUDA. In the priority queue, the heapify function where queue is to be updated cannot be done using recursion and hence a loop is used for that. In case of recursion it is not able to calculate the stack memory that will be required in the GPU.

Each thread handles a priority queue. THE nodes generated by each thread, which can be a maximum of 4, are pushed into different priority queues, which are processed parallely in the next epoch. The priority queues are not related to each other. Any priority queue's state can expand and push the generated neighbour states in any other queue. This was a done using a deterministic function where a priority queue will push the data into the subsequent 4 priority queues.

8. RESULTS

We observed that serial implementation of A* and IDA* mostly perform best as compared to Serial implementation of Bfs, Dfs Algorithm. Below graph can give much better comparison between all algorithms. Refer to Fig 7. In this the line for DFS is not visible because for n=15 DFS is not able to find the solution state, it takes a lot of time. We have completed serial implementation of all four algorithms in our project and bfs cuda implementation. We have implemented generic code for different value of N and different complexity of Start state.

Table 1 shows the time for the algorithms along with the matrix dimension.

Table 3 refers to the parallel BFS algorithm, how the time varies as the number of levels to visit increases. Here level refers to the level in the graph where the solution state of the puzzle lies. Table 2 refers to the parallel A* algorithm, where as we increase the number of priority queues(or the number of threads), the kernel time is noted.

Figure 8 represents the execution time with the number of priority queues for the parallel A* algorithm.

Speedup - The Formula for speedup is given by (Serial Time) divided by (Parallel Time), For the Serial time we have to choose the best serial algorithm, the data mentioned in Table 1 refers to simpler start state and hence cannot be used for comparison. Thus we provide a sufficiently good start state and compare the speedups. The speedup is between Serial A* and Parallel A*. Table 4 represents the values of speedups on increasing the number of priority queues. Figure 9 represents the graph for speedup, as we increase the number of priority queues, speedup increases uptil 10 priority queues.

Table I.

Matrix Dim\Algorithms	**Bfs Serial**	**Dfs Serial**	**A* Serial**	**IDA* Serial**
2x2	0.069ms	0.017ms	0.039ms	0.015ms
3x3	0.058ms	2.429ms	0.014ms	0.010ms
4x4	8905.442ms	-	680.001ms	0.321ms

Table II.

Priority Queues	**Time Kernel**	**Time Kernel+Memory**
1	140.094ms	142.933ms
2	50.174ms	50.969ms
3	88.7876ms	89.2429ms
5	47.582ms	48.0329ms
7	47.324ms	47.879ms
11	48.4481ms	49.2266ms

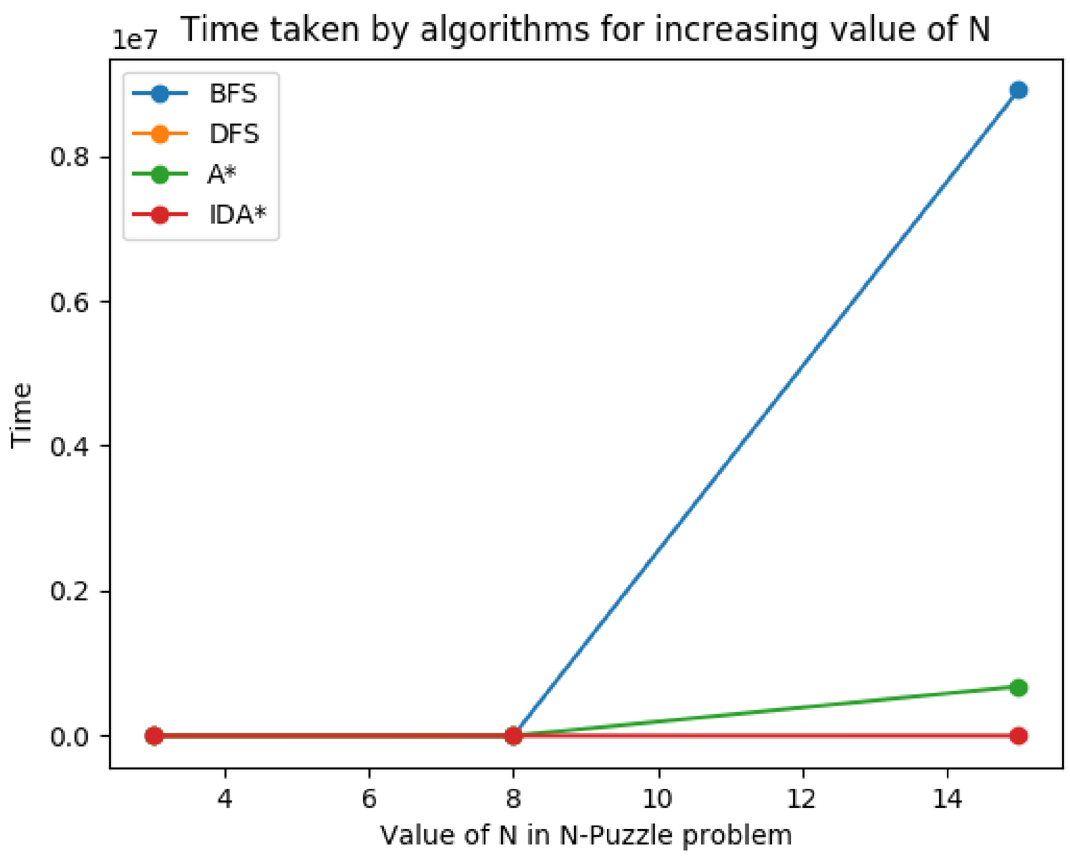

Fig. 7. Execution time of A* for different number of priority queues -

Table III.

Level	**Time Kernel**	**Time Kernel+Memory**
0	0.08ms	0.37ms
1	0.12ms	0.43ms
2	0.17ms	0.54ms
3	0.23ms	0.60ms
4	0.26ms	0.71ms
5	0.31ms	0.77ms
6	0.35ms	0.80ms

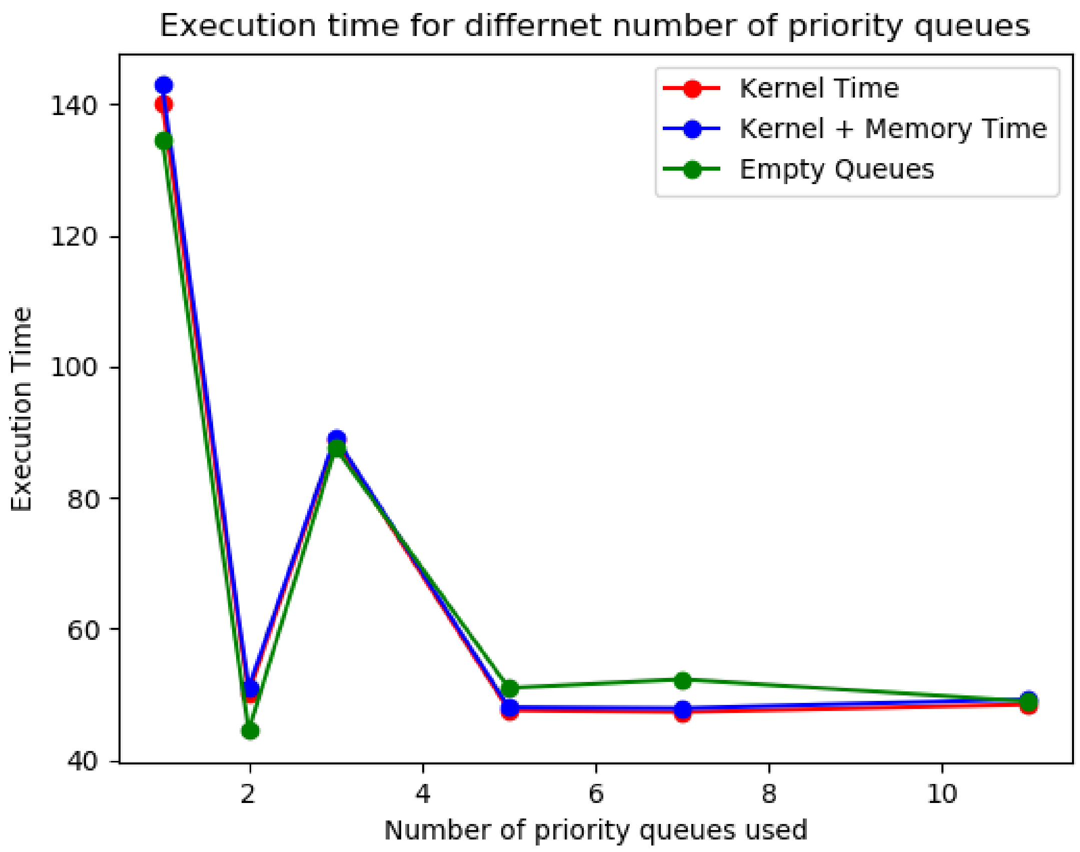

Fig. 8. Execution time of A* for different number of priority queues -

Table IV.

Priority Queues	**Parallel Time**	**speedup**
2	50.174ms	2.7921
3	88.7876ms	1.5778
5	47.582ms	2.9442
7	47.324ms	2.9603
11	48.4481ms	2.8916

9. MILESTONES

(1) Mid Term
  (a) Serial implementation of A*, BFS, DFS and IDA* Algorithm - Done
  (b) Parallel implementation of BFS Algorithm - Done
  (c) Literature survey of parallel algorithm of A* - Done

(2) End Term

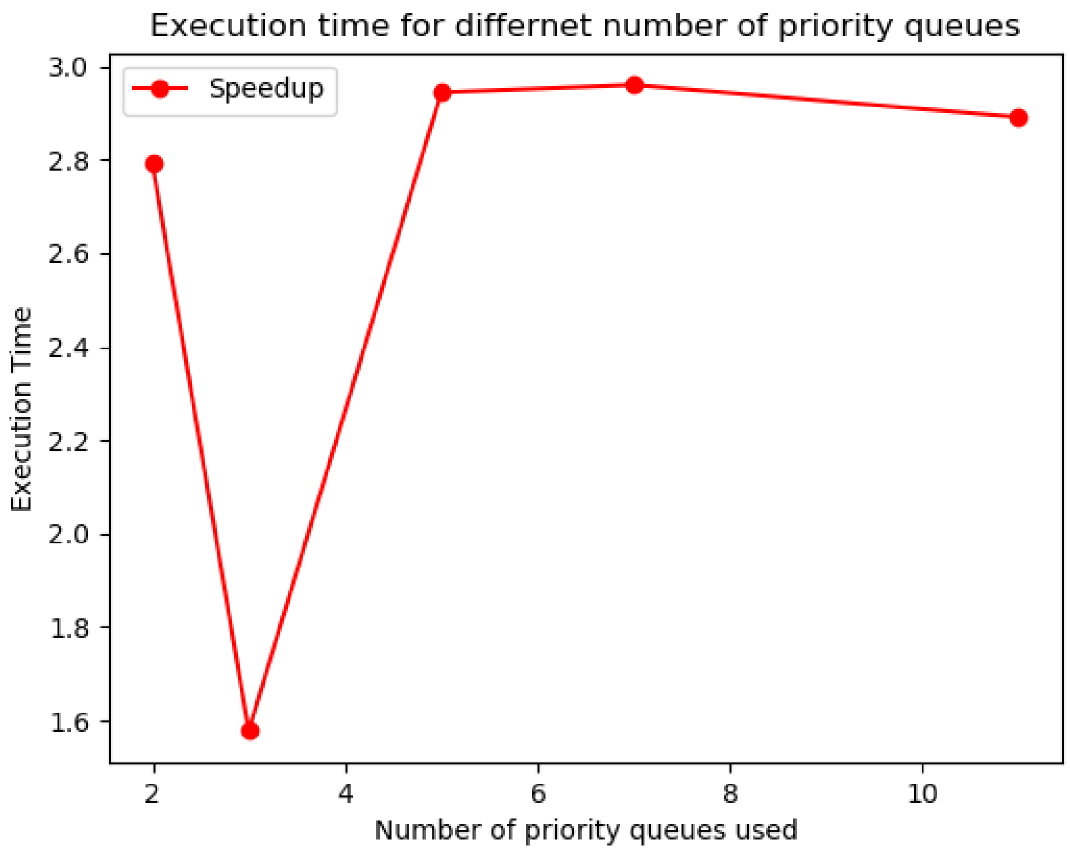

Fig. 9. Speed up Graph on varying the number of prioirity queues in the A* parallel Algorithm

(a) Custom Priority Queue Implementation - Done
(b) Implementing the parallel BFS algorithm for the N puzzle problem - Done but GPU memory does not allow for storing the states of problems with N 9 and above.
(c) Parallel implementation of A* algorithms proposed. - Done
(d) Comparison and analysis of running time of all the algorithms for varying value of N. - Done

10. FINAL REMARKS AND CHALLENGES FACED

—AI algorithms are much faster than other search algorithm

—No cuda parallel algorithm available for DFS and IDA*. We implemented the paper which uses priority queue on GPU and maintains the property of the A* algorithm

—For the BFS algorithm, one of the challenges is the creation of the adjacency list representation that is not known before, hence creating a list and then using it is similar to the serial version of the BFS We tried different variants but were mostly performing worst than the serial implementation of A* algorithm.

—A* uses multiple helper functions, which can be parallelized to achieve the speedup. We observed that for some cases, parallelizing the part makes it slower than before.

—Memory problem for 15-Puzzle and 24-Puzzle problem for parallel A*.

11. INDIVIDUAL CONTRIBUTION

Kaustav - Get Neighbours function implemented parallely along with other functions used in A*. The whole structure of the Algorithms, including all supplementary functions with serial implementation.

Anubhav - Multiple Kernel call implementation of A* algorithm along with serial implementations

Arshdeep - Single Kernel call implementation of A* algorithm along with serial implementations

12. REFERENCES

[1] https://github.com/siddharths2710/cuda_bfs/blob/master/cuda_bfs/kernel.cu
[2] https://www.nvidia.co.uk/content/cudazone/CUDABrowser/downloads/Accelerate_Large _Graph_Algorithms/HiPC.pdf
[3] https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/download/9620/9366