Parallelizing A Star for Path Planning in CUDA

Aman Kushwaha (amankush), Priya Thanneermalai (tpriya)

Introduction:

A* is a graph traversal and path search algorithm, which is used in many fields of computer science due to its completeness, optimality, and optimal efficiency. Though it has a high space complexity of **$O(b^d)$** where b is the branching factor and d is the shortest path, A* is still the best solution in many cases. In this project, we have parallelized the A Star Algorithm using the algorithm described in the paper, Massively Parallel A* Search on a GPU, co-authored by Yichao Zhou and Jianyang Zeng. This paper talks about a speed up of 45 times faster than the serial version.

A CPU usually contains several highly optimized cores for sequential instruction execution, while a GPU typically contains thousands of simpler but more efficient cores that are good at manipulating different data at the same time. This makes solving really large problem sizes very fast on GPUs. So we need to modify search algorithms originally designed for a CPU to exploit such a large amount of parallelism brought by a GPU. The paper applies parallelized A Star to problems like Path Planning, Sliding Puzzle and Protein Design. In this project, we have solved the problem of Path Planning where we find the most efficient route between any source and destination on a 2D Grid World. This solution can further find applications in Robotics, Machine Learning and Artificial Intelligence.

Our motivation behind picking this problem lies in our interest in A Star algorithm and the Path Planning problem. The serial A star algorithm, while simple when sequential, presents multiple complexities when parallelizing on GPUs as described in this report.

Serial A Star Search:

Traditional implementations of A* search usually uses two lists to store the states during its expansion, i.e., the open list and the closed list. The closed list stores all the visited states, and is used to prevent unnecessary repeated expansion of the same state. This list is often implemented by a linked hash table to detect the duplicated nodes. The open list normally stores the states whose successors have not been fully explored yet. The open list uses a priority queue as its data structure, typically implemented by a binary heap. States in the open list are sorted according to a heuristic function $f(x)=g(x) + h(x)$ where $g(x)$ is the distance from the starting node to current state x, and the function $h(x)$ is the estimated distance from current state x to the end node.

In each round of A* search, we extract the state with the minimum f value from the open list, expand its outer neighbors and check for duplication. After that, we calculate the heuristic functions of the resulting states and then push them back to the open list. If the nodes of some states have already been stored in the open list, we only update their f values for the open list.

Also note that we require the heuristic function to be admissible for optimality, i.e., h(x) is never greater than the actual distance to the end node.

Parallel A Star Search Algorithm and Figures:

Algorithm 1 GA*: Parallel A* search on a GPU

```
1: procedure GA*(s, t, k)
                ▷ find the shortest path from s to t with k queues
2:     Let {Q_i}_{i=1}^k be the priority queues of the open list
3:     Let H be the closed list
4:     PUSH(Q_1, s)
5:     m ← nil                          ▷ m stores the best target state
6:     while Q is not empty do
7:         Let S be an empty list
8:         for i ← 1 to k in parallel do
9:             if Q_i is empty then
10:                continue
11:            end if
12:            q_i ← EXTRACT(Q_i)
13:            if q_i.node = t then
14:                if m = nil or f(q_i) < f(m) then
15:                    m ← q_i
16:                end if
17:                continue
18:            end if
19:            S ← S + EXPAND(q_i)
20:        end for
21:        if m ≠ nil and f(m) ≤ min_{q∈Q} f(q) then
22:            return the path generated from m
23:        end if
24:        T ← S
25:        for s' ∈ S in parallel do
26:            if s'.node ∈ H and H[s'.node].g < s'.g then
27:                remove s' from T
28:            end if
29:        end for
30:        for t' ∈ T in parallel do
31:            t'.f ← f(t')
32:            Push t' to one of priority queues
33:            H[t'.node] ← t'
34:        end for
35:    end while
36: end procedure
```

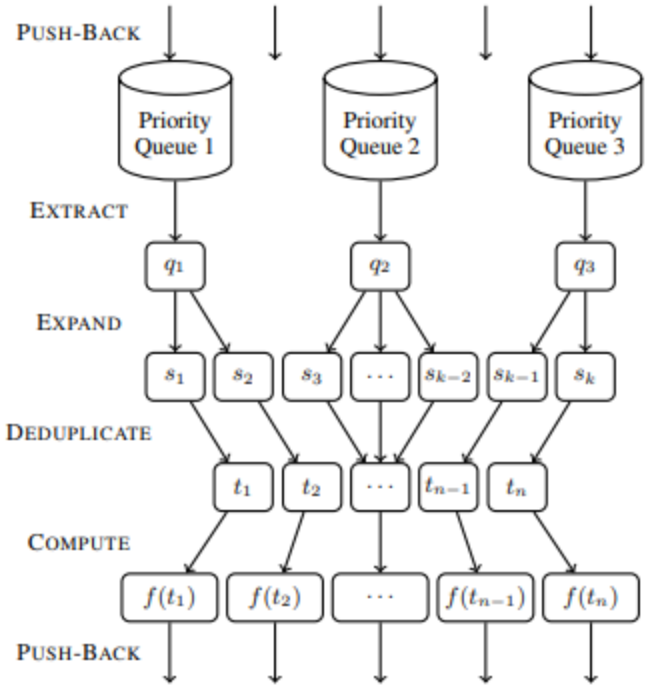


Figure 1: Data flow of the open list. The symbols in this diagram matches those in Algorithm 1. In this example, the number of the parallel priority queues is three. In practice, we usually use thousands of parallel priority queues for a typical GPU processor.

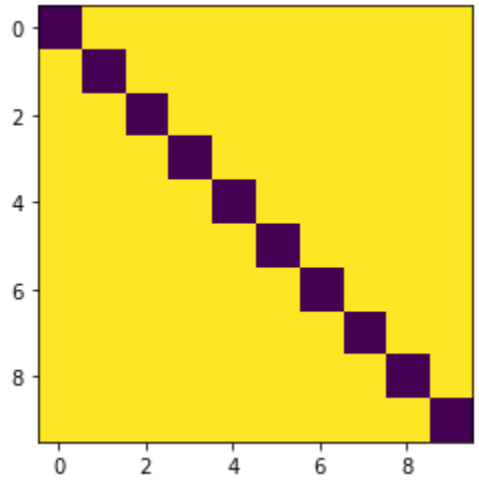


Figure 2. Figure showing how serial A* explores the grid from start (0,0) to goal (9,9) position. The heuristic helps the Algorithm to explore towards the direction of the goal. Dark ones are the points which are explored during the search.

Algorithm 1 and Figure 1 depict the Massively Parallel A Star Search from the paper. We implemented this algorithm completely by using 3 priority queues.

Parallel A Star Search Implementation Steps:

In our implementation, we divided Algorithm 1 into four parts and implemented each part by launching a separate kernel. We run a while loop in our CPU code until the goal is found. We initialized the priority queues Q (Q1, Q2, Q3 i.e. 3 queue), an intermediate list S, and closed list H in the following way:

1. We used four arrays (Q1_x, Q1_y, Q1_z, Q1_f) to initialize the priority queue Q1. Q1_x stores the x coordinate value, Q1_y stores the y coordinate value, Q1_z stores the z value and Q1_f stores the f value. The ith point in Q1 will be x = Q1_x[i], y = Q1_y[i], z = Q1_z[i] and f = Q1_f[i]. We defined Q2 (Q2_x, Q2_y, Q2_z, Q2_f) , Q3 (Q3_x, Q3_y, Q3_z, Q3_f) , H (closed_x, closed_y, closed_z, closed_f) and S (S_x, S_y, S_z, S_f) in a similar fashion. Since we implemented the algorithm in CUDA, we could not use sophisticated data structures like hash tables, linked lists, trees but only arrays to store values.
2. Additionally, we used another integer variable in GPU(Q1_ptr, Q2_ptr, Q3_ptr, S_ptr, and closed_ptr) which stores the index of the next element to be inserted element in the Q1, Q2, Q3, S, closed lists respectively. This helps us to keep track of the number of elements in each list. This also enables us to keep track of relevant values in the open and closed list arrays during parallel operations.
3. We used up all the 1024 threads in a block and dedicated one queue to each block so that all the queues can be processed in parallel. We ensured that we updated the f and g values as soon as we expanded a point in the S list. All steps are done in parallel.
4. We had to strategically decide the number of blocks, priority queue division and think carefully about how to implement each of the closed, open lists and S list. We also had to think about deduplication quite a bit.
5. We also had to allocate memory as per the expected size of the lists. For instance, our S list was really big in size as the input grid size grew since it effectively contains all points expanded.

<u>Initializing the Queues with initial values:</u>

1. Before starting the while loop in the CPU, we initialized Q1 by adding the starting point from the grid to Q1. We could use any of the priority queues here to start with, but we used Q1. Thus Q1_x[0] = 0, Q1_y[0] = 0, Q1_f[0] = euclidian distance from start to goal, Q1_z[0] = 0, Q1_ptr = 1. Remember Q1_ptr must hold the index of the element we want to enter next into Q1. We started the grid from coordinates (0,0) and set our target to the top right corner of the grid.
2. We then added the starting point to the explored list (closed list) thus closed_x[0] = 0, closed_y[0] = 0, closed_f[0] = euclidian distance from start to goal, closed_ptr[0] = 1.
3. We also have to initialize Q2_ptr = 0, Q3_ptr = 0, and S_ptr = 0 as all the arrays had to be empty before starting the while loop in the CPU code.
4. Now the following four steps will run in the while loop until the target or goal is reached.

Step 1 Parallelization and Implementation:

1. The S list should be emptied at the start of the while loop. To empty the S list we simply set S_ptr = 0.
2. In step1, each point from Q (all of Q1, Q2, Q3) is explored in parallel. Each thread picks a point (index is decided by looking at its blockIdx and threadIdx) from Q and checks if the index point is valid or not and if the index is valid it takes the x,y, z points from Q and expands these points. It checks this validity by checking if its index is less than Q_ptr (no. of elements in Q).
3. Each point is expanded by taking 8 points around the given point in the grid and the f values are calculated as the sum of z + 1 + heuristics. Here z + 1 is the cost to reach the given point and the f value is the sum of z value for the expanded point (z+1) and the heuristics (Euclidean distance between expanded point and the goal). This Euclidean distance helps in finding the shortest path and moving towards the goal in A* parallel search and during backtracking in serial and parallel code.
4. After expanding, these 8 points are added to the S list. The S list, by design, will contain a lot of duplicate points. This is because a lot of the 8 expanded points are duplicates on the grid. To add these points in the S list the S_ptr is updated using the AtomicAdd operation. The returned value is the index where the x,y,z and f value of the expanded points should be put and we put these values of the expanded point in the given index returned by AtomicAdd operation. In this way we ensure that each point is put in a unique location, thereby avoiding any race conditions. Note that after all this, S_ptr gives the number of elements in the S.
5. In summary, this step involves looking at a point in the priority queue and expanding it's 8 neighbors and dumping all of them into S list. This is done in parallel meaning each thread picks a point from a priority queue and calculates its 8 neighbors.
6. If any of these neighbors match the target node then we know that we found a path from source to target. In that case, we update a variable m to store this point.

Step 2 Parallelization and Implementation:

1. We used Euclidean distance to calculate distances between points on the grid. Note that the algorithm also uses a T list which is used to remove duplicate values in S. However, we did not need to create this additional T list. Alternatively, we set the value of the corresponding point in the S list to -99 wherever we needed to remove nodes from the T list.
2. The second step involves de-duplication. We first looked into the closed list and if a point from S was already in the closed list, we set its corresponding S_f value to -99 to prevent it from being expanded next time. Since our S list is global, we used blockIdx.x*1024 + threadIdx.x to calculate the global index of S list. This is done in parallel by all threads. We also checked that this global index was valid by comparing it to S_ptr.

3. The paper uses either the Cuckoo Search Algorithm or the Parallel Hashing with Replacement Algorithm to detect duplicates. Since hash tables are hard to implement in CUDA, we used a linear search to locate duplicates.
4. For this, we launched a kernel function from the CPU in a for loop since launching a kernel is relatively cheaper and quicker. The for loop ran the same number of times as the number of values in the S list which we got from the pointer to S list (S_ptr) that stored the last relevant index in S list.
5. We passed every point from the S list from the CPU function and our kernel function in the for loop, searched for this point in the entire S list except its own position. If found, we set its S_f value to -99. We calculated the global index of the S list (blockIdx.x * 1024 + threadIdx.x) and compared it to S_ptr as before.
6. Our S list now has been freed of duplicates by comparing to it the closed list as well as S list itself.

Step 3 Parallelization and Implementation:

1. The third step in our code involves going through each value in the S list and pushing it to the closed list if it does not already exist in it. We use our useful -99 in S_f to check this and calculate global index of S list using blockIdx.x * 1024 + threadIdx.x. Since the closed list is in global memory as well, we used atomicAdd on closed_ptr to get the relevant index in the closed list and prevent a race condition once again. This is done by all threads in parallel as well.

Step 4 Parallelization and Implementation:

1. The fourth and last step is pushing everything from the S list (which has been freed of duplicates) into one of the priority queues. We divided S list points by the number of priority queues, here 3 so that each queue got equal points.
2. We used our global index to get all points from the S list in parallel. If it was a valid point meaning if its corresponding S_f value is not -99 and if the point is less than S_ptr, we stored the point in one of the queues using the modulo operator to divide the global index and decide between Q1, Q2 and Q3.
3. Since this was also done by all threads in parallel, we had to use atomicAdd again on pointers to queues to avoid a race condition.

Confirmation of Correctness:

In order to confirm the correctness of our parallel A* algorithm implementation, we first wrote a sequential serial code in C++. We printed the path in this serial code along with points expanded. We also drew by hand on paper the steps that the parallel algorithm must take at

each step and confirmed that by printing our S list, closed list, priority queues. We did this by copying all these lists from device to host and printing from CPU code. We matched these values of points expanded to the serial code and to our paper pencil drawing. We have our serial and parallel codes and reference papers in this google drive, if needed:

https://drive.google.com/drive/folders/1jdFrUvxYpqj5df52o9EmkMndLoEFRhk2?usp=sharing

Confirmation value table for 4*4 grid (start (0,0) and goal (4,4)):

Iteration no.	**S list values after expansion**	**S after dedublication and removing elements that is in closed list**
0	[(0,1), (1,1), (1,0)]	[(0,1), (1,1), (1,0)]
1	[(0,0), (0,2), (1,0), (1,2), (1,1), (0,0), (0,1), (0,2), (1,0), (1,2), (2,0), (2,1), (2,2), (0,0), (2,0), (0,1), (1,1), (2,1)]	[(0,2), (1,2), (2,0), (2,1), (2,2)]
2	[(0,1), (0,3), (1,1), (1,2), (1,3), (0,1), (0,3), (0,2), (1,1), (1,3), (2,1), (2,2), (2,3), (1,0), (1,1), (2,1), (3,0), (3,1), (1,0), (1,1), (1,2), (2,0), (2,2), (3,0), (3,1), (3,2)]	[(0,3), (1,3), (2,3), (3,0), (3,1), (3,2)]
3	[(0,2), (0,4), (1,2), (1,3),(1,4), (0,2), (0,3), (0,4), (1,2), (1,4), (2,2), (2,3), (2,4), (1,2), (1,3), (1,4), (2,2), (2,4), (3,2), (3,3), (3,4), (2,0), (2,1), (3,1), (4,0), (4,1), (2,0), (2,1), (2,2), (3,0), (3,2), (4,0), (4,1), (4,2), (2,1), (2,2), (2,3), (3,1), (3,3), (4,1), (4,2), (4,3)]	[(4,0), (1,4), (2,4), (3,3), (3,4), (0,4), (4,1), (4,2), (4,3)]
4	[(3,0), (3,1), (4,1), (0,3), (0,4), (1,3), (2,3), (2,4), (1,3), (1,4), (2,3), (3,3), (3,4), (2,2), (2,3), (2,4), (3,2), (3,4), (4,2), (4,3), **(4,4),** (2,3), (2,4), (3,3), (4,3), **(4,4)**, (0,3), (1,3), (1,4), (3,0), (3,1), (3,2), (4,0), (4,2), (3,1), (3,2), (3,3), (4,1), (4,3), (3,2), (3,3), (3,4), (4,2), **(4,4)**	[(4,4)]

Timing Runs:

Grid Size	**Serial Time (ms)**	**Parallel Time (ms)**
4 * 4	44	0.557312

10 * 10	110	2.46029
20 * 20	245	9.53002
35 * 35	489	30.0336
50 * 50	791	61.8787
100 * 100	2163	246.513
200 * 200	6617	1168.05
500 * 500	32151	19851.8
1000 * 1000	117222	57139.6
1200 * 1200	165416	68231.3

Calculations:

The speed ups for each input size as seen from above table can be calculated as follows:

Speed Up (4) = 44 / 0.557312 = 78.95039

Speed Up (10) = 110 / 2.46029 = 44.71017

Speed Up (20) = 245 / 9.53002 = 25.7082

Speed Up (35) = 489 / 30.0336 = 16.28176

Speed Up (50) = 791 / 61.8787 = 12.78307

Speed Up (100) = 2163 / 246.513 = 8.77438

Speed Up (200) = 6617 / 1168.05 = 5.66499

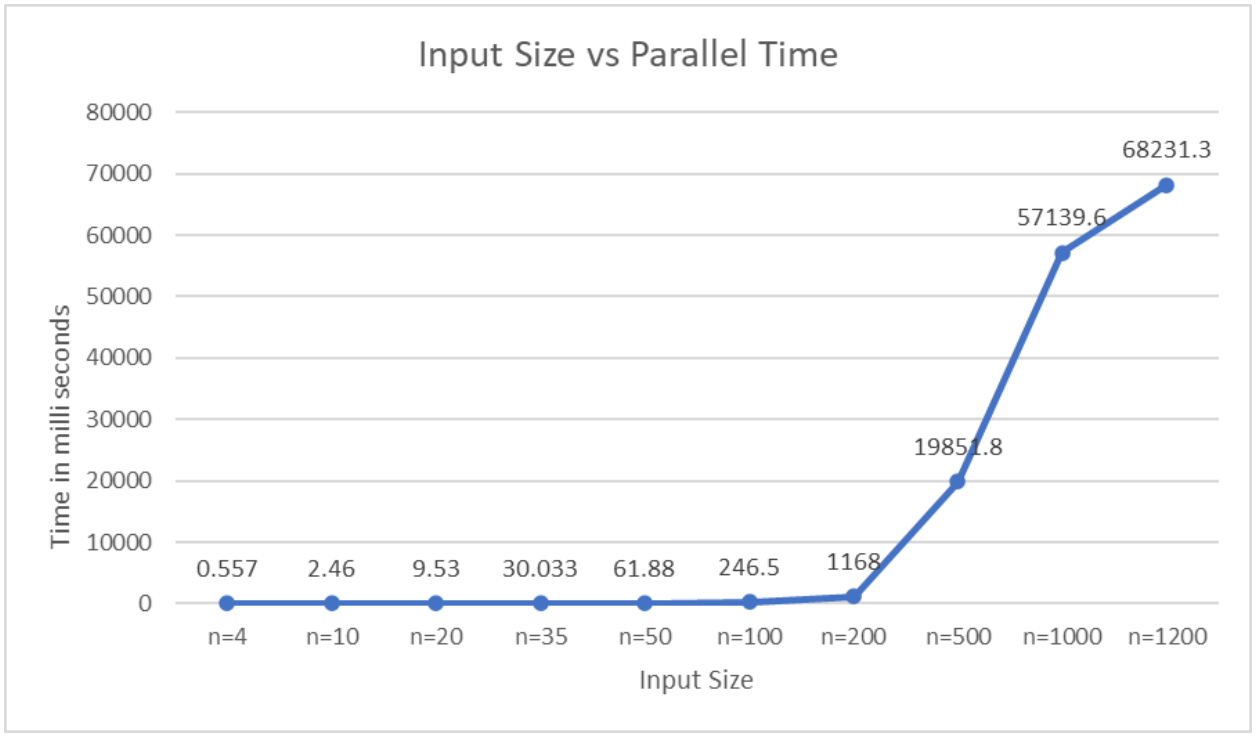
Speed Up (500) = 32151 / 19851.8 = 1.61955

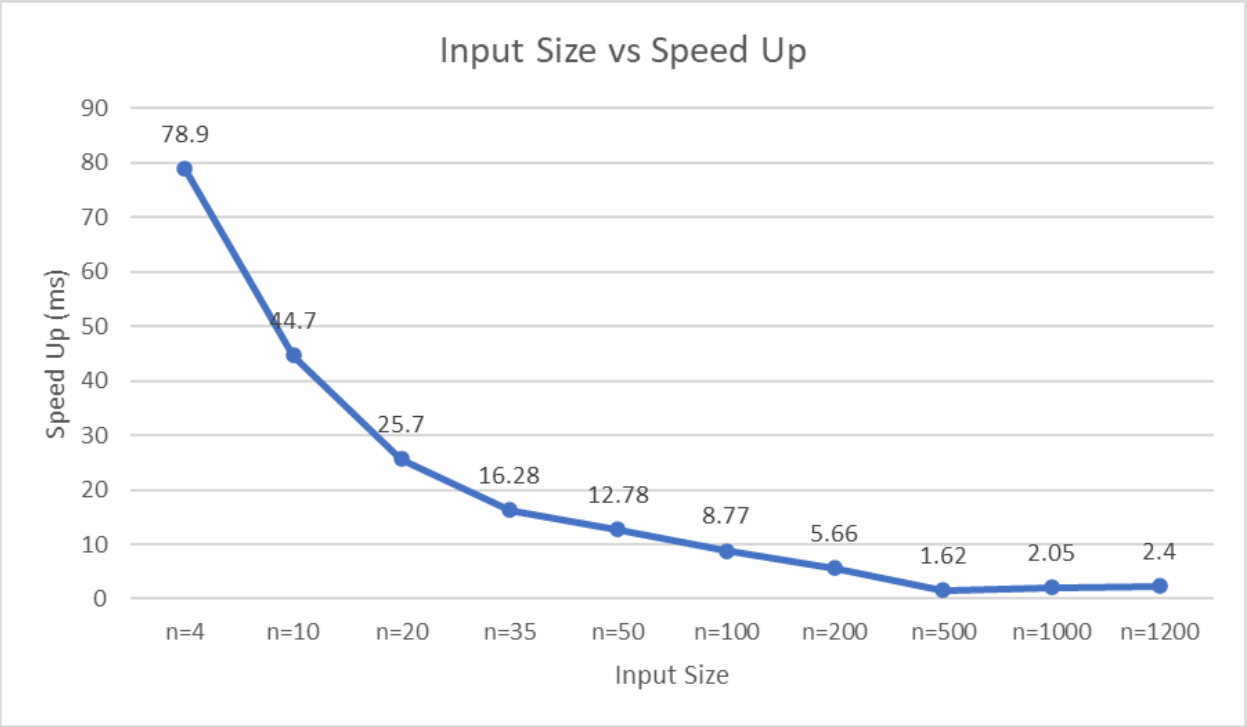
Speed Up (1000) = 117222 / 57139.6 = 2.0515

Speed Up (1200) = 165416 / 68231.3 = 2.424

The work / cost and efficiency calculations don't make sense for GPUs since the number of processors don't vary.

Graphs:





Conclusion:

In summary, the parallel A Star Search has a great speed up on CUDA as compared to the serial version. Other applications include Protein Design, Sliding Puzzle, etc. The proposes the first A* search algorithm on a GPU platform. The experiments stated in the paper and proved in our project have demonstrated that the GPU-based massively parallel A* algorithm can have a considerable speedup compared to the traditional sequential A* algorithm, especially when search space is exponential. By fully exploiting the power of the GPU hardware, our GPU based parallel A* can significantly accelerate various computation tasks.

References:

1. Massively Parallel A* search on a GPU, by Yichao Zhou and Jianyang Zeng (https://yichaozhou.com/publication/1501massive/paper.pdf)
2. Massively Parallel A* Search on a GPU: Appendix (https://yichaozhou.com/publication/1501massive/appendix.pdf)
3. We also used a ton of links to learn more about parallelization on GPUs, avoiding race conditions, memory allocation techniques, etc.!

To dos:

1. Proof read on Wednesday once all changes are done : Both

2. Print and submit to professor on Thursday : Priya