Parallelizing Rapidly-exploring Random Tree (RRT)

Team 04

AMAN MEHRA, PARAS SHARMA, and SAVIT GUPTA

1 INTRODUCTION

In this project we aim to parallelize a path planning algorithm known as Rapidly-Exploring Random Tree (RRT) on a continuous 2D space. The RRT algorithm is amenable to parallelization on the GPU due to the SIMD nature of the collision check against obstacles (which can be in the millions). In this project, first we shall parallelize these collision checks and expect to see considerable speedup over the CPU based implementation. Subsequently, we propose to explore and implement two previously unexplored optimizations to the parallel implementation of the RRT algorithm. A Quad-tree data structure over the 2D space can be used to reduce the total number of obstacle checks and potentially lead to further improvements on the naive parallel implementation. Additionally, we will try to parallelize the tree based exploration strategy of RRT to explore multiple paths in parallel. We would like to note at this point that to the best of our knowledge, both these extensions have not been previously experimented with on the GPU and it is unclear whether they would lead to a significant speed up. Nonetheless, it is a promising direction and one that we shall explore in this project.

2 LITERATURE REVIEW

There has been developments in parallelizing RRT, Bialkowski, et al [1] found a bottleneck of performance while checking for collisions for each algorithm and hence parallelized it. [2] has an implementation of the [1]. [4] discussed about parallelzing the nearest neighbour search while associating a new sample in the tree. As in [5], code for a parallel nearnest neighbour search with parallel collision can be found. As discussed in [1], obstacles are one of the main bottle of the algorithm, we thought of implementing Quad-tree to perform fast collision checks. Quadtree can be used to find obstacles near to the new explored position. Quadtree are tree based data-structure and can be parallelized as well which again can boost the speed. In [3], they discuss about parallelizing the Quadtree.

Authors' address: Aman Mehra, aman1707@iiitd.ac.in; Paras Sharma, paras17250@iiitd.ac.in; Savit Gupta, savit17098@iiitd.ac.in.

3 MILESTONES

S. No.	Milestone	Status	Contributors
	Mid evaluation		
1	Implement Exploration (RRT) on CPU	Completed	Paras(40%), Savit(60%)
2	Implement Exploration (bi-RRT) on CPU	Completed	Aman(100%)
3	Implement Collision Check (RRT) on CPU	Completed	Paras(100%)
4	Implement Quad-tree on CPU	Completed	Aman(60%), Savit(40%)
	Final evaluation		
5	Integrate Quad-tree with RRT on CPU	Completed	Savit(50%), Aman(50%)
6	Parallelize Collision Check (RRT) on GPU	Completed	Paras(60%), Savit(40%)
7	Parallelize Quad-tree on GPU	Incomplete	-
8	Integrate Quad-tree with RRT on GPU	Incomplete	-
9	Parallelize Exploration on GPU	Incomplete	-
10	Parallelize bi-RRT on GPU	Incomplete	-
11	Evaluating and benchmarking implementations	Partially Completed	Savit(30%), Aman(40%), Paras(30%)
	Miscellaneous		
12	Report and Presentation	Completed	Savit(25%), Aman(50%), Paras(25%)

3.1 Overall Contribution

Code Writing - Paras(50%), Savit(30%), Aman(20%)
Analysis/Debugging - Savit(50%), Aman(30%), Paras(20%)
Bench-marking/Map-making - Aman(50%), Paras(50%)
Presentation - Aman(50%), Savit(30%), Paras(20%)

4 MID EVALUATION CHECKPOINT

For the mid evaluation we have successfully implemented the milestones we had set out to do, i.e. the first four milestones. In addition, we have also completed tasks 5 and 6 along with all the evaluation, analysis and benchmarking on these completed tasks.

4.1 Identified Performance Bottlenecks

Here we list down the performance bottlenecks that were observed in the naive RRT algorithm implementation.

- Collision Check - Checking for collision against obstacle list has a complexity of O(n) operation, thus this scales badly when the number of obstacles is large.
- Nearest Neighbour Check - When adding a randomly sampled point to the search tree, the complexity of finding the nearest tree node to expand scales linearly in number of nodes in the search tree.
- Points of Exploration - At each iteration a single random point is sampled and a single selected tree node is grown in its direction.

4.2 Proposed Optimizations

Here we list down the the optimization that we came up with to reduce the performance bottlenecks observed in the previous section.

- Collision Check - By parallelizing this operation on the GPU, we can reduce this to a constant time check by assigning a small fixed number of obstacles to a kernel thread.
- Collision Check - A serial alternative is to use the Quadtree data structure to subdivide the search space to achieve a log(n) time complexity.
- Nearest Neighbour Check - This can be treated as a map reduce problem.
- Points of Exploration - Simultaneously add multiple points. This can be incorporated with the nearest neighbour map reduce to simultaneously perform checks on a small finite number of points.

4.3 Work Done

For the current deadline we have implemented all the CPU baselines for both RRT and Bi-RRT. In addition we have completed optimizations, both Quadtree and GPU based, for the collision check. We have benchmarked both algorithms with and without these optimizations across a variety of maps with varying number of obstacles as well as different distributions of obstacles (structured or uniformly distributed at random)

4.4 Maps

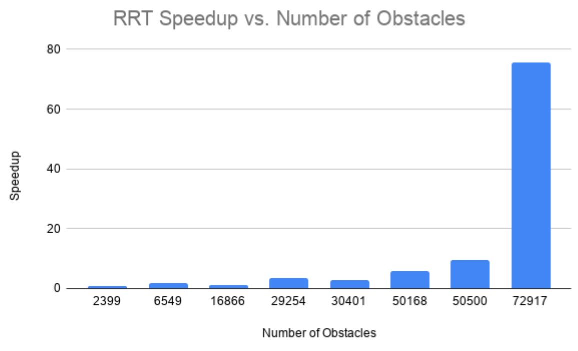
We have tested our implementations on 8 different maps. Maps 1-3 and 5-6 are structured maps with hand drawn obstacles. Essentially, these maps contain obstacles with pixel showcasing a high degree of spacial locality. On the other hand maps 7-8 are unstructured maps with obstacles derived from single pixels sampled uniformly at random across the exploration space. Map 4 is a hybrid map with some structured obstacles and a large number of unstructured obstacles.

The following table presents the details about the various maps

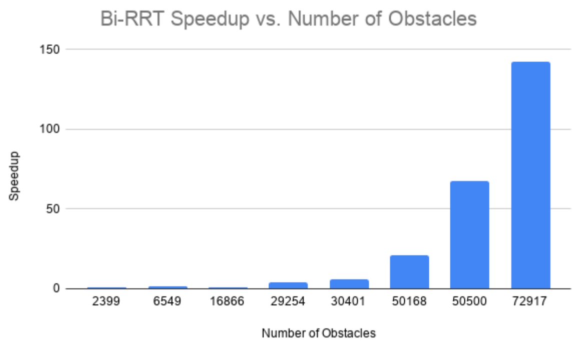
Map Name	**Obstacles**	**Type**
Map 1	16866	Structured
Map 2	2399	Structured
Map 3	6549	Structured
Map 4	72917	Hybrid
Map 5	50168	Structured
Map 6	30401	Structured
Map 7	29254	Unstructured
Map 8	50500	Unstructured

4.5 Quadtree Speedup

Now we observe the speedup graph for the Quadtree data structure. We can notice in the figures below that a significant speedup is observed when the number of obstacles is large. This makes sense as we are fully maximizing the benefit of the logarithmic run-time of the Quadtree. In addition, we note that the speedup is also dependent on the type of obstacles. Concretely, unstructured maps depict a higher speedup as compared to structured maps. This is because in the unstructured setting, it is highly likely that the collision check returns true in the first Quadtree leaf node bounding region. On the other hand, in the structured environment, the collision check will have to look in adjacent boxes which can increase search time upto $9x$.



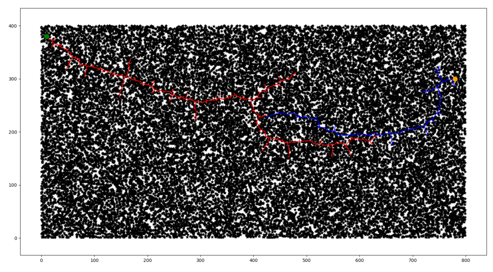
(a) RRT Quadtree Speedup



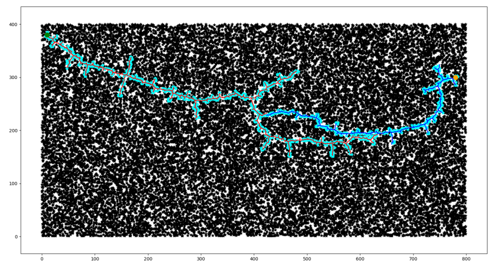
(b) Bi-RRT Quadtree Speedup

Fig. 1. Speedup Graphs when optimizing with Quadtree

Figure 2 illustrates an example of why Quadtree demonstrates such a high speedup especially in unstructured environments. The small fraction of obstacles used for collision check, especially so close to the search path reinforce our observations.



(a) Search tree



(b) Obstacles checked over entire run. The blue pixel depict those obstacles that the Quadtree check for collision against

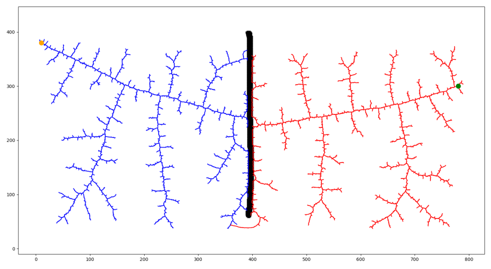
Fig. 2. Map7 Exploration

4.6 Map Demos

Next we show some of the paths that were found by our implementation in various map setting.

Figure3 shows the difference in the search trees across RRT and Bi-RRT, clearly highlighting the superiority of Bi-RRT in finding a search path faster.

Figure 4 5 6 show the search tree for various maps and also show the zoomed in point of termination of the algorithm (note that these may be for different runs of the algorithm). Figure 7 show the paths for Map4 and Map6


(a) Bi-RRT

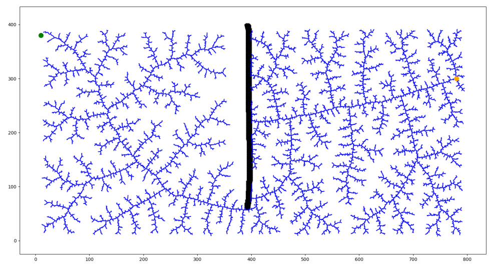

(b) RRT

Fig. 3. The search tree compared across RRT and Bi-RRT

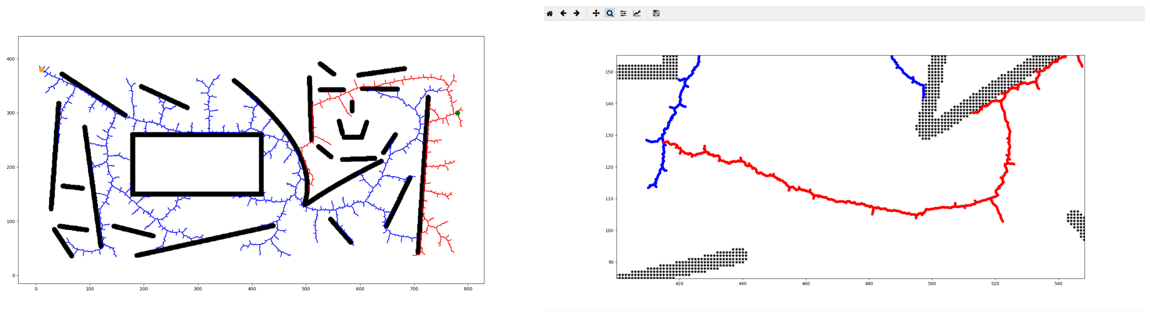


Fig. 4. Map 1 Exploration

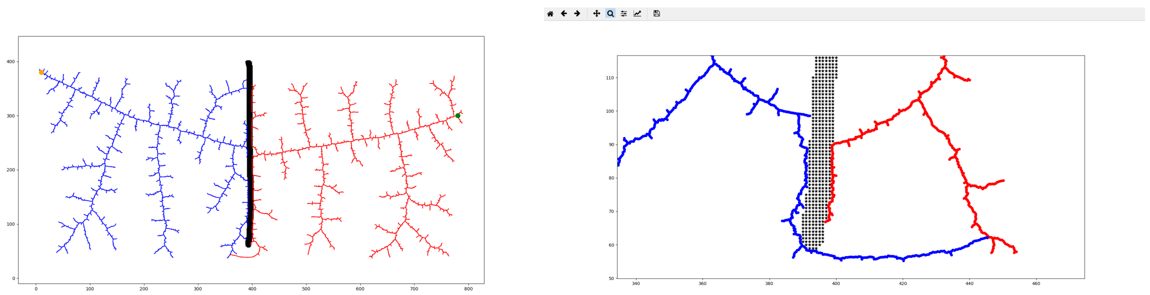


Fig. 5. Map2 Exploration

4.7 GPU Optimization

Figure 8 depicts the speedup achieved by RRt and bi-RRT on the GPU vs the CPU implementation.

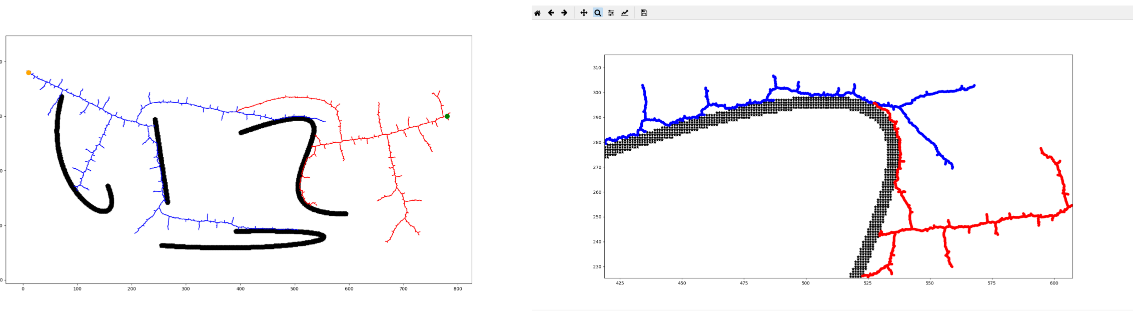


Fig. 6. Map3 Exploration

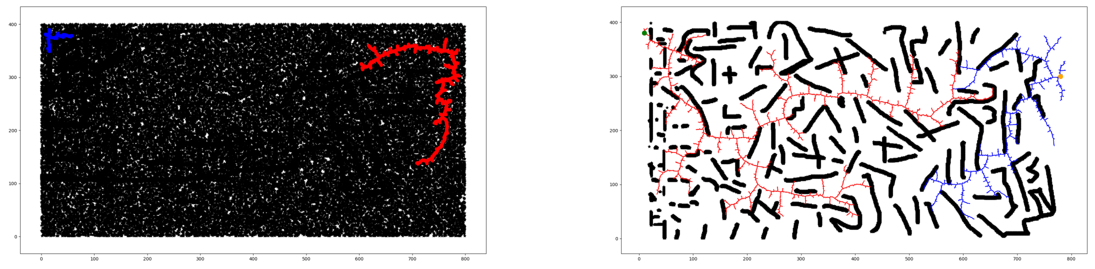
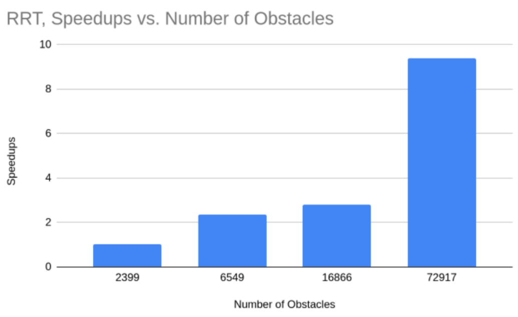


Fig. 7. Map4 and Map6 Exploration



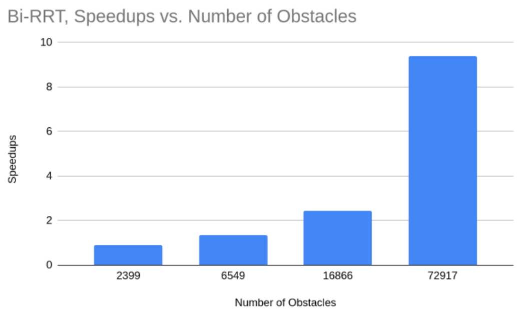
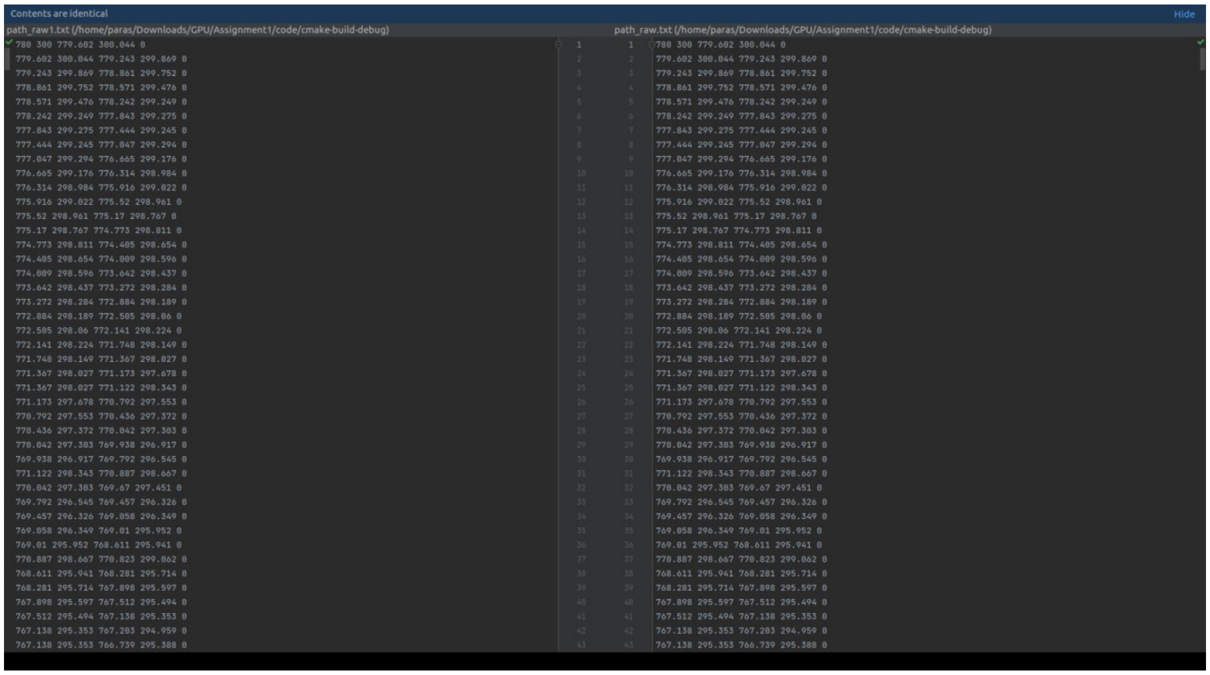


Fig. 8. Speedup graph for GPU vs CPU for RRT(left) and Bi-RRT(right)

To verify the correctness of our implementation, we used the diff function in CLion. Figure 9 contains the screenshot of the diff command's output.



Contents are identical

Hide

path_raw1.txt (/home/paras/Downloads/GPU/Assignment1/code/cmake-build-debug)

```
780 300 779.602 300.044 0
779.602 300.044 779.243 299.869 0
779.243 299.869 778.861 299.752 0
778.861 299.752 778.571 299.476 0
778.571 299.476 778.242 299.249 0
778.242 299.249 777.843 299.275 0
777.843 299.275 777.444 299.245 0
777.444 299.245 777.047 299.294 0
777.047 299.294 776.665 299.176 0
776.665 299.176 776.314 298.984 0
776.314 298.984 775.916 299.022 0
775.916 299.022 775.52 298.961 0
775.52 298.961 775.17 298.767 0
775.17 298.767 774.773 298.811 0
774.773 298.811 774.405 298.654 0
774.405 298.654 774.009 298.596 0
774.009 298.596 773.642 298.437 0
773.642 298.437 773.272 298.284 0
773.272 298.284 772.884 298.189 0
772.884 298.189 772.505 298.06 0
772.505 298.06 772.141 298.224 0
772.141 298.224 771.748 298.149 0
771.748 298.149 771.367 298.027 0
771.367 298.027 771.173 297.678 0
771.367 298.027 771.122 298.343 0
771.173 297.678 770.792 297.553 0
770.792 297.553 770.436 297.372 0
770.436 297.372 770.042 297.303 0
770.042 297.303 769.938 296.917 0
769.938 296.917 769.792 296.545 0
771.122 298.343 770.887 298.667 0
770.042 297.303 769.67 297.451 0
769.792 296.545 769.457 296.326 0
769.457 296.326 769.058 296.349 0
769.058 296.349 769.01 295.952 0
769.01 295.952 768.611 295.941 0
770.887 298.667 770.823 299.062 0
768.611 295.941 768.281 295.714 0
768.281 295.714 767.898 295.597 0
767.898 295.597 767.512 295.494 0
767.512 295.494 767.138 295.353 0
767.138 295.353 767.203 294.959 0
767.138 295.353 766.739 295.388 0
```

path_raw.txt (/home/paras/Downloads/GPU/Assignment1/code/cmake-build-debug)

```
780 300 779.602 300.044 0
779.602 300.044 779.243 299.869 0
779.243 299.869 778.861 299.752 0
778.861 299.752 778.571 299.476 0
778.571 299.476 778.242 299.249 0
778.242 299.249 777.843 299.275 0
777.843 299.275 777.444 299.245 0
777.444 299.245 777.047 299.294 0
777.047 299.294 776.665 299.176 0
776.665 299.176 776.314 298.984 0
776.314 298.984 775.916 299.022 0
775.916 299.022 775.52 298.961 0
775.52 298.961 775.17 298.767 0
775.17 298.767 774.773 298.811 0
774.773 298.811 774.405 298.654 0
774.405 298.654 774.009 298.596 0
774.009 298.596 773.642 298.437 0
773.642 298.437 773.272 298.284 0
773.272 298.284 772.884 298.189 0
772.884 298.189 772.505 298.06 0
772.505 298.06 772.141 298.224 0
772.141 298.224 771.748 298.149 0
771.748 298.149 771.367 298.027 0
771.367 298.027 771.173 297.678 0
771.367 298.027 771.122 298.343 0
771.173 297.678 770.792 297.553 0
770.792 297.553 770.436 297.372 0
770.436 297.372 770.042 297.303 0
770.042 297.303 769.938 296.917 0
769.938 296.917 769.792 296.545 0
771.122 298.343 770.887 298.667 0
770.042 297.303 769.67 297.451 0
769.792 296.545 769.457 296.326 0
769.457 296.326 769.058 296.349 0
769.058 296.349 769.01 295.952 0
769.01 295.952 768.611 295.941 0
770.887 298.667 770.823 299.062 0
768.611 295.941 768.281 295.714 0
768.281 295.714 767.898 295.597 0
767.898 295.597 767.512 295.494 0
767.512 295.494 767.138 295.353 0
767.138 295.353 767.203 294.959 0
767.138 295.353 766.739 295.388 0
```

Fig. 9. Diff output of CPU and GPU outputs. We validated our implementation by verifying that there is no difference in the computed nodes

REFERENCES

[1] Joshua Bialkowski, Sertac Karaman, and Emilio Frazzoli. 2011. Massively parallelizing the RRT and the RRT. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 3513–3518.

[2] Josh Cohen and Boston Cleek. [n.d.]. CUDA RRT. https://github.com/bostoncleek/CUDA-RRT

[3] Maria Kelly and Alexander Breslow. [n.d.]. Quadtree Construction on the GPU: A Hybrid CPU-GPU Approach. https://www.sccs.swarthmore.edu/users/10/mkelly1/quadtrees.pdf

[4] S. Sengupta. 2006. A Parallel Randomized Path Planner for Robot Navigation. *International Journal of Advanced Robotic Systems* 3, 3 (2006), 37. https://doi.org/10.5772/5730 arXiv:https://doi.org/10.5772/5730

[5] Anshul Sungra. [n.d.]. Parallel-Computing-of-Rapidly-exploring-random-tree-RRT. https://github.com/AnsSUN/Parallel-Computing-of-Rapidly-exploring-random-tree-RRT-/tree/master/parallel_RRT