Massively Parallelizing the RRT

Josh Cohen
Center for Robotics and Biosystems
Northwestern University
joshuacohen2019@u.northwestern.edu

Boston Cleek
Center for Robotics and Biosystems
Northwestern University
bostoncleek2020@u.northwestern.edu

***Abstract*—The goal of this project was to explore the possibilities of utilizing massively parallel processors to parallelize the Rapidly-Exploring Random Trees algorithm. Collision checking was found to be the most significant bottleneck, therefore research efforts were focused on paralellizing this aspect of the algorithm. A naive approach was implemented and analyzed, while the infrastructure for further optimizations were built out and the nature of those optimizations are discussed. It was found that for search spaces in which the obstacle count was greater than 2048 the massively parallel implementation on a GPU outperformed the same algorithm run on a CPU, further it was found that future implementations should work to ensure CUDA threads are performing useful computation with techniques such as binning and thread coarsening.**

***Index Terms*—RRT, CUDA, Parallelization**

I. Introduction

A. Why Parallelize the RRT

The Rapidly-Exploring Random Tree (RRT) algorithm is of canonical importance to the field of robotics. With known start, goal, and obstacle locations the RRT attempts to find a route from the start to the goal by following a set of simple rules (detailed in B). Importantly, the RRT scales to arbitrarily high dimensionality. The RRT therefore allows path planning to occur in multiple dimensional spaces with obstacle avoidance. This can be applied to physical obstacles in space or operational obstacles such as singularities in the kinematics of a robot. The RRT is prime for parallelization because for it is physical system agnostic and all collision checking happens in parallel. Effectively parallelizing this algorithm would allow robotiscits to more rapidly explore spaces which can have many positive downstream effects.

B. What is the RRT

The general procedure for constructing and RRT is as follows:

1. Place random point in space
2. Find nearest vertex in tree
3. Take a unit step in direction of randomly placed point
4. **Collision check to see if newly placed point or the edge connecting the point collide with any obstacles**
5. If no collisions add the new point to the tree
6. Check if there now exists a collision-free path to goal
7. Else loop

The following allows for arbitrarily dimensioned spaces to be searched for a path between two points. An example of the resulting solution can be seen in fig: 1 where the blue tree is the explored space and the yellow overlayed tree is the recursively found path between the start and goal points.

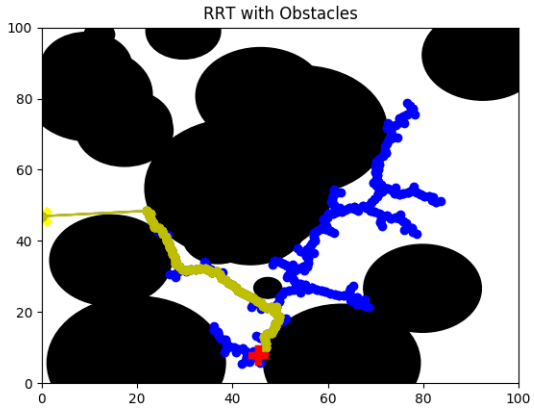

Fig. 1. CPU RRT

Result: Path Complete
initialization;
circles [] = randomlyPlaceCircles()
qinit = initConfiguration(start, goal)
G.init(qinit)
while *Goal not reached* **do**
  instructions;
  qrand = randomPoint()
  qnear = nearestVertex()
  qnew = newConfiguration()
  bool collision_flag = false
  CUDA_Collision_Check(qnew, qnear, circles, collision_flag)
  if *Not collision_flag* **then**
    G.add_vertex(qnew)
    G.add_edge(qnew, qnear)
  else
    continue
  end
end
return G

Algorithm 1: CUDA RRT

C. Computation Bottleneck

The time complexity for the RRT governed by graph operations which is $O(\log n)$ for nearest neighbor search and

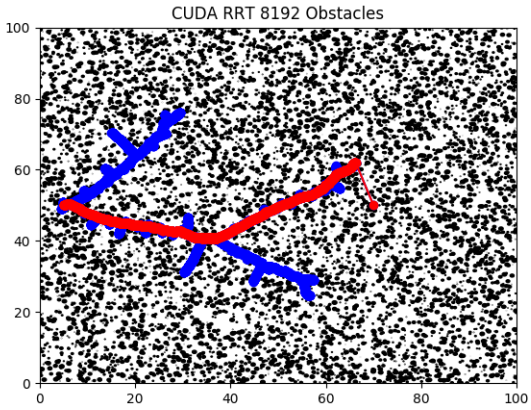

Fig. 2. Naive CUDA Implementation

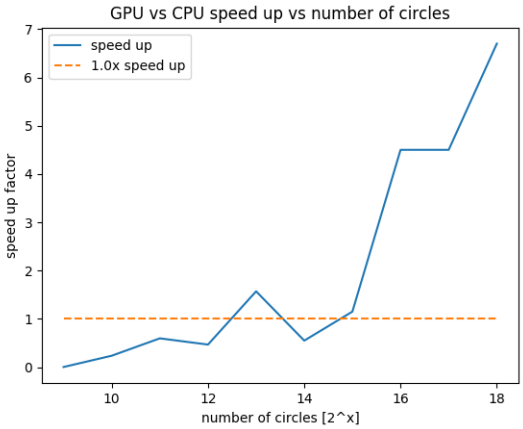

Fig. 3. Performance Comparisons Between GPU and CPU

insertion, where n is the number of vertices in the graph. The collision checking is limited by $O(n)$, where n is the number of circles. This paper considers the case where there are more circles than vertices. This assumption is evident in the resulting size of the graph and number of circles used in the experiments.

II. Naive Parallelization

The goal of the naive implementation was to have a working parallelized algorithm that served as an effective baseline to examine the results of parallelization on the algorithm. This entailed creating a collision checking kernel invoked by the CPU once per loop of the main algorithm. This kernel took the proposed new point, the closest existing vertex, and a list of the obstacles. The kernel invoking function then chose block and grid dimensionality such that a single thread could be assigned to every obstacle for collsiion checking purposes. Work was done to ensure coalesced accesses of the global circle data as well as utilizing shared memory for writes and reads to the global collision flag variable. The x, y center location of each circle and the radius were loaded into separate arrays. This ensured the global memory reads were coalesced.

The performance of the naive implementation on the GPU was tested against the standard CPU implementation for a quantity of obstacles that ranged from 2^9(512)-2^{19}(524,288) As can be seen in fig: 4 for trials with less than 512 obstacles the CPU was significantly faster. However, once the number of obstacles exceeds 2048, GPU performance begins to dominate CPU performance. It is important to note that for the last 3 trials the map was so densely packed that an RRT over 100,000 iterations could not find a path to goal; however, the CUDA GPU implementation calculated this much more rapidly than the CPU.

III. Optimizing for Parallelism

Though the basic optimizations of coalesced accesses to global memory and utilization of shared memory were already implemented, there are still a wealth of further optimizations that can be taken. When re-evaluating the algorithm for these further optimizations it was found that the prime benefit to be gained was in trimming wasted computations which manifest as useless thread work. The algorithm collision checks against all obstacles in the search space regardless of the unit step size to the prospective new point. Since the prospective new point can never be more than a unit step away all obstacles searched not unit step radius of the prospective new point are wasted computations.

A. Binning

There is no need to check collisions outside of the unit step distance for placing a new vertex. Therefore, the obstacles are reprocessed and placed into bins before the start of the RRT. An max initial number of circles per bin is set. If a bin overflows the computations are sent to the CPU. The max number of circles per bin was set to a 100. When more than 100 circles per bin were allocated the space to be explored was too crowded to achieve a path to goal. A histogram was used to keep track of the number of circles per bin. When a circle crossing the boundary into another bin it is included in that bin as well. Given the location of a new vertex the bin or bins that need to be checked for collision can easily be determined. Although the dramatically improves performance it decreases the need for parallelization.

B. Further Optimizations

- Each block constructs a separate RRT, a seperate kernel determines which one contains the ”optimal” path
- Use thread coarsening by using each thread to check many obstacles and increasing instruction level parallelism
- Use GPU based recursion for dynamic work generation
  - In real world examples, obstacles are clustered spatially
  - dyanmic work generation for bin sizing can ensure that bins are made to contain a set number of

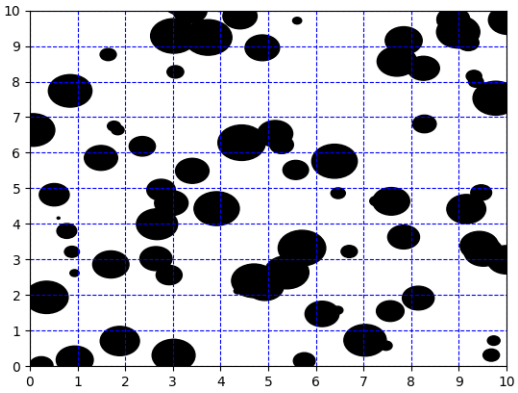

Fig. 4. Preprocessed Circle Binning

obstacles such that all blocks are utilizing threads for useful computation

IV. Conclusion

The RRT is a highly utilized path finding algorithm in the robotics community. It's nature incentives and facilitates parallelization. As can be seen in fig: 4, the naive CUDA implementation demonstrated promising results for RRT searches with greater than 2048 obstacles. Especially considering how the number of obstacles can scale in higher dimensional search spaces this advantage over CPU based computation can be effectively leveraged. However, this naive CUDA implementation of the RRT still must reconcile it's wasteful use of thread computation on obstacles that the path, by the nature of its step size, will never collide with. Pre-processing our input to implement binning allows for spatial localization which allows for all threads to do meaningful work by only checking bins within the unit step distance of the new point, thereby only checking the obstacles that can be collided with. While this reduces thread level parallelism due to not searching tens of thousands of obstacles, the work performed is more meaningful and this also opens opportunities for novel optimizations. Such optimizations could look like computing multiple RRTs simultaneously, stitching together partially computed RRTs, or using dynamic work generation to ensure that blocks have meaningful work for their threads to compute. Through this work it can be seen that while the naive implementation of a massively parallelized RRT with CUDA did yield performance benifits when the obstacle count was greater the 2048, the RRT algorithm and CUDA do not mesh perfectly due to the nature of the parallelizable aspect of the RRT, the collision checking, also having a dependence on spatial locality. With the goal of implementing massively parallelized path finding algorithms however, we can utilize many of the proposed optimizations to take inspiration form the RRT algorithm while also making it run optimally on a GPU.

References

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