GPU Accelerated Dynamic Obstacle Avoidance and Path Planning

Zhihao Ruan, Nathan Brown, Gregory Meyer, Martin Deegan

***Abstract*—Human robot interaction is becoming more commonplace. Factories will want more interaction with robotic arms and humans workers. However, current factory robots are unsafe to operate around humans due to their lack of perception. Thus, an efficient and safe motion planner is required to avoid dynamic obstacles. We present a GPU accelerated approach to motion planning around dynamic obstacles on a small scale using 3D scene reconstruction and a GPU accelerated stochastic planning solver.**

I. Introduction

Human heavily rely on robots for manufacturing and production. Robots can operate with greater speed and precision than even the most skilled of workers, cementing their role as an essential part of assembly lines. With the help of assembly robots, cars on an assembly line can be produced within hours. In comparison, it might take days or even weeks for human workers to do so. Furthermore, it costs much more for the factory to hire human workers than to purchase and maintain assembly robots. We predict that robots will only become more popular in modern life due to their high efficiency and economic benefits.

However, establishing safe and stable systems of human-robot interaction is still an open question. News of assembly robots hurting people is still reported from time to time. Typical robots move fast and are equipped with high-power actuators, making them likely to collide with any obstacles in their path. As a result, high-performance robots that can dynamically detect and avoid obstacles are in high demand.

We propose a novel solution on building a fast robot arm with dynamic obstacle avoidance. We analyze this problem in two aspects: 1) the algorithm should be running in real time and 2) the algorithm should be able to detect any kind of moving obstacles. To approach this problem, we implemented a high performance motion planning algorithm that can be parallelized on a graphics processing unit (GPU)[1]. We also constructed a 3-D point cloud with three Intel Realsense D415 cameras that was subsequently converted into an occupancy grid at 30 Hz. This occupancy grid provided real-time obstacle detection for motion planning.

All source code is available at https://github.com/EECS467-DOAPP/doapp. Our software was built using the Robot Operating System (ROS) framework and is formatted as a set of packages for ROS Melodic Morena (ROS Melodic).

II. Mapping

A 3-dimensional representation of the scene around the robot arm is required to avoid obstacles. We accomplished this goal using three Intel Realsense D415 depth cameras positioned around the arm.

A. Calibration

Calibration is a crucial step to building a 3D scene. We used a flat grid of tag25h9 AprilTags [2] to localize the cameras. This grid was a mosaic of all 34 tag IDs, but we exclusively used tags 5, 9, 25, and 29 for localization. These tags were placed on the corners of the cutout so that at least three of them could be seen at all times, no matter the orientation of the Rexarm. The arm was placed in the center of these four tags. Each tag had a side length of 2.8cm and the tags were separated by 1.3cm of space. Using the AprilTags library, we were able to get the relative pose of each tag with respect to the base of the arm, and by extension, the pose of the cameras with respect to the arm. The four tags were combined in a tag bundle to jointly estimate the pose using multiple tags. Using a tag bundle provided greatly increased calibration accuracy.

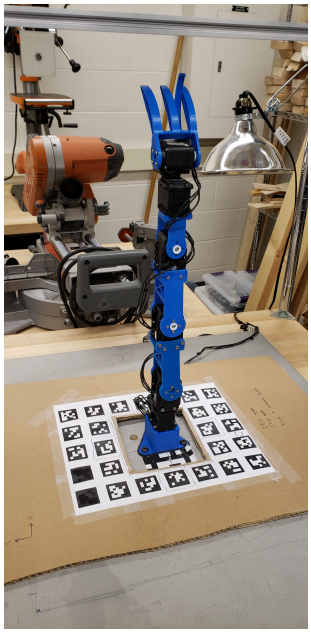

Fig. 1. Robot arm setup with the AprilTag mosaic surrounding the base.

In our initial design, we ran calibration at the same time as mapping. However, we found that the point cloud would jitter due to noise in the reported tag bundle pose. Sometimes, the orientation could be off by as much as 90 degrees. To remedy this, we save the camera poses to a file in a calibration phase, then load these parameters as static poses during the mapping phase of operation.

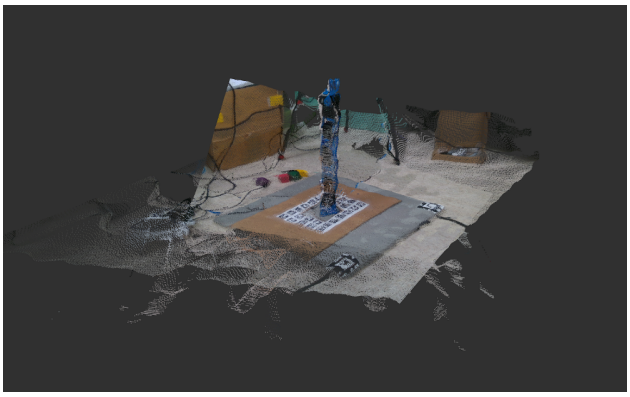

Fig. 2. Raw point clouds created by the depth cameras during calibration. Points are colored according to the camera images. AprilTags can be seen surround the arm base.

B. Mapping

Mapping was performed by deprojecting the depth images into point clouds. We used factory calibrated camera intrinsics for this step. Using a 480 $\times$ 270 depth image produced a point cloud with 129,600 points in the camera's coordinate frame. The cloud was then filtered in two steps. First, we cropped out points outside a 1 meter box around the arm frame. Next, we downsampled the point cloud using a voxel grid with a cell size of 1.5cm. Finally, we transformed the three point clouds into the arm base coordinate frame and merge the point clouds, giving us a final point cloud with roughly 30,000 points.

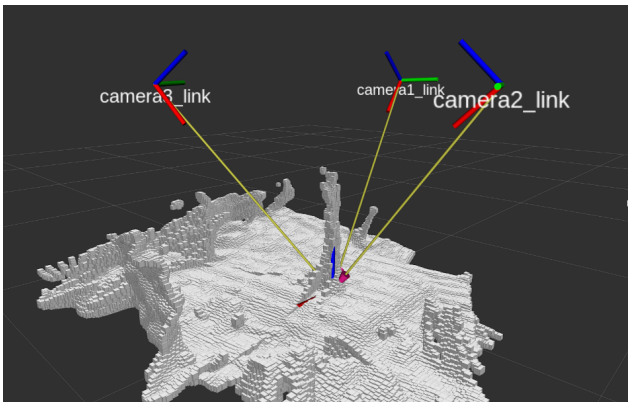

Fig. 3. Processed point cloud during operation. Points are represented as boxes. The calibrated camera poses are also visible.

C. Distance Grid

In order to compute collisions in our randomized trajectory optimization, we compute a distance grid using the pointcloud produced by the mapping. Construction of the distance grid is carried out in two steps. First, a k-d tree is built using the mapping pointcloud. K-d trees offer logarithmic time nearest neighbor lookups in the typical case, significantly speeding up these kinds of queries when compared to brute force searches. We use the C++ library nanoflann as an out-of-the box k-d tree implementation, both to increase performance and to decrease engineering time. After the k-d tree was built, we divided up the distance voxel grid by height and ran parallel queries on the k-d tree. Each vertical slice of the voxel grid was carefully aligned to 32-byte boundaries to enable us to issue vectorized AVX loads and stores of the data, accelerating memory accesses and increasing throughput. These two optimizations were enough to allow the distance grid step to run at 30Hz on an octa-core laptop, but most of this time was spent querying the k-d tree for the nearest neighbor.

We also considered accelerating construction of the distance grid by moving to a GPU-based algorithm, but initial results showed performance regressions instead of gains. Further work should be done so that the k-d tree can be built and subsequently queried in parallel on the GPU.

III. Motion Planning

Our motion planner is heavily inspired by the work done by Park et al [1].

A. Optimization based trajectory planning

Our motion planner plans in the configuration space of the robot arm, where the dimensionality is equal to the number of joints. While our arm does have 6 joints, we noted that the last two motors simply spin the end affector and open/close the gripper. Thus, the motion planner assumes these motors can be held in the same place without much impact on the resulting trajectory. Under this assumption, the motion planner can reduce the dimensionality of the state space it explores by a factor of 1.5. Each trajectory has a static amount of time to finish. In our implementation, this amount of time was 0.1 seconds to ensure that the arm moved quickly. The waypoints are equally spaced in time along this trajectory.

Our motion planner is a stochastic search optimization solver. Each trajectory is discretized into N waypoints, where each waypoint is a point in the configuration space. Given an initial trajectory, we generate M random noise vectors of the same dimension as the configuration space and apply them to the initial trajectory to generate M noisy trajectories. We then apply each noise vector to each each waypoint in a trajectory. Then, the cost of each noisy trajectory is calculated according to (1).

The lowest-cost trajectory, considering both the noisy trajectories and the initial trajectory, is preserved as the initial guess for the next iteration. The planner will execute as many iterations of the optimizing loop as it can before it runs out of time. See Algorithm 1 for more detail.

Algorithm 1: Motion Planning Algorithm

Input : Initial trajectory $traj$ to optimize, 3D obstacle distance grid, time expired flag
Output: Optimized trajectory and its score
best trajectory $\leftarrow traj$;
best score $\leftarrow$ `cost` $(traj)$;
while *time expired is False* **do**
  noise vectors $\leftarrow randomnoise$;
  noisy trajectories $\leftarrow$ noise vectors $+$ best trajectory;
  scores $\leftarrow$ [];
  for $noisytraj$ *in* noisy trajectories **do**
    scores $+=$ `cost` $(noisytraj)$
  end
  best score $\leftarrow min($scores $)$;
  best trajectory $\leftarrow traj$ with lowest score;
end
return best trajectory, best score;

We compute a linear interpolation between the start and end angles for each motor for the initial trajectory. Park et al. started with multiple random trajectories, but we did not find a good distribution to sample for these initial trajectories that was able to be optimized into a good trajectory in a reasonable amount of time. The noise vectors are generated by sampling each component of the vector from a standard normal distribution.

Our cost function is presented in (1). The cost function sums the cost of each waypoint along the trajectory, where each waypoint is scored with a collision factor and a smoothness factor. The collision factor ($does\_collide$) is determined by applying forward kinematics to each of the joint angles, then indexing into the 3D obstacle distance grid to determine the presence or absence of any obstacles. The smoothness factor is computed by the sum of squared accelerations ($\|\vec{a_i}\|_2^2$) for each motor. For the safety of the robot's operation, it is imperative that a trajectory with a collision has a higher cost than anay collision-free trajectory.

$$cost(trajectory) = \sum_{i=0}^{N} does\_collide(i) + \frac{1}{2}\|\vec{a_i}\|_2^2 \quad (1)$$

B. Exploiting Parallelism

There are multiple parallelizable aspects of the motion planner. First, Algorithm 1 can be run multiple times in parallel, and the best trajectory can be found by taking the minimum of each cost computed. Additionally, most steps of the algorithm can leverage parallelism themselves. See Table I for a breakdown of how many threads operate in parallel at each step of Algorithm 1.

TABLE I
THREAD BREAKDOWN PER STEP OF ALGORITHM 1

Step	# of threads
1. best trajectory $\leftarrow traj$	$N*4$
2. best score $\leftarrow cost(traj)$	1
3. noise vectors generation	$M*4$
4. noisy trajectories generation	$N*M*4$
5. Scoring noisy trajectories	$N*M*4$
6. best score $\leftarrow min($ all scores $)$	$N*M*4$
7. best trajectory $\leftarrow traj$ with lowest score	$N*4$

For step 1, each thread loads one joint value for one waypoint. For step 2, the lowest cost is shared across all threads. For step 3, each thread generates one random number for one value of one noise vector. For step 4, each thread computes one value of one waypoint using one noise vector. For step 5, one thread computes the acceleration for one joint at one waypoint on one noisy trajectory. $N*M$ threads are used to determine the number of collisions; each thread performs the forward kinematics and collision detection for all joints of a single waypoint on one noisy trajectory. For step 6, all threads that participate in scoring coordinate to maintain the best score using atomic memory operations. For step 7, each thread copies one value from one waypoint on the best noisy trajectory to the best trajectory variable. If no noisy trajectories resulted in an improvement, this step is a no-op.

On a GPU, all of the threads are partitioned into blocks of 1024. Threads within a block can synchronize and communicate through memory with the same latency as L2 cache. Threads in different blocks cannot directly communicate. While there are workarounds, they are typically not advantageous. Instead, we have multiple thread blocks execute Algorithm 1 independently, and have the CPU compare the results from multiple blocks. We settled on using 100 thread blocks, as that was adequate for obtaining good trajectories, and using more thread blocks slowed down the CPU-side code and increased the amount of data to be transferred between CPU and GPU. Our testing indicating that setting $N = 50$ generated smooth trajectories to follow. M was calculated as $floor(\frac{1024}{N*4})$ so as to not exceed the 1024 threads/block limit.

IV. TRAJECTORY FOLLOWING

Our 6-DoF robotic arm is assembled from six Dynamixel servos, including the gripper. Each servo can be rotated between $-\pi/2$ to $\pi/2$. The servo angle and velocity can be controlled by the Dynamixel SDK provided by ROBOTIS.

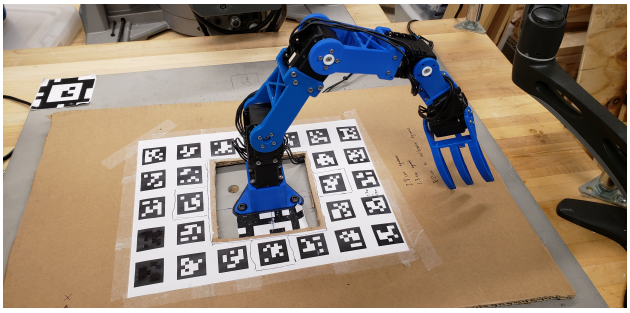

Fig. 4. The Rexarm uses six servos to facilitate manipulation of objects in free space.

Trajectories are a sequence of timestamps associated to joint angles. The trajectory follower captures trajectories published by the motion planner and executes them at 100Hz. We also publish joint angles for feedback in the motion planner. However, these angles are dead reckoned from the trajectory, as the SDK did not provide adequate support for reading servo angles without a severe loss in performance. Fortunately, we found that the servos' in-built position and velocity controls were precise enough for us to get away with this approximation.

V. Conclusion

We have presented our real-time trajectory planning system for a robot arm. We believe that with some additional work to further integrate the mapping and trajectory mapping systems, namely additional filtering of the mapping pointcloud, we would be able to use this method to facilitate safe operation of robot arms around humans. All of our components were running at performance targets, both in terms of runtime and latency requirements and in terms of the quality of our results, so the primary remaining obstacle towards completing the deployment of our methods would be this pointcloud filtration step. We believe that our methods can be used to great effect in factory and home environments, allowing safe and fast interaction with humans in the environments where robots can have the greatest impact.

References

[1] C. Park, J. Pan, and D. Manocha, "Real-time optimization-based planning in dynamic environments using gpus," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2013.

[2] J. Wang and E. Olson, "AprilTag 2: Efficient and robust fiducial detection," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016.