Particle-based SLAM for Autonomous Vehicles

A parallel particle-based SLAM implementation for the 15-418 Parallelism Competition (Spring 2017)

Nishad Gothoskar and Cyrus Tabrizi

SUMMARY:

We worked towards an implementation of Simultaneous Localization and Mapping (SLAM) that used GPU-accelerated with CUDA to make possible the use of large particle filters and full LIDAR 3D point clouds.

BACKGROUND:

Our project takes 3D point cloud data, 3D velocity measurements, and 3D gyro measurements (roll, pitch, yaw) from the KITTI Vision Benchmark Suite and uses SLAM to generate more accurate vehicle trajectories and better-aligned point cloud maps.

Figure 1: External system overview of our project

Running this algorithm across multiple frames of sensor data yields a full map of the environment you have moved through and the associated positions the vehicle was in with respect to that map.

The key operations are described in the figure below.

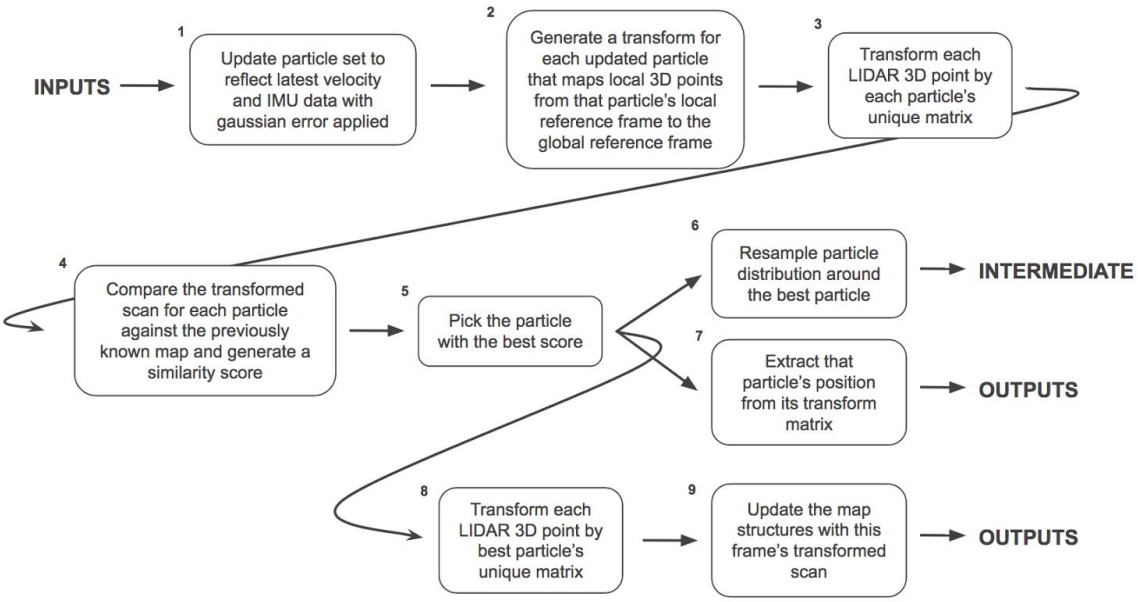

Figure 2: Internal system overview of our project

There is additional parallelism to exploit within some of these steps (for example, each particle's pose consists of 6 dimensions so there's some parallelism to exploit in the matrix updates and matrix arithmetic) but the most important parallelism to exploit is that across particles and that across LIDAR points (because of their large sizes).

The most computationally expensive step is #4 as the work involves comparing two sets of 100,000 3D points for each and every particle (one set is the new data transformed by that specific particle's pose and the other is the existing map).

The dependencies hidden in the chart above are as follows:

- Steps 1, 2, 3 and 4 are independent across particles and LIDAR points, but must happen sequentially with respect to each other. That is, any given combination of particles and LIDAR points can be streamed through steps 1, 2, 3 and 4 independently of any other combination of particles and LIDAR points going through these steps.
- There is a barrier between steps 4 and 5. That is, all the particles and all the LIDAR points must be processed in step 4 before we can evaluate which particle has the best score in step 5.

While there are lots of opportunities for parallelism (especially with the two-way independence of LIDAR points and particles), different implementations with face memory constraints. For example, if we implemented this pipeline with a barrier between step 3 and 4, we'd need to have storage for (N lidar points per scan) x (M many particles) and either allocate space on the GPU so that this step is fused with later ones or efficiently manage the transfer of memory back and forth from GPU and CPU space. Some of these considerations can be solved through our choice of algorithm and others are dealt with at the implementation level. For example, one algorithm for evaluating map-to-map similarity would involve operating over two maps, both of which have already been transformed into the global reference frame and stored into memory. Another approach would only require new memory for transforming one map, but then transforming that map to be in the reference frame of the second map, then doing the same comparison work as before. A third approach would keep both maps in memory in their respective reference frames and then fuse the transformation process into the comparison itself (i.e. compare this grid cell to another grid cell and compute the location of that grid cell through a transformation we do "live" for each grid cell, instead of transforming the points beforehand and comparing grid cells from the same location in both maps).

Of related importance is the locality present in the LIDAR scans before performing this transformation. In one approach, there's spatial locality in processing an entire scan for one particle, since the points are stored as a single array in memory. In a different approach, there is temporal locality if a single point from the scan is operated on by all of the particles at the same time.

The basic structure of our algorithm involves maintaining a set of particles that help us correct for error in IMU data (velocities and angle measurements). There are roughly 1000 particles (we can control the size of this “particle filter”). Each of these particles represents a possible pose the vehicle could be in.

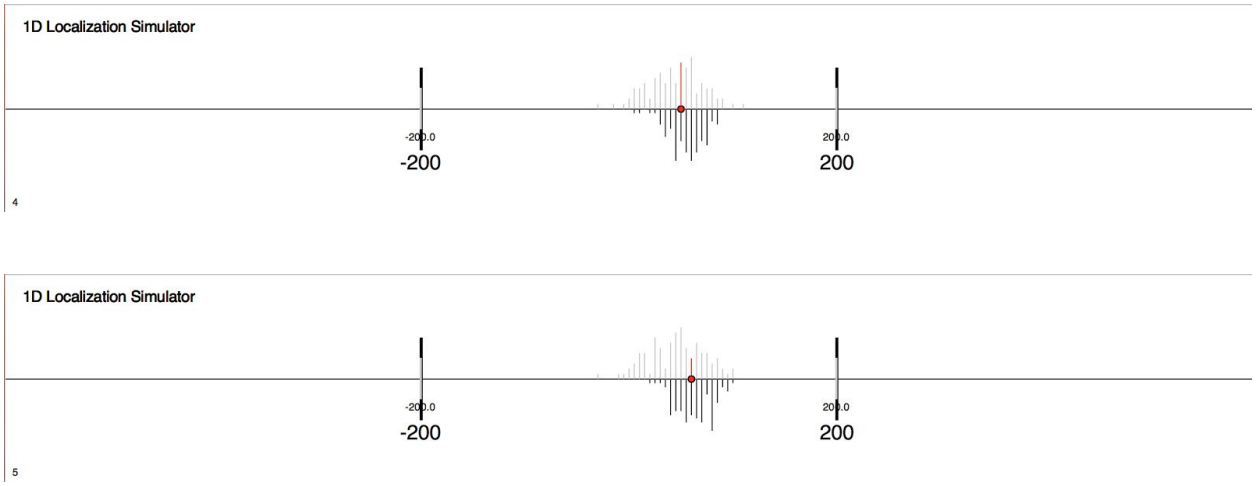

Figure 3: Screenshots from our 1D simulation of particle-based SLAM algorithm

To help us understand and learn how this algorithm worked, we implemented a 1D simulation in Python which let us to explore the behavior of these particle filters.

Initially, all of the particles are reset to the origin (in 3D, this means X = 0, Y = 0, Z = 0, Rx = 0, Ry = 0, Rz = 0, but in 1D, this would just be X = 0). Our map will be built relative to the origin and could be adjusted to a different reference frame later. As LIDAR data is received, it will have to be remapped from the local reference frame of the vehicle at that time back into this global reference frame. On the first time step, this means the only error in the map will be from the LIDAR data itself, not the IMU. SLAM will help us compensate for the IMU error that shows up later, but we’ll always be limited to the precision and accuracy of our LIDAR data.

For each timestep, the particles are offset by the velocity and rotation observed by the IMU. Error offsets for Vx, Vy, Vz, Rx, Ry, Rz are then sampled from normal distributions and added to each of the particles. If the actual sensor error distributions were known, these would be sampled instead. Normal distributions are used as an approximation of the real error distributions. These new particles poses are then used to offset every point in the LIDAR scan back into the original reference frame. For each of these

particles, a score is computed reflecting how well the LIDAR scan matches the previous map.

The particle with the best score is picked and the map that was transformed by its pose is merged with the preexisting map. Using the map similarity scores for each particle, the particles are resampled. The idea here is that the next set of particles will reflect the best of the previous set of particles, as determined in the map comparison stage.

This process repeats for each time step.

APPROACH:

Our final implementation used CUDA on an NVIDIA Tesla K40 GPU from the Latedays cluster. The Latedays cluster also features several Intel Xeon E5-2620 CPU's, which we only executed on with a single thread.

Each step involved a different mapping to the target machine. In the first step where we compute transforms for each particle, the mapping is from particles to threads. In the next steps, which involve applying this transform, we parallelize over the LIDAR point cloud instead.

For both the map composition and map comparison stages, we explored several data structures for representing the map before reaching our current approach.

Our initial approach was to just keep all the points but this approach does not scale for map composition as it involves keeping (T timesteps) * (N many points kept from each LIDAR scan) points in memory. Importantly, the high frequency of our sensor data (we had access to 10 Hz sensor data), meant that the changes from frame to frame were relatively little. While this overlap is both important for building confidence in and precision for features that are present in both and important for picking out moving objects, representing this by keeping all of the original points is very redundant.

Our next choice was to use some type of tree structure to reduce the redundancy in the image by mapping different parts of the tree to spatial regions in the map. This makes it easier for us to prevent the addition of points that are close together. This implementation would have been similar to a Kd or BVH tree. However for us to take advantage of this for our project, the tree data structure would have had to support parallel lookup, parallel removal and parallel inserts. Implementing this was infeasible

given the timeline of our project, but it would have brought significant performance improvements.

Our final choice was to aggregate points into a grid where each cell tracked how many points were mapped to that location in space. Thresholding on this “density map” allowed us to preserve important clusters and also prevent the addition of redundant data. The main problem with using a grid cell is that it fails to preserve the precision of the LIDAR data. This is mostly a problem for map comparisons where we’re trying to distinguish between very small transformation differences across particles. If we lose all the LIDAR precision due to voxelization or gridding (we mapped vertical columns of points into 2D), this will make comparisons ineffective. Gridding is also bad for LIDAR because the point clouds are very sparse. However, it is a more manageable approach to implement and relatively inexpensive from a compute standpoint.

We wrote our entire procedure from scratch, including all the math needed to implement SLAM correctly. This presented us with many issues during implementation that would have been avoided had we used outside libraries.

RESULTS:

The goal of our project was take the best of what we learned from 15-418 and use it to improve performance in particle-based SLAM for LIDAR. To evaluate our work, our goal was to analyze the performance of the individual components in our pipeline and also the output quality of the pipeline overall.

The experimental setup for evaluating our code was to vary the size of the workload as well as the kernel configuration (number of blocks). The first test we ran analyzed the performance of the particle update stage as we vary the number of particles. The reason this number is important is that a large particle filter allows the SLAM algorithm to consider more possible vehicle states, improving the estimate of the true pose. The next test we ran was to analyze the speedup of our LIDAR data transformations as we varied how many points we were modifying. This is an important number as map composition and map comparisons theoretically improve as the amount of data increases. This makes understanding the performance tradeoff of enabling that extra data processing an important step.

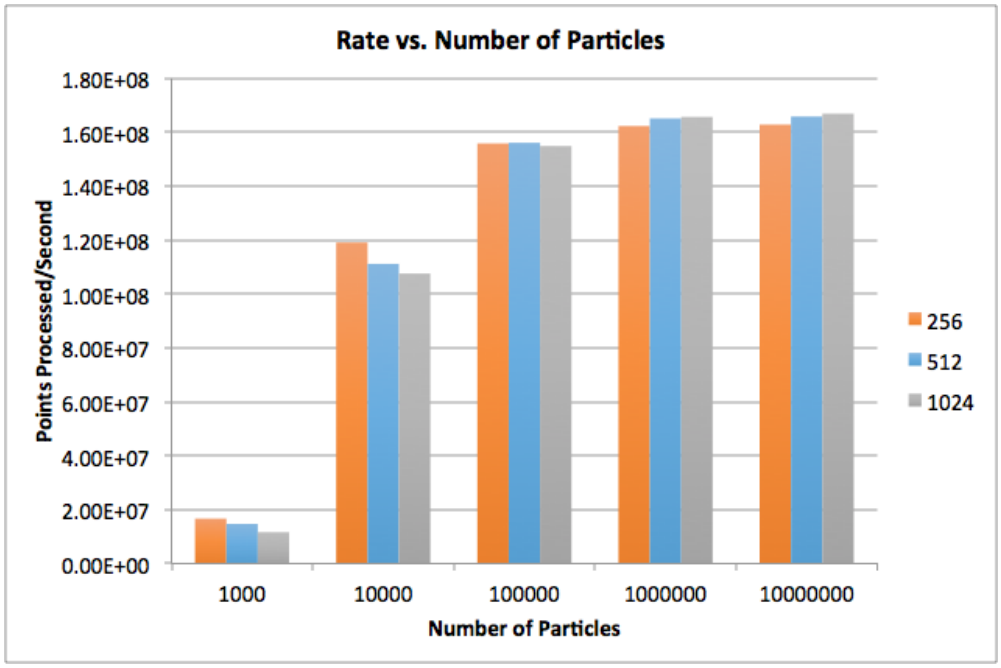

Figure 4: Analysis of the particle filter update step reveals the GPU's capacity. Note, this test was done on an NVIDIA Tesla K40m GPU.

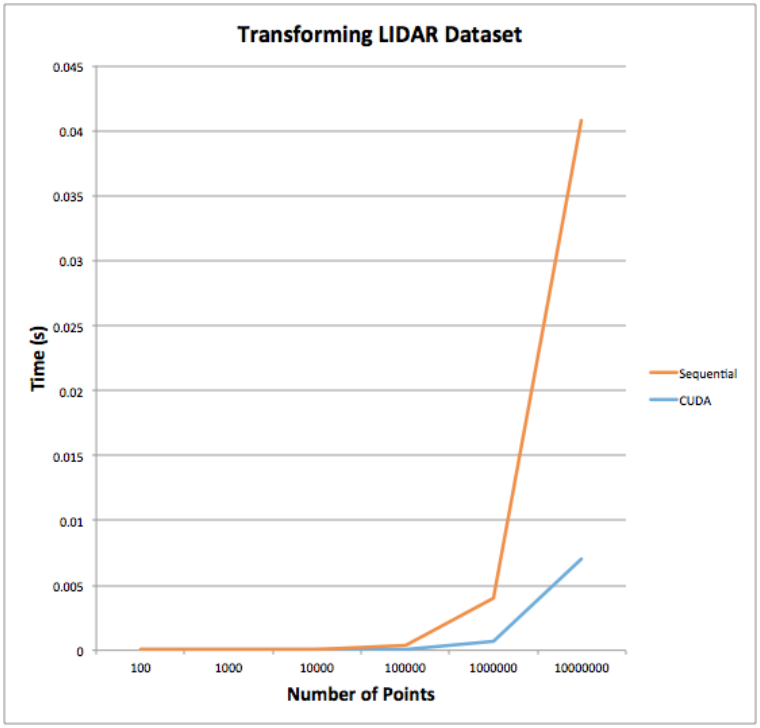

Figure 5: Comparison of our LIDAR transformations as the scan size is increased and across single-threaded sequential and parallel implementations

Figure 4 shows the rate at which the GPU is handling the particle-filter update step as we change the number of particles we are considering. The first result is that there is very little effect of block size on the rate of processing. Another result is that once we hit around 1,000,000 particles, we appear to hit an upper bound on the processing rate. This tells us that we are not making full use of resources until we push to 1,000,000 but this does not necessarily mean that we should scale to this number of particles because then we are increasing our latency. It also doesn't reveal whether our parallel implementation itself is performant. The tradeoff to consider here is latency vs. resource utilization vs. accuracy (dependent on number of particles). This graph was useful in our decisions about particle-filter size.

Figure 5 compares the runtime of a sequential CPU algorithm to apply transformations to a LIDAR scan to the runtime of our parallelized version in CUDA. Note the log scale on the "Number of Points" axis. A sequential version cannot scale well as it grows to be over 4x slower than our parallelized version at 10,000,000 points

Testing these properties of our algorithm with respect to workload size was important because it helped us understand the size of a map that would be viable to process and update efficiently. The problem size is particularly important because our size is changing as the algorithm continues to run since a larger map is being built.

We definitely made a sound choice of machine target because we wanted to be realistic in what an autonomous vehicle would be able to take advantage of. We didn't want to use many nodes because it is not realistic for a vehicle to have that. Given that we have a compute intensive and data-parallel algorithm, we believe we are positioned to make good use of a GPU.

CONCLUSION:

In the end, our work is actually incomplete. We were able to analyze and verify correctness of multiple stages in our particle-based SLAM pipeline, but our ability to apply 15-418 knowledge to our SLAM pipeline and actually demonstrate working SLAM was limited by our failure to properly scope the project, our decision to process the KITTI dataset and develop our SLAM implementation entirely from scratch, and the large amount of setup required to integrate the pipeline tightly. We think there is still lots of room for performance optimization and analysis in the space of SLAM and LIDAR data processing.

REFERENCES:

1. “CUDA Accelerated Robot Localization and Mapping.” Haiyang Zhang, Fred Martin. University of Massachusetts Lowell.

LIST OF WORK BY EACH STUDENT:

Equal work done by group members.