Parallel Nearest Neighbor Search for Velodyne LiDAR Point Clouds

Summary

We implemented a parallel nearest neighbor search that is optimized for point clouds returned by Velodyne LiDAR sensor, which are universally used on autonomous vehicles for sensing, localization and mapping. Nearest neighbor search is a fundamental step in iterative closest point (ICP) algorithms used to perform mapping. Our algorithm is achieves a 2x speed up over an optimized sequential kdtree-based nearest neighbors library implementation through GPU parallelism and by taking advantage of the 3D structure of Velodyne LiDAR point clouds.

Background

Lidar sensors provide 3D information about the environment. The information they return is in the form of point clouds: sets of 3D points, each with an (x,y,z) coordinate. Lidar sensors shoot laser beams into the environment. The laser beams hit surfaces and return to the sensor. By measuring the time of flight for each beam, the sensor is able to know how far surfaces are in particular directions. Using many laser beams, a lidar sensor generates a point cloud that encodes the 3D structure of the environment.

Lidar sensors are almost universally used on autonomous vehicle. In particular, the Velodyne lidar sensor is an extremely common sensor that is mounted on top of many self-driving cars. The point clouds it returns are useful for sensing, localization and mapping. Velodyne point clouds look like this:

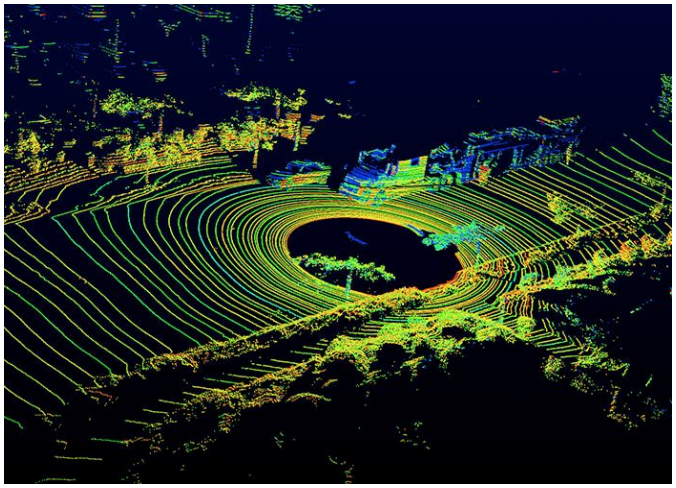


Figure 1

A common algorithm used for mapping is point cloud matching, where the difference between two point clouds is minimized. By overlapping point clouds returned sequentially from the lidar, a map is built, and the vehicle can localize itself in this map simultaneously.

A key step in point cloud matching is nearest neighbor search. In the matching of two point clouds, each point in one point cloud looks for its nearest neighbor (by Euclidean distance) in the other point cloud. General nearest neighbor search is often implemented with a k-d tree, which gives an average O(log n) search time.

Data structures

3D Point

```
typedef struct {
  float x;
  float x;
  float x;
} Point;
```

Point cloud

```
typedef std::vector<Point> PointCloud;
```

Problem statement

Input: PointCloud p0 with N0 points, PointCloud p1 with N1 points
Output: int nearestNeighbor[N1] such that nearestNeighbor[i] is the index of p1[i]'s nearest neighbor in p0, i.e. p0[nearestNeighbor[i]] is the closest point in p0 to p1[i]

Parallelizing this algorithm is not trivial due to hardware and memory constraints. Point clouds have more than 120,000 points on average, so a naive pairwise brute force distance computation would need enough memory for $120{,}000^2$ = 14 billion entries, or 14 billion threads in total. Therefore we need to be smart about data reuse in order to reduce the memory footprint of the algorithm. We also try to use shared memory where possible to reduce the latency of memory access.

Approach

Establishing a baseline

We are trying to outperform a sequential kd-tree-based nearest neighbor search implementation that is fast and optimized (the Fast Library for Approximate Nearest Neighbors or FLANN), so the first step is to figure out how long that is taking.

We took two timings: the time it took to insert the 120,000 points in the k-d tree (construction), and the time it took to query the k-d tree for the nearest neighbor for each of 120,000 points (querying).

Trial	Construction (s)	Querying (s)	Total (s)
1	0.0292	0.199	0.228
2	0.0331	0.204	0.237
3	0.0263	0.165	0.192
4	0.0287	0.159	0.188
5	0.0540	0.345	0.399
6	0.0269	0.175	0.202
7	0.0229	0.196	0.219
8	0.0266	0.178	0.204
9	0.0263	0.178	0.205
10	0.0351	0.185	0.220
Average	**0.0309**	**0.198**	**0.229**

The overall timing we are looking beat for 120,000 queries into a cloud of 120,000 points is 0.229s.

Naive sequential implementation (brute-force)

```
Point* cloud0; // N0 points
Point* cloud1; // N1 points

int nearestNeighbor[N1]; // nearestNeighbor[i]:index of the nearest neighbor of
cloud1[i] in cloud0
for (int i = 0; i < N1; i++) {
  int minIndex;
  float minDist = FLT_MAX;
  for (int j = 0; j < N0; j++) {
    float distance = computeDistance(cloud0[i], cloud1[j]);
    if (distance < minDist) {
      minDist = distance;
      minIndex = j;
    }
  }
  nearestNeighbor[i] = minIndex;
}
```

Deciding to use CUDA

The problem finding the nearest neighbor between a point from cloud1 and all points of cloud0 involves calculating the distance between a point and all others. This will invoke the same set of instructions on different pair of points, making it amenable to parallelism with SIMD. This, combined with the facts that cloud0 contains thousands of points and that the GPU has hundreds of vector lanes for SIMD parallelism, makes this a good problem to be carried out with CUDA.

Parallelizing over points

Our first pass was a CUDA implementation that parallelized over the pixels in the reference point cloud and calculated the nearest neighbor for each point in the target point cloud sequentially. We mapped each point in the reference point cloud to a CUDA thread. For each point *a* in the query cloud, we launched a CUDA kernel. Every thread in the kernel calculated the Euclidian between the coordinates of the point from the reference cloud it was assigned to, and the *a.* When this was done, we reduced across the threads within a threadblock to find the index of the point in cloud 0 that was nearest to *a.* We then performed another reduction across the values returned from all threadblocks to find the minimum. Pseudocode for what we did is as follows.

```
int findNearestNeighbor(Point* cloud0, Point X, int N) {
  __shared__ float distances[N];

  distances[threadID] = computeDistance(X, cloud0[threadID]);
  __syncthreads();
  // Reduce to find smallest distance
  return minReduce(distances, N);
}

Point* cloud0; // N0 points
Point* cloud1; // N1 points

int nearestNeighbor[N1];
for (int i = 0; i < N1, i++) {
  nearestNeighbor[i] = findNearestNeighbor<<<N0>>>(cloud0, cloud1[i], N0);
}
```

In order to find out if we are maximizing the parallelism available with CUDA, we varied the number of points in the reference point cloud and obtained the following graph.

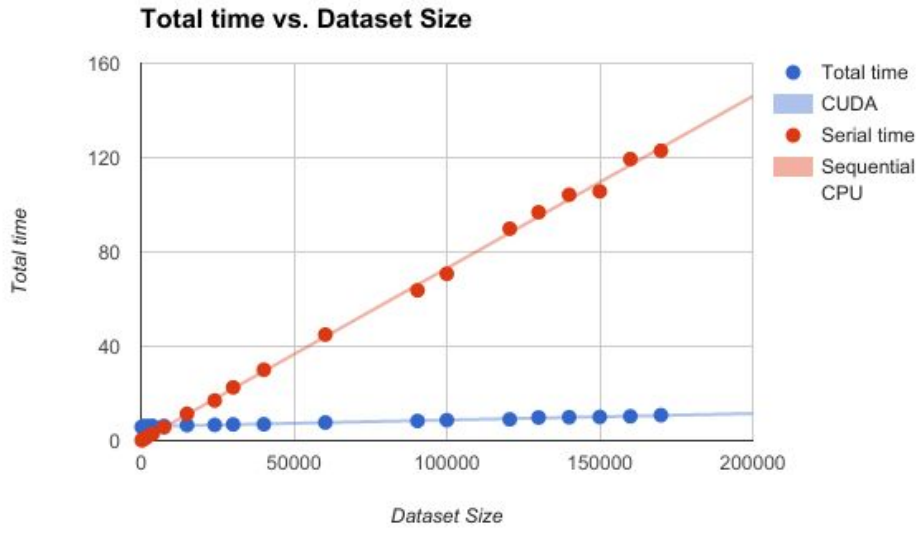


Figure 1

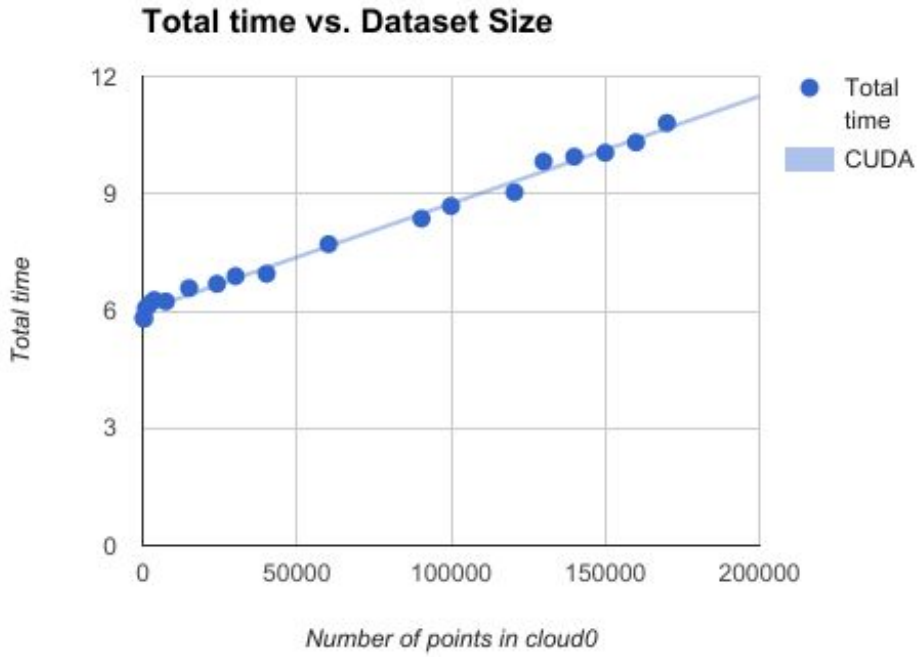


Figure 2. Zoomed in version of figure 1

As we can see from Figures 1 and 2 above, the CUDA times fluctuated somewhat (and were actually slower than the sequential implementation) when the reference dataset contained around 2,000 points.

Figure 2 also indicated that the overheads of the CUDA kernel calls dominated the overall time taken, especially when there were fewer points.

When there was a larger number of points in cloud 0, there was a somewhat linear increase in time as the number of points in the dataset increased. This indicated that we were achieving the maximum parallelism capabilities of the machine given the constraints of our algorithm and dataset.

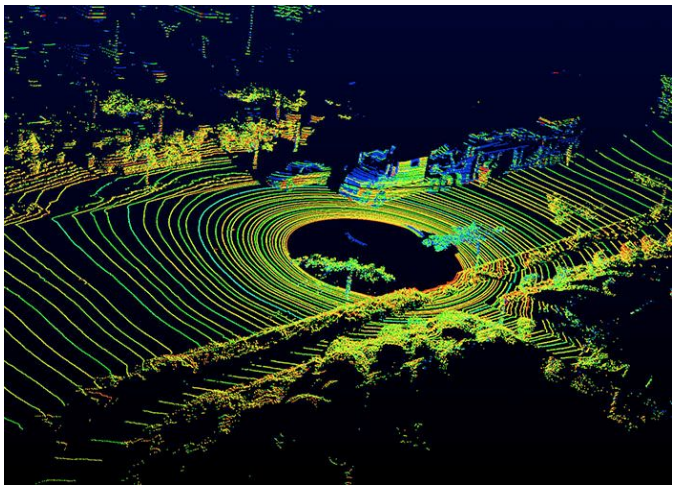
Despite the massive parallelism that CUDA brought, finding the nearest neighbors took 9.4s, 40.3x slower than the sequential PCL kd-tree implementation with FLANN for the same dataset. This led us to think of other implementations that reduced the number of times we launched a CUDA kernel.

Parallelizing over points in batches

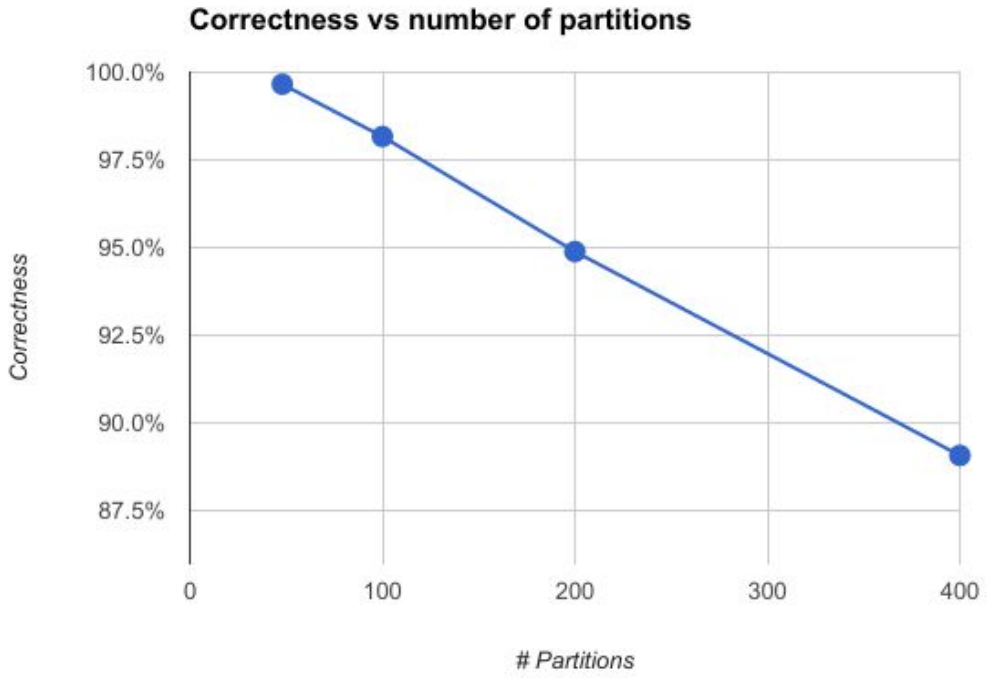
We changed tack and shifted the looping over query points into the kernel itself to minimize the number of kernel calls. The mapping of cloud 0 to threads remain unchanged. However, each threadblock is now computing the nearest neighbors for multiple values before returning. This took 2.0s, 4.3x faster our prior parallel CUDA attempt. This reduction in time took place because we were now reducing the number of CUDA kernel calls by a few orders of magnitude.

Reducing search space

After our first 2 tries were met with limited success, we came to the conclusion that we would have to do some prior processing of the point cloud to reduce the search space to have any hope of beating the optimized FLANN algorithm. The point clouds produced by lidar sensors have a very specific 3-D structure like the one below. This led to the idea of partitioning the reference into bins, and searching only in a particular bin for a point's nearest neighbor.



Reducing the search space for nearest neighbors gives us significant additional speed up. We found that 99% of the time, the nearest neighbor of a point would be within 2.5º of it in the azimuth. Taking advantage of the 3D structure of Velodyne point clouds, we can partition the point clouds by azimuth, and only search for nearest neighbors in the partition with similar azimuth. We sacrifice some correctness in reducing the search space because the correct nearest neighbor might not be in the partition we are searching, but for most applications having some outliers is acceptable to get a speedup.



Depending on the level of correctness needed by an application and whether it is finding nearest neighbors in consecutive or non-consecutive frames, one can choose the maximum amount of speedup that meets the constraints.

The pseudocode for this iteration of our code looked as follows:

```
void findNearestNeighbor(Point* cloud0, Point* cloud1, int N0, int N1, int*
nearestNeighbor) {
  __shared__ float distances[512];

  // Each thread block is responsible for finding the nearest neighbor of one point
in cloud1
  Point X = cloud1[blockID]
  int partitionOffset = computePartitionOffset(X); // index in cloud0 where X's
partition begins
  distances[threadID] = computeDistance(X, cloud0[partitionOffset + threadID]);
```

```
  // Reduce to find smallest distance
  nearestNeighbor[blockID] = minReduce(distances, 512);
}


Point* cloud0; // N0 points
Point* cloud1; // N1 points

partition(cloud0, 300); // partitions cloud0 into 300 partitions

int nearestNeighbor[N1];
findNearestNeighbor<<<N1,512>>>(cloud0, cloud1, N0, N1, nearestNeighbor);
```

We noticed from our partitioning into of the points into 300 segments meant that each segment contained less than 512 points from cloud 0 in each bin. Like Attempts 1 and 2, we assigned each point in cloud 0 to a CUDA thread and computed the distance from a point in cloud 1 to all these points, before doing a reduction to find the minimum. However, since the sample space has been reduced, the computation needed to find the nearest neighbor for any point in cloud 1 has been localized to a single thread block. This meant that there was no need to wait for all threadblocks to return doing a final reduction. This eliminated a large portion of work done from our earlier implementations.

Summary of Results

The timings and speedups that we obtained from the approaches mentioned above are summarized in the following table. Problem sizes were consistent across testing for all algorithms. Performance was measured according to the speedup w.r.t. the FLANN implementation.

Algorithm	**Construction Time (s)**	**Query Time (s)**	**Total Time (s)**	**Speedup**
FLANN	0.0309	0.198	0.229	1.0
Sequential Algorithm	-	62.305	62.305	0.00368
Parallelizing over points	0.296	9.108	9.384	0.0244
Parallelizing over points with fewer CUDA kernel launches	0.245	1.706	1.95	0.117
Parallelizing over reduced search space	0.246	0.117	0.363	0.631

* The timings above were the averages of 10 trials taken on the same machine.

One interesting thing we noticed was that although our final implementation did not beat the total time taken by FLANN, its query time was actually 1.69x faster than FLANN's, and our overall speedup was limited due to the high overhead of constructing our data structures.

To go deeper into understanding why the time required was so much higher than FLANN's, we broke down the construction time for our last approach into 2 sections: (1) Calculating which partition each point belong to and (2) Copying this data (1-dimensional array) into CUDA memory. We found that the latter took up 93% of the 0.246 second construction cost. This problem could potentially be eliminated by the use of Unified Memory, which is something that we regretfully did not have the time to explore.

We believe our choice of target machine was sound, but it will be interesting to see how runtimes will differ with ISPC and whether the high construction overheads could be avoided here.

References

Point Clouds Library

Point Cloud Datasets

K-d trees

Nvidia Blog on Unified Memory

List of work done by each student

Equal work was performed by each student