Parallelizing Perception Algorithms for Autonomous Racing Vehicles

Alexander Blasberg, Alice Tran

12/15/2024

Website Link

https://aqattran.github.io/liDar-CUDA-Parallelization/final/

Repository

https://github.com/carnegiemellonracing/PerceptionsLibrary24a/tree/finalproj_418_ablas

Summary

This project focuses on parallelizing the filtering, clustering, and cone coloring algorithms that are part of the Carnegie Mellon Racing (CMR) autonomous vehicle perception pipeline. Currently, the autonomous pipeline takes data points from the LiDAR sensor creating a point cloud for each information frame. Using CUDA, we demonstrate how one can parallelize and improve the process of filtering out the ground, clustering the points based on cone objects, and then coloring the cones into left and right compared to the original python pipeline.

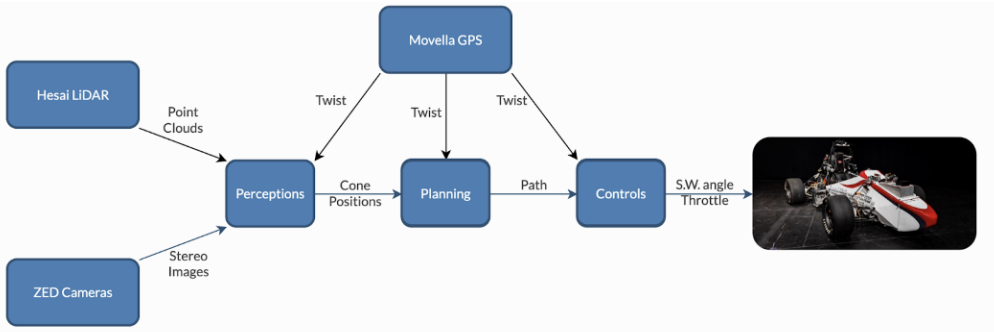


Figure 1: Autonomous Racing Pipeline

Background

1 Autonomous Racing Pipeline

Carnegie Mellon Racing (CMR) is a student organization that builds and designs autonomous vehicles to compete in the FSAE Driverless competition. The car uses various sensors to interpret the surrounding environment and the cones the car must drive between.

The autonomous vehicle relies on three distinct codebases to accurately distinguish between the cones on the track. The perceptions codebase is focused on interpreting the information delivered from sensors such as the Hesai LiDAR and ZED2 cameras in order to map out an accurate view of the course track and the placement of the cones. From the information of the cone layout from perceptions, the car can then begin planning its route between the left and right cones, either using the midline, or the fastest possible route, known as the raceline. Given this path information, the car then relies on the controls algorithms to determine the optimal positioning for the car and the changes necessary to the car's steering wheel angle, throttle, and brakes.

For our project, we chose to parallelize CMR's entire perceptions pipeline. This codebase benefitted the most from parallelizing due to its ingestion of tens of thousands of data points per frame. Specifically, the point clouds from the LiDAR sensor meant that a CUDA implementation would be a significant improvement compared to the current Python data frames approach to parsing information. Additionally, while some of the other pipelines' code-

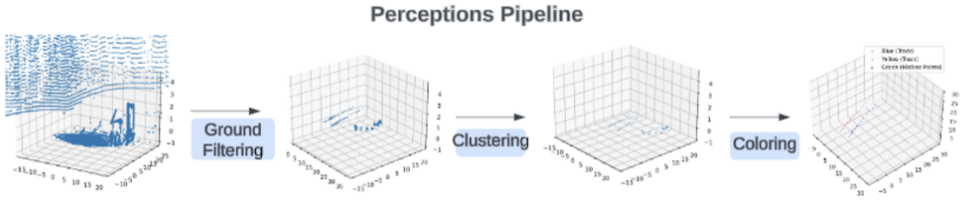


Figure 2: Perception Pipeline With Points

bases were written in C, the perception pipeline is still primarily written in Python. This contributed to its high latency, where the Python perceptions pipeline took upwards of 250ms on some frames. With our CUDA implementation, the entire pipeline runs in under 15ms. Our parallelization also contributes to the long term transition of the codebase to C.

1.1 Ground Filtering

Light Detection and Ranging (LiDAR) systems produce point clouds by returning the travel time between when the signal was emitted from the sensor and when it returns. Therefore, it is one of the standard sensors used for surveying topographies and creating digital terrain models (Yilmaz, 2021).

The process of ground filtering entails identifying and extracting the ground from the point cloud. By establishing the position of the ground in relation to the vehicle, we are able to more accurately understand the surrounding objects such as the cones. While there are multiple methods of ground filtering, the current algorithm does an alternate method of a common technique known as progressive morphological 2D (PM2D).

Here, the current algorithm chose to separate each of the points in the point cloud into individual buckets. In each individual bucket, it applies the PM2D method where the three minimum points in the bucket are identified as the starting point for a plane fit. Given this minimum ground point, it identifies the distance from all other points in the cell to the plane. If a point's distance from the ground plane exceeds the height threshold then it is classified as a non-ground point.

In CMR's current representation they iterate through the bins which are structured in the NumPy array. This algorithm lends well to parallelization

as we can split the computation geospatially and compute each segment concurrently. By executing the algorithm in parallel, we stand to see a large speedup, cutting down on the perceptions pipeline latency greatly.

1.2 Clustering

After identifying the ground in our point cloud, we must identify the objects within the car's reference view. The track is set up such that we have yellow cones on the left and blue cones on the right, denoting the track bounds.

Currently, CMR uses Density-Based Spatial Clustering of Applications with Noise (DBSCAN). Using this model they are able to aggregate the remaining points in the point clouds into clusters based on their similarities. They then compute the centroid of each cluster to give an approximation of the center of that cone.

For our parallel implementation there are two potential areas for improvement. We can apply DBSCAN on each individual point to calculate all individual clusters in parallel. Additionally, instead of sequentially iterating through the clusters to identify the centers we can then divide the work to calculate each separately.

1.3 Cone Coloring

After clustering, we have a list containing all of the cones within the autonomous vehicle's field of range. CMR's current algorithm approximates a starting yellow and blue cone by identifying the closest two cones within the car's field of vision that are left and right respectively of the y axis. From each starting color cone, they identify the next closest cone that is further from the origin and label this cone as the same color as its predecessor. They continue this process until they cannot find another cone that is within the current cone's range.

This algorithm is sequential given that it is dependent on the current yellow or blue cone in order to identify the next cone. We hope to explore various methods of partitioning the method in order to parallelize our information and reduce the runtime of the coloring algorithm.

2 Approach

For the parallelization of the following algorithms, we chose to implement them in C++20, NVIDIA's CUDA 12.6, with the help of some Thrust. In order to convert test cases files from npz format to csv along with benchmarking times, we wrote utility code in Python. Additionally, once the cones within the frames were identified after coloring the information was outputted into a csv file and then visualized using another Python script. We then ran our test cases on GHC machines.

2.1 CUDA Ground Filtering

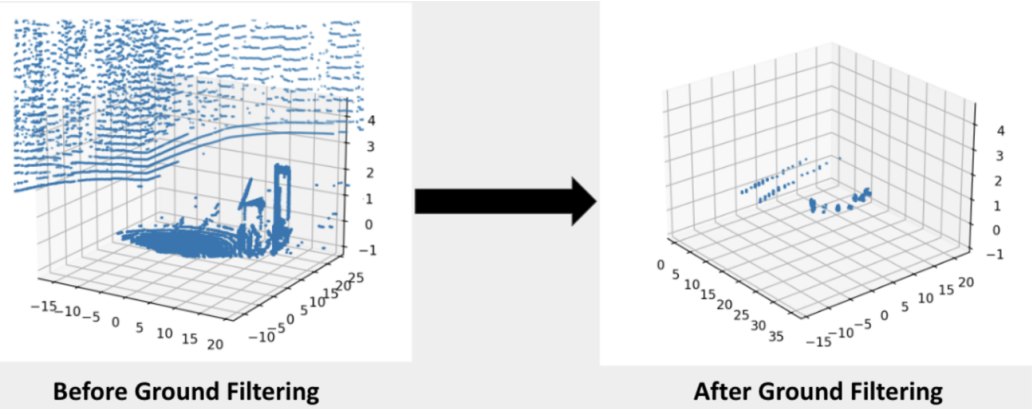


Figure 3: Point Cloud After Filtering

For ground filtering, we began with the originally sequential Python algorithm that was written in CMR's perception database (Carnegie Mellon Racing). The original sequential algorithm chooses to divide up the point cloud into multiple cells before performing various operations on each cell sequentially to determine the ground.

The points in the point cloud are given as (x,y,z) coordinates in three dimensional and therefore we chose to first flatten the points on the z plane. We do this by using thrust to calculate angle between each of the points on the x-y plane and then finding the minimum angle between the points and the maximum angle between the points. Points can then be partitioned into a k number of horizontal bands (called segments) determined by the difference

between the minimum and maximum angle divided by a given alpha (size of the segment).

We then launch a kernel in order to allocate each of our points into a cell in a 2D grid defined in polar coordinates (radius, angle). The angle determines which segment the point is placed and the radius indicates which bin within the segment. Therefore, when we segment our work we end up assigning a thread to each point in our point cloud.

2.1.1 Process and Iterations

Once we had allocated each point into its respective segment and bin we then iterated through various methods of ground filtering. We initially tried a common ground filtering technique known as maximum local slope (MLS) (Yilmaz). In MLS, we relied on the theory that points located on the ground are closer together and therefore have a smaller slope between them than the slope between a point on the ground and one above it. We chose to implement this by launching a kernel with *num_segment* of blocks and *bin_num* of threads per blocks. Therefore within the device code we could then identify the specific block we were in through the block.id and the specific bin through the use of the thread.id.

Each thread therefore corresponded to a unique bin which limited the communication overhead between individual threads. Within each bin, we then could identify the lowest point and designate it as a guaranteed point on the ground.

While the MLS provided passable accuracy, its main disadvantage was speed. This was due to the fact that in our MLS implementation we compared every point to every other point to determine the slope between them. While this provided us with greater accuracy this resulted in a slower run time.

Typical ground filtering techniques are meant to account for unpredictable and uneven terrain such as hills, valleys, shrubs, and foliage. However, given that our dataset came from a comparably flat race course, we hypothesized that a precise ground filtering algorithm for predicting uneven ground would be unnecessary for our data. Therefore we pivoted to a strategy where we calculated a plane fit within each of our cells in order to improve the speed and reduce communication between corresponding cells.

In this iteration, we retained our initial strategy of allocating the points to a specific bin in parallel through the launch of a kernel. We then chose to

partition the work such that each thread represented the work of one cell by launching a kernel of segment number of blocks and bin number of threads per block. However, this time instead of calculating only the minimum point we found the three lowest points in the cell in order to do a plane fit. By plane fitting each individual cell in parallel, we hoped to approximate where the ground was located in reference to the car. While this method further reduced communication, this resulted in a decrease in accuracy. Therefore in our final strategy we sought to strike the balance for finding an accurate enough model for our use case while also limiting communication overhead.

2.1.2 Final Solution

In our final solution, we wanted to reduce the frequency of synchronization and communication of our threads while also predicting the ground with accuracy comparable to the sequential algorithm.

In order to retain the accuracy of the ground filtering algorithm, we chose to use a similar calculation approach as the sequential algorithm. Additionally, we used the same approach of launching a kernel as our previous iterations in order to allocate the points into their designated segments.

In our first kernel, we distribute work such that there is one thread per point. With this distribution each thread only calculates which segment and bin its respective point belongs to. We also aimed to reduce conflicting shared memory in order to avoid synchronization across our threads. We did this by choosing to store the information for a point's segment and bin allocation in an array where each point would only modify its respective information. Therefore only one thread would access a specific memory location which results in no conflicting memory accesses.

Algorithm 1: CUDA Shared Memory Setup

```
bin[point_index]
segment[point_index]
```

Once we have all of the points segmented in their respective bins, we can then go through each bin and identify the lowest point in each bin. We do this by launching another kernel. Here we now associate one thread per bin in order to search through all the points in a similar geographic location. If a bin thread finds a point within its region it then compares the point to its current minimum.

This allows all of our bins to find their respective minimum point in parallel. Additionally, each thread only reads information from the bin[point_index] array but does not write to it. This prevents the need for synchronization between the threads. Instead, once a bin finds the minimum point it stores this information in a separate shared memory where each bin only writes to its allocated space.

Given this information, we then launch a kernel with num. of segments blocks in order to find a line for each segment. In this kernel, we take the block.id to determine which segment we are accessing. Given that we have already calculated the minimum point in each of the bins we can then easily go through and line fit through these lowest points in order to approximate the ground height within the segment. This allows us to calculate each segment separately instead of iterating through all points to create one collective ground height.

Once we established a line approximating the ground, the last thing we did is once again reiterate through all of our points and filter out unnecessary ones. In this newly launched kernel, we allocate one thread per point. The thread is able to access the information about the ground line for the point's corresponding segment. If the point is above the "ground line" by more than variable meters, further than our look ahead distance of 40 m, or above our maximum cutoff height, then we denote the point as an outlier and eliminate it.

Our final solution provided us with the best accuracy and speed compared to our previous iterations. We chose to reduce the communication between threads by ensuring that our work was split geographically across our threads and that each thread only did work on a specific region. Additionally, we ensured that threads did not have to modify any shared information that would be modified by another thread. Instead once we had found some necessary information, we would write into the shared information and then exit from the kernel with this information. Once we had exited from the kernel that found the necessary information we would only read from this shared memory in any operations that came afterwards. This meant that we chose to synchronize our information only after each kernel call and we had to make our code more modular to compensate.

2.2 CUDA DBSCAN

In the original sequential algorithm (Carnegie Mellon Racing), DBSCAN was implemented using sklearn. However, a typical DBSCAN algorithm can take up to O(n^2) time if each point has to check every other point in order to determine if it is close enough to be part of the same cluster. This would be incredibly slow with the thousands of non-ground points returned from the ground filtering step.

2.2.1 Process and Iterations

Since we wanted to avoid going through the multiple depths of relations between the nodes, we chose to implement our version in CUDA to explore how we could shorten the path from a node to the main node representing the parent of the cluster. We took inspiration from union finding. Here, if we designated each point originally as the center point of a cluster but then went through and started grouping together points that were in close proximity to one another (union) and made sure all of these points pointed to the root point (path compression) then we would have a much more efficient clustering algorithm.

Algorithm 2: FindParent Function

Function `findParent(`*id*`):`
  while *parent*(*id*) $\neq$ *id* **do**
    parent(*id*) $\leftarrow$ parent(parent(*id*));
    id $\leftarrow$ parent(*id*);
  end
  return *id*;

In path compression, we choose to point all the points from its direct parent to the overall parent of the group. In clustering, we are thus able to more quickly find the parent of a neighboring group. Therefore instead of doing DFS from a neighboring point in order to find all the closest points to join with, a point can simply join another group's parent and therefore join a group of points already without searching for each close one individually.

If a point is part of two separate clusters, we can then combine them into a larger cluster by doing a union by size. By also calling the findParent option on each of the nodes we want to combine together we also perform

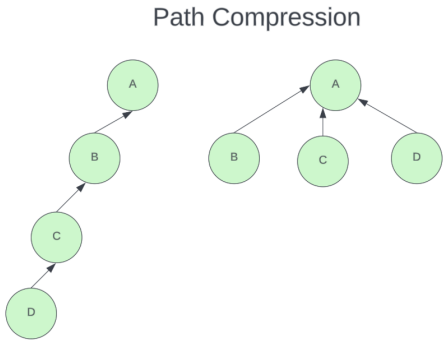


Figure 4: Path Compression to Root A

path compression on longer paths. Therefore this method allows us to quickly find the root of the group and combine them together into clusters.

Algorithm 3: Union Function

Function `union`(x, y)**:**
- **Data:** Two elements x and y
- **Result:** Union of the sets containing x and y
- $parent_x \leftarrow \text{findParent}(x)$;
- $parent_y \leftarrow \text{findParent}(y)$;
- **if** $size(parent_x) > size(parent_y)$ **then**
  - $\text{parent}(parent_y) \leftarrow parent_x$;
- **end**
- **else**
  - $\text{parent}(parent_x) \leftarrow parent_y$;
- **end**

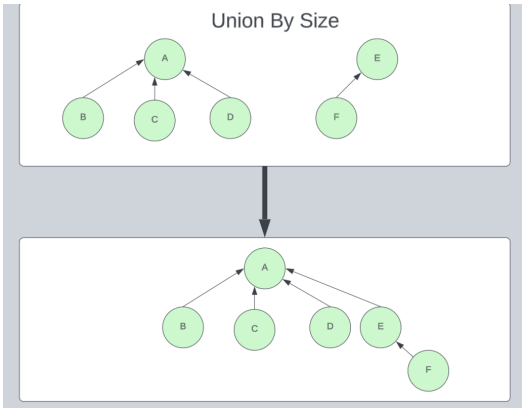


Figure 5: Union By Size

2.2.2 Final Solution

Algorithm 4: CUDA DBSCAN Algorithm

1. Initialize Clusters:
- Launch Kernel where each thread is a point that establishes its own cluster;

2. Find and Union Clusters:
- Launch Kernel where each thread is a point and run union find algorithm;

3. Flatten Clusters:
- Launch Kernel that runs path compression on all the points to ensure that all points point to their main root;

4. Compute Centroids:
- Launch Kernel where each thread adds to a point's root centroid to get approximate shape of cone;

In order to parallelize DBSCAN we first decided to launch a kernel to initialize all of our separate clusters. Similar to the union find algorithm we begin with each point as its own individual group. We do this by assigning each thread to a point and then initializing each point as its own parent.

We then launch a kernel where again each thread is a point. Here each point must search through all the points in order to determine if a point is close enough. If it is close enough then we union this point to the other point with our algorithm described above. We continue with this until we have exhausted all of the points. Since every point is searching for its neighbors at the same time we must synchronize at certain points of this process.

For example, when we are comparing the size of two clusters and which cluster to merge into we must use atomic functions. This is to prevent a conflict when we are determining which cluster is the parent. Otherwise, our threads mainly do independent work despite searching through nearby points who are associated with their own threads. The reason for this is that each thread only modifies its own parent. Although the array of parents is shared, each point only modifies its own parent, so we are able to run path compression without any conflicts between the threads.

While this algorithm ended up working relatively well for our use cases there are some noticeable tradeoffs. In order to minimize communication between the threads we had to use a lot of shared memory, with each point

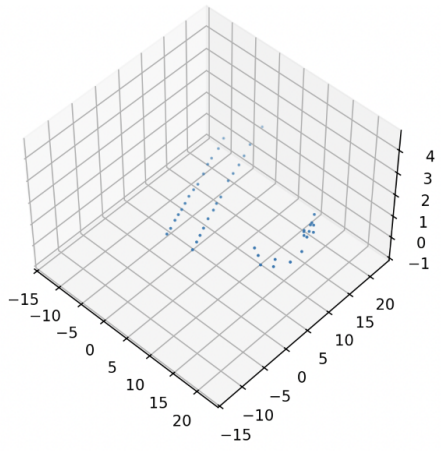


Figure 6: points after clustering

having its own index into the parent array. This would not be scalable if the amount of points exceeded the amount of shared memory. Additionally, each point must search through all other points in order to observe which ones are close enough to combine with. This would not be optimal if the points are so spread out that there are no clusters, the worst case scenario. Thankfully, we know that our algorithm will be used to identify the cones on the track, so there will be clusters at all times.

After we have combined all of our points into clusters we must then flatten our clusters and then create a definitive list of all the clusters. In order to do this, we go through and find the parent of all of our points. We then add in the x,y,z values of our point to the parent coordinates. Therefore at the end we have a centroid that approximately represents where a cone is in relation to our car. When we are adding the point's values to our parent node we must add the information atomically in order to ensure we are accurately maintaining the dimensions of the centroid.

3 Coloring

3.0.1 Process and Iterations

The parallelization of coloring was our most difficult task. The original algorithm (Carnegie Mellon Racing) was inherently sequential by examining a cone and then attempting to find its nearest neighbors. This implementation was extremely slow as it required dependence from one cone to the next, and relied on ICP to check against previous frames. We initially thought that we could partition the cones and attempt to find both the left and right cones at the same time in parallel. However, in order to do this we would have to make the assumption that the cones were evenly spaced from one another.

We first attempted to color the cones based on intensity. Given the cones were two separate colors, they would return different intensities of light to the LiDAR, as they would absorb different amounts. Therefore, we implemented an algorithm that used intensity as a function of distance to classify cones. Unfortunately, this algorithm was not robust enough, as the differences in intensity between cones was often spotty.

On the track, cones are irregularly spaced depending on if there is a curve or a straight line. Therefore instead of attempting to brute force a midline we experimented with segmenting the track into groups. For example, we experimented with breaking up the track based on a Voronoi diagram. By segmenting the track in this way, we could make a Voronoi diagram representing the bounds of the area closest to each cone and attempt to follow the longest path through to establish a midline. However, the downside of this was that it was too complicated for the necessary data. From this idea, we established a base approach for how we could cluster our cones and also implement a midline approach in sections of the track.

3.0.2 Final Solution

For our final solution, we came up with two algorithms. Firstly, we adapted our clustering algorithm in order to group the cones into a specific color. This worked for larger dense tracks which are often curves and bends in the track. We also discovered a midline approach which worked best on sparse tracks that are often going in a straight line.

Clustered Coloring Algorithm

For our clustering coloring algorithm, we transformed our DBSCAN algorithm in order to also color our cones. We retained our original paralleliza-

tion of DBSCAN but this time ran it with a larger epsilon to identify the clusters instead. Once we have identified the clusters from the car's viewpoint we created the race track from the viewpoint by identifying the closest large cluster to the car.

Once we identified the large closest cluster we broke it down into smaller sub-clusters by rerunning DBSCAN with a smaller epsilon. Afterwards we label the two closest subclusters based on their x coordinates to either be blue or yellow cones. For this algorithm, we parallelized by reusing our old DBSCAN algorithm. Therefore by calling it we essentially are launching 6 kernels in total (three per DBSCAN call) and synchronizing our threads after each call. While this provided us with 250x speedup over the original coloring implementation, we found that we could more efficiently run this algorithm without parallelism given the small use case. We discuss this in further detail in the results section.

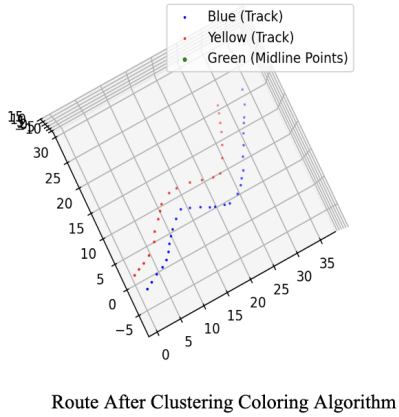


Figure 7: Clustered Coloring

Midline Coloring Algorithm

In our midline algorithm, we started with the hypothesis that if we split a track into sections of points we can then draw a midline in between the two points. Once a midline is discovered, cones on the left and right can be easily separated. The algorithm goes as follows:

As the pseudo algorithm briefly states, we identify the segments of track that are closest to the vehicle and then split them into sections. All of the processing is done sequentially until we compute the midline points, as the

Algorithm 5: DBSCAN-based Clustering Process for Racecar Path

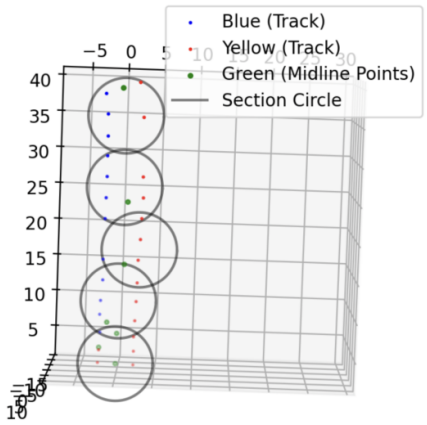
Function `dbscanClusteringProcess(`*points*`):`
- **Data:** A set of points (racecar trajectory)
- **Result:** Classified cone sections based on DBSCAN clustering
- **Step 1: DBSCAN Clustering**;
- **Run DBSCAN:** Apply DBSCAN clustering algorithm to identify clusters;
- **Find Closest Cluster:** Identify the cluster closest to the racecar;
- **Step 2: Process Points**;
- **Extract Points:** Extract points from the closest cluster;
- **Step 3: Determine Reference Points**;
- **Identify Reference Points:** Identify the closest points on the left and right sides from the closest cluster;
- **Step 4: Create a Section of Points**;
- **Divide into Sections:** Based on each cone point, separate them into sections of the track;
- **Step 5: Compute Midline Points**;
- **Launch Kernel:** Compute each section's midline point;
- **Step 6: Classify Cones**;
- **Launch Kernel:** Classify the cone coloring based on midline points and cone positions;

segments need to be found sequentially to stay on track. This is relatively fast, as we have reduced the scope of the problem to less than 50 points in a frame. Therefore, the processing of our information sequentially is relatively low overhead. Instead, we decide to parallelize the process of computing a segment's midline and then classifying a cone, where the most computation occurs.

To calculate each segment's midline, we launch a kernel where one thread is equivalent to one segment. Due to all the preprocessing beforehand, we already have information about which points are contained on each segment. Therefore, each thread can read information from shared memory about the coordinates of the cones in its sections, Afterwards the thread takes the average of all the cones points in order to approximate a midpoint. Each thread writes the information about the segment's midpoint to shared mem-

ory. However, since we allocate a specific space for each thread we avoid the need for synchronization between the threads as we are writing. Since we allocate each segment based on a geographic radius it is possible for one section to take longer to compute compared to others. Therefore, we must synchronize after this in order to ensure that all threads have finished their work and avoid divergence.

Afterwards, we launch another kernel to classify the cones. In this kernel, we allocate one thread to one cone. Since we synchronized before we can guarantee that all the segments' midlines have been calculated. Therefore given a cone, we can then go through each of the midline's segments and find which one is the closest. Once we have found the closest segment, we can then identify the cone's color through a perpendicular comparison to the midline. Since we color each cone through its own thread there is no need for communication between the threads. We only must synchronize afterwards in order to ensure that each cone was colored at the end of the process.



Route After Midline Algorithm

Figure 8: Midline Algorithm

Results

In real time application, the perceptions pipeline takes in point clouds of information from each frame and then establishes a viewpoint from the car's perspective. In order to test our parallel improvements we chose a variety of frames (each with their own respective point cloud) which represented the variety of scenarios that could be encountered by the vehicle. For example, test case 1 and 2 represents a track that bends multiple times within the frame. Therefore, this shows how our algorithm runs on test cases with a higher density of points and clusters.

Additionally, for test cases 3 and 6 these are sparser and straighter tracks within our frame. Here we can observe how much improvement is made for a computationally easier test case for the sequential algorithm. In test case 5, we also include a dropped frame meaning that the information within the point cloud was corrupted in some way. This simulates a possible input that we could receive in real life due to the unpredictability of LiDAR readings to our systems.

Each frame contains $\tilde{6}$5,000 points represented as coordinates on the (x,y,z) axes. Additionally, the points contained the intensity of the LiDAR reading for use in coloring if needed.

In all cases, we compared our parallel implementation run on the GHC machines to the baseline sequential algorithm written in Python on GHC.

3.1 Filtering

For ground filtering, we chose to examine the results of our parallel algorithm in terms of wall clock measurement by timing the start and end of the process for both the sequential and parallel speedup over multiple iterations. From there we also calculated the speedup in comparison to the Python sequential version.

The parallel CUDA ground filtering algorithm performed the best on frames that contained a denser collection of points which were present in test cases 1 and 2. In these frames, the car must approach a series of turns which results in more points (cones) closer together. Due to our segmentation of the points geographically into separate work, we were able to more evenly distribute the work compared to the sequential algorithm. Therefore we saw the greatest speedup of around 4x compared to the sequential algorithm.

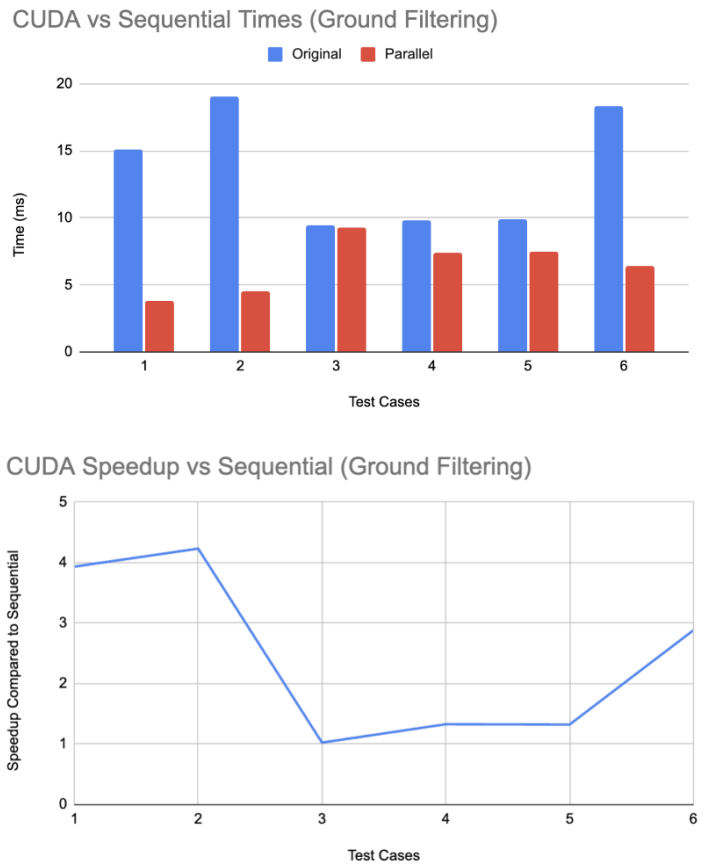


Figure 9: Results for Ground Filter

3.1.1 Limitations

However, when dealing with a sparser and more spread out collection of points for a straight track, the overhead of our increased communication results in reduced speedup. As described in the approach section, after we calculate the minimum point within each cell, we must still fit a line across each of the minimum points in the horizontal bins. This high cost of communication means that computationally easier test cases (such as those with more spread out points) benefit less from this optimized line fit. Additionally, when partitioning the points geographically, we use a static allocation for each of the buckets. This means that for sparse graphs we should be creating larger buckets in order to ensure that each thread has enough work. An extremely fine granularity of bucket size results in high communication cost and explains the lack of speedup for test cases 3,4,5 where we were filtering out tracks with points spread further apart. Therefore in the case of sparse cones, we would have an increased synchronization overhead due to partitioning into fine tasks, without enough work per thread to compensate.

For denser graphs, we deal with the issue where the work is not accurately balanced between each of the tasks. A geographic allocation means that cells may either contain either a densely populated area or one that has comparatively fewer points.This causes thread divergence where threads iterating through a bin with less points finish much faster than those without many points. This explains the reason for imperfect speedup in test cases 1,2 with denser points.

We can also see this issue more clearly in the test case 6. This case almost has a speedup of 3x which is much greater than the other straight track test cases (3, 4, 5), however it does worse than the test cases with curves (1,2). This is because test case 6 is a straight track but has a large cluster off to the side which is a later curved portion of the track. Therefore test case 6 benefits from some speedup compared to the sequential version due to this high density cluster. However, this also causes the issue of even work distribution between the threads. as the bins that contain the straight part of the track have much less points than those that contain the large cluster. Therefore, this explains why case 6 has a reduced speedup compared to cases 1 and 2.

3.2 DBSCAN

Similar to ground filtering, we chose to examine the results of our parallel algorithm in terms of wall clock measurement and also speedup. Compared to the sequential algorithm even in the worst case, we have a speedup of around 5x for tracks that are sparser (have less cones) and are in a straight line. However, this algorithm optimizes best in scenarios with large clusters. This is primarily due to how inefficient the sequential version of DBSCAN is when examining dense clusters and finding their neighbors. With the parallel path compression and union find, we are able to limit the depth of our searches in parallel. Therefore this better benefits dense clusters.

We can see that due to our more equal distribution of work in DBSCAN–where every point in a thread must search for nearby clusters in a union-find—we are not hindered by an unequal distribution of work like in ground filtering. Therefore, here we can truly see the impact of how larger clusters (test cases 1,2,6) result in much greater speedup compared to the straight line test cases.

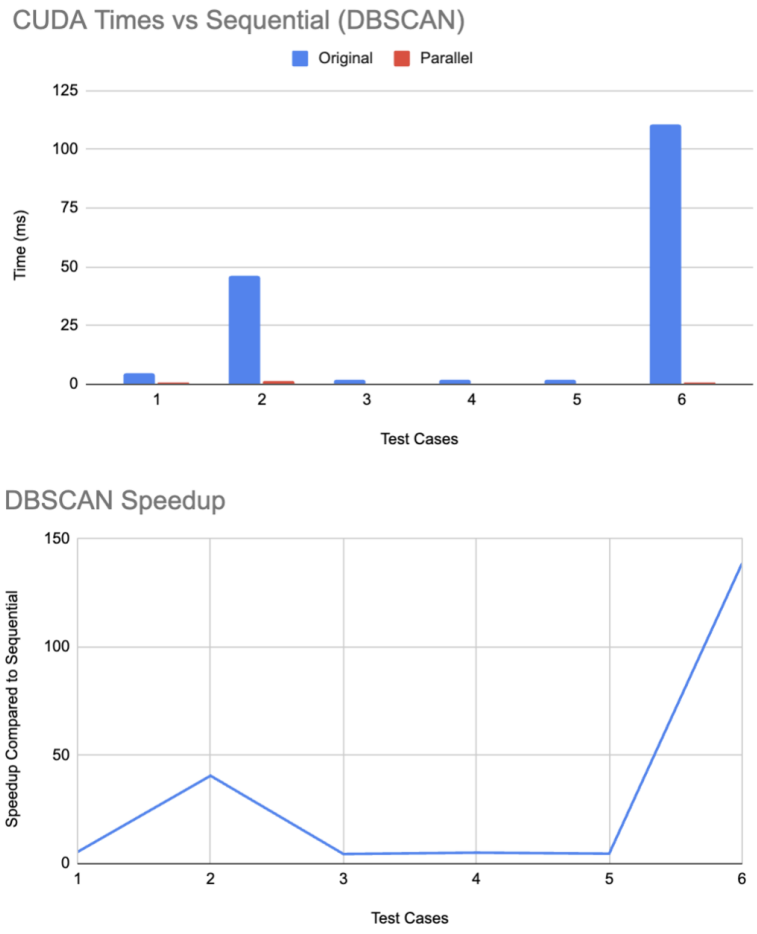


Figure 10: DB Scan Timings

3.2.1 Limitations

However, there are still limitations preventing a perfect speedup, specifically in cases where there are sparser clusters. In our algorithm, each point must check all surrounding points to determine if they should join into a larger cluster. While this is beneficial in cases where we have a large cluster, if there are many small clusters this becomes inefficient as it is unnecessary to check every single point.

Additionally, when we are deciding which cluster to combine we must take an atomic minimum between the two root clusters. With larger clusters, this synchronization overhead is overcome by the efficiency of combining two larger clusters. However, if we are dealing with smaller clusters the synchronization steps become unnecessary overhead as we may be combining very spare groups for little benefit. Therefore this results in a less than ideal speedup in those cases.

3.3 Coloring

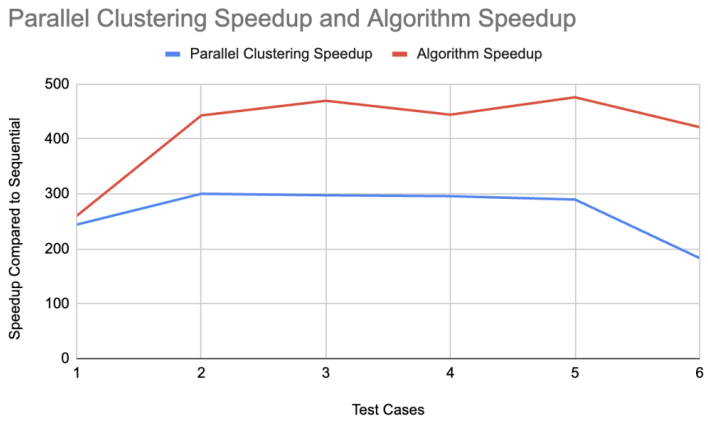


Figure 11: Clustering Algorithm Comparison

Due to the requirement of active GPS data and previous LiDAR frames to test the sequential coloring algorithm, we can only approximate the amount of time the sequential coloring takes, based on amortized analysis of its performance. For relatively simple frames the sequential algorithm runs at around 80ms with computationally harder cases containing more clusters

running up to 200ms. For a baseline comparison, we chose to compare all of our times to an average of 140 ms.

With our two algorithms, all of our test cases received times faster than a millisecond. Therefore we only graphed the speedup for both of the algorithms relative to the average sequential time. Here, we can see that in all cases that clustering was much faster than midline. We can attribute this to the fact that midline algorithms require a much higher sequential overhead when processing the data compared to clustering.

3.3.1 Limitations

One of the reasons for an imperfect speedup for the midline algorithm is due to the static allocation of our cones. When we break up the cones on the track into segments we do this based on their geographic location on the track. However, this results in some segments having many more cones than others. Therefore, we can see in test cases one and two where there is a curve and therefore a denser group of cones we have a slower speedup compared to the test cases with a straight line. This is due to the divergence of our threads which are doing work per segment of the track. Some threads are finishing before other threads however due to the need for synchronization we must wait for all threads to finish.

With our clustering algorithm, we ran into an interesting case where launching the CUDA kernels caused slowdown compared to simply writing the code on CPU.

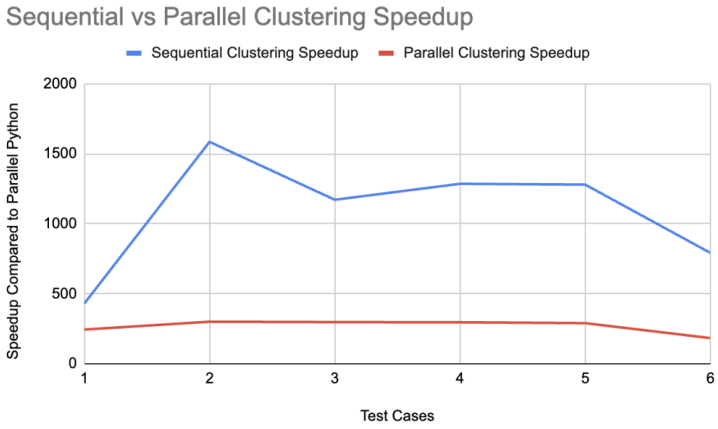


Figure 12: GPU vs CPU Algorithms

In the parallel case, when we launch a CUDA cluster we can see that the speedup is only around 250 times that of sequential compared to the algorithm that was simply written in host code. Once we are coloring the cone points, we are dealing with very few ($\tilde{40}$) points. Therefore an algorithm that goes through and does assignment to these cones could be faster as we are not spending time transferring data to device code to use the GPU and additionally not launching any kernels. When we launch a kernel, we are often assigning the work such that there is one thread per point. However, given how few points this fine granularity means that there is not enough work per thread. Additionally, there are synchronization overheads in order to ensure that all threads have correctly done their work. In use cases such as assigning color to a small amount of cones we overall found that our host code was faster than any parallel code written with the use of CUDA.

Reflection on GPU Usage

For the majority of our project, we wrote device code on the GPU. However, as illustrated above there are certain cases where CPU is superior to GPU. Specifically, when it is a simpler case that can be solved without parallelism. Throughout our project we chose to format our code in segments where we would launch and kill multiple kernels. This caused some unnecessary overhead, however given the high number of points we were working with, the synchronization costs were overcome by the speedup from parallelism. In smaller test cases it would be more beneficial to explore a non GPU/CUDA solution. Additionally, there were cases where dynamic task allocation would have led to better workload balance. Therefore, using OpenMP and coding on CPU could have been another, possibly better, option in that aspect.

Further Improvements

Given more time, we would have liked to explore quite a bit more with optimizing this perceptions pipeline in parallel. Firstly, we would like to look at different algorithms for coloring, and any sort of way to combine our two current algorithms to make a more robust singular algorithm. We would also like to explore the possibility of using cameras in the LiDAR pipeline for coloring, as this would lend itself quite well to parallel work. In places

where there is less overall work, we would also like to explore CPU parallel approaches, such as OpenMP or SIMD operations.

Work Distribution

Alex (60%)	Ground Filtering Optimizations Clustering Optimizations Coloring Optimizations Pipeline for Testing/Benchmarking
Alice (40%)	Website and All Previous Reports Final Report Graphics Poster

Table 1: Task Distribution

Works Cited

1. Carnegie Mellon Racing. "Carnegie Mellon Racing Driverless Documentation." CMR.RED, https://cmr.red/LandingPageDVDocs/build/html/index.html. Accessed 15 December 2024.

2. Carnegie Mellon Racing. Filtering, Clustering, and Coloring Sequential Algorithms. GitHub repository, https://github.com/carnegiemellonracing/PerceptionsLibrary22a.

3. Tran, Alice. *Data for 418 Graphs*. Google Sheets, 2024. Available at: https://docs.google.com/spreadsheets/d/1BdHnE45pxDm2VyiuUUqvSxFxayUeZHUJSKgmFkVVWCw/edit?usp=sharing.

4. Yilmaz, Volkan. "Automated ground filtering of LiDAR and UAS point clouds with metaheuristics." *Optics & Laser Technology*, vol. 138, June 2021, https://www.sciencedirect.com/science/article/pii/S0030399220315231.