Parallelization of RANSAC

E4750_2021Fall_KCAK_report
Kyle Coelho kc3415 Arvind Kanesan Rathna ak4728
Columbia University

Abstract

RANSAC is an iterative algorithm that relies on randomly sampling points to fit models to noisy data [1]. This can be massively sped up to improve run time and computational efficiency so that downstream tasks aren't throttled. We proposed CUDA kernels to speed up several stages of the algorithm from model estimation to error checking to the final inlier checking. One of the challenges was implementing this for large data sizes in the range of millions. We achieved speed ups of ~200x over serial Python implementations on LiDAR test data.

Key Words*: RANSAC, Point Clouds, Euclidean Distance*

1. Overview

1.1 Problem in a Nutshell

This algorithm, RANSAC, relies on random sampling consensus. For a given set of data, it will first randomly sample the minimum number of points needed to fit a model to these points. To fit a plane, this means sampling three points and to fit a simple line, two points. For these sampled points, the model parameters are then calculated, which would be four float values for the plane case. Then, the Euclidean distance is calculated between this plane and every single other point in the dataset. Those that lie within a certain threshold distance are considered inliers and the rest outliers. These have to be counted and if within a certain ratio, then that model can be considered the best model. There is also the more robust version which involves caching the best inliers and then finally using those at the end to fit a final best model [2]. However, that case is not considered in our approach

From the high level overview above, there are clear areas where this can be optimized. While the random sampling nature of this algorithm allows it to perform better than others like least squares, per each iteration of sampling, the entire downstream pipeline is identical. The model estimation can easily be parallelized as it usually involves calculating normal vectors and cross products, which are prime candidates for optimization. The bulk of the computation comes firstly from the distance calculation between the points and model. It is the same 2 calculations performed *n* number of times, where *n* is the number of data points without those chosen for the model; find point of intersection and then Euclidean distance between these two points.

The second area that is also ideal for parallelization is the inlier counting. It also relies on iteratively performing a check on whether the previously calculated distance is within our set threshold and then summing. Lastly, a major point of optimization is also the initial for loop that performs the iterations over the number of models that will be calculated. If this can be parallelized, then it could perform all downstream tasks on the GPU itself instead of having intermediate stages that run on the CPU [4].

1.2 Prior Work

Unfortunately, there is no body of prior work that tries to parallelize RANSAC using CUDA kernels except one Github repo [5]. There are, however, several implementations of this algorithm in popular libraries that have optimized this code in C++. Some of the libraries include OpenCV, SciPy and PCL.

2. Description

Section 2 will provide an overview of the goal and intermediate steps that we took. Section 3 outlines our results in terms of computational speedups achieved and Section 4 shows tests on a LiDAR dataset (KiTTI Benchmark). Lastly, Section 5 offers our thoughts on the results obtained.

2.1. Goal and Objectives

The goal is to achieve a speedup on a real world dataset using our CUDA implementation of RANSAC. We break down our end goal into several intermediate steps to benchmark performance along the way. Each step incrementally builds upon the previous and is timed & tested to match our Python implementation.

2.2. Problem Formulation and Design

As mentioned above, we follow a stage by stage procedure to parallelize the entire RANSAC algorithm.

We divided our algorithm into several stages to pinpoint where the bottlenecks are in our run time performance:

- Develop a serial Python implementation
- Level 1: Parallelize Euclidean Distance Calculations
- Level 2: Parallelize: Model Parameter Calculation
- Level 3: Parallelize the entire for loop that obtains the model and calculates distances for each model
- Level 4: Enable kernels to handle large dataset sizes (> 1024) through careful use of block and grid dimension
- Parallelize model inlier checking

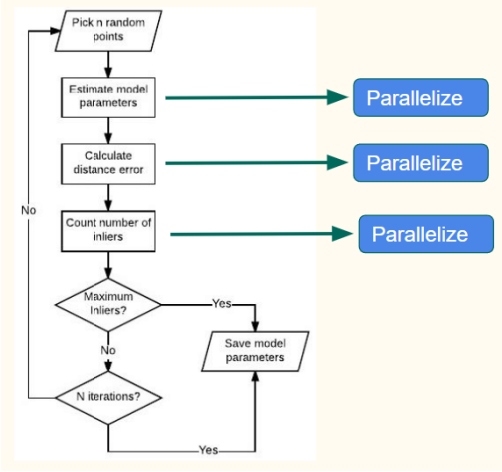

Figure 1: Flowchart for parallelization of our algorithm

The overall flowchart is shown in Figure 1. Each individual level breakdown is detailed further in Section 2.3 along with the pseudocode for this flow.

2.3 System and Software Design

The pseudocode for an example case of 1000 data points that uses 100 models to estimate the best fit plane. This is just a representative example and is generalizable beyond.

1. Count = 0; thresh = X, ratio = Y
2. For iteration in range(100):
    a. Sample 3 points
    b. Calculate a, b, c, d model parameters
    c. For point in (1000):
        i. Calculate intersection point to plane
        ii. Euclidean distance(Intersection, point)
        iii. if distance < thresh:
            1. count += 1
        iv. else:
            1. continue
    d. if count/1000 > ration:
        i. break
    e. else:
        i. continue

The description of the levels of parallelization is as follows:

Level 1: The model points are chosen randomly on CPU and model is estimated also on CPU. These parameters are then allocated on the GPU along with the input array. The calculation for the distances are performed on GPU and an output array is returned containing distances to the model. The inlier checking, ration calculation and model choice are all performed on CPU. From the pseudocode, only step 2.c.ii is performed on the GPU.

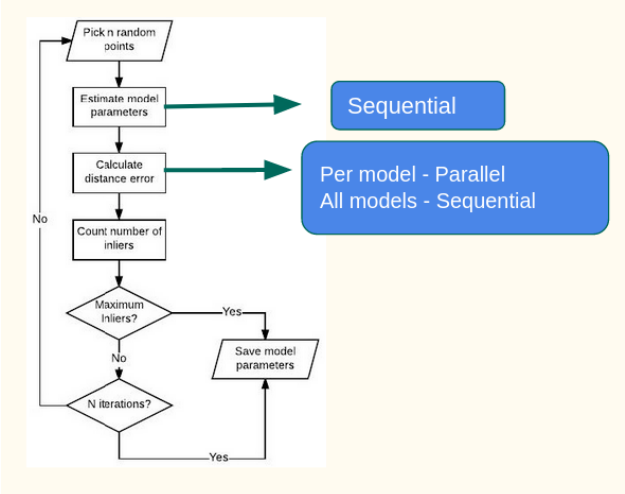

Figure 2: Level 1 Parallelization

Level 2: The model points are chosen randomly on the CPU and then these are allocated directly on the GPU along with the entire data array. The subsequent distance calculation is performed for the model and then the resulting output array is returned back to the host device.

The following steps are carried out on the CPU. From the pseudocode, steps 2.b →2.c.ii are performed on the GPU.

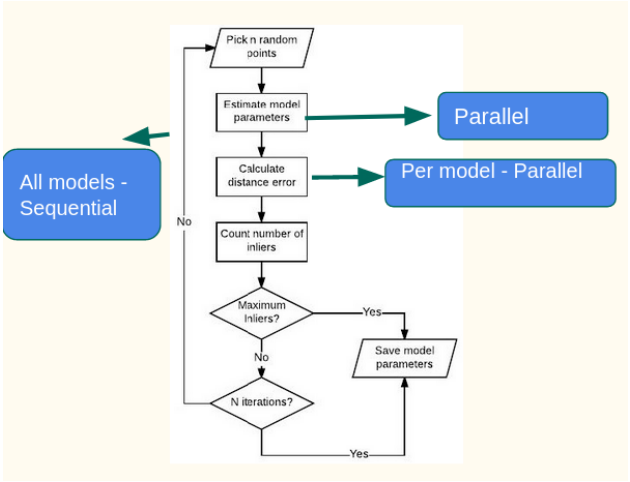

Figure 3: Level 2 Parallelization

Level 3: Now nearly the entire algorithm is ported to the GPU and able to handle a maximum of 1024 points in the dataset. The random selection of points are chosen on the CPU but then the subsequent looping over the models and error calculations are all performed on GPU. The key difference between this and Level 2 is that in Level 2 the CPU serial for loop was used to spawn the n models that were set. But now, this is done in parallel. From the pseudocode, steps 2 →2.c.ii are performed on the GPU excluding the point sampling.

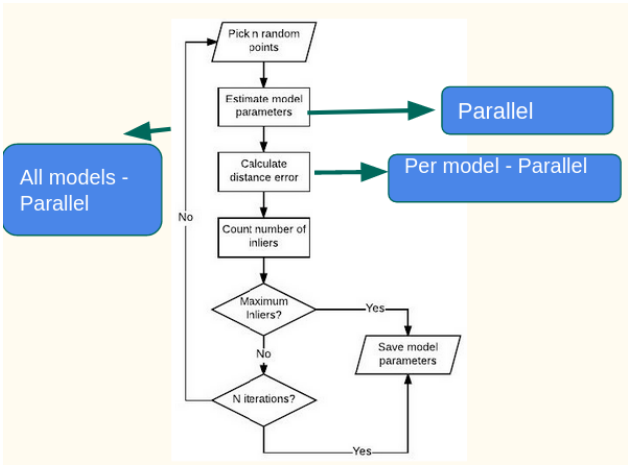

Figure 3: Level 2 Parallelization

Level 4: This remains largely the same however, the improvement is in the handling of data through the careful use of block and grid dimension allocation. All previous steps would not be able to handle dataset sizes larger than 1024 very well. The modification made in this stage was to allocate thread blocks per model in the y axis and thread blocks per 1024 data points in the x direction. So theoretically, if our parameters were set to iterate over 100 models with the dataset having 102,400 data points, then the block dimensions would be (1024, 1, 1) and the grid dimensions would be (1024, 100).

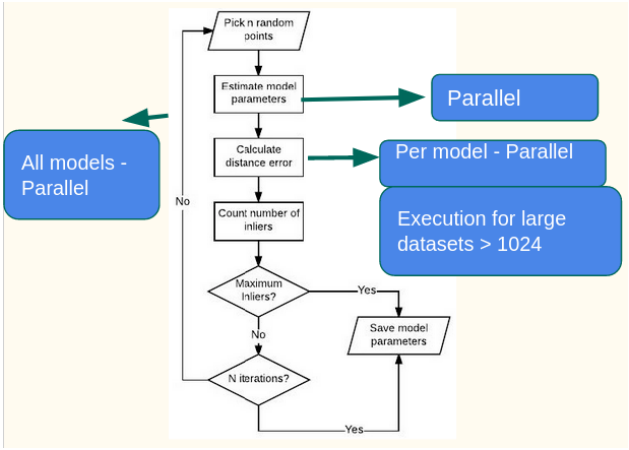

Figure 4: Level 3 Parallelization

Overall, the inlier checking did not necessarily involve any specific computation except verifying a condition so we decided to keep it on the CPU for the testing of all the Levels above. However, our results section will show the impact on performance that it actually has

GithubLink:
https://github.com/eecse4750/e4750-2021Fall-Project-kcak-kc3415-ak4728

3. Results

We were able to achieve results that matched some of our expectations in terms of performance gains. All our results were measured using the following hardware specifications:

Hardware	Model	Memory	Cores
GPU	NVIDIA Tesla T4	16 GB	2560
CPU	Intel Xeon	15 GB	4 vCPU ~ 4 cores

CUDA version 10.2 is used. All CUDA kernels were timed using the CUDA event library and ensured that the start and end times included memory transfers between host and device. Furthermore, NVIDIA's Nsight profiler

was also used to further understand memory and compute utilizations.

We primarily divide our testing into 2 main stress tests: increasing the number of data points and increasing the number of total models used. We tested up to a limit of ~8 million data points and 1000 models. We did not increase beyond these numbers because 8 million is a realistic number of data points up to which one would consider using RANSAC and for the number of models, 1000 is a reasonable number to get a good final estimate.

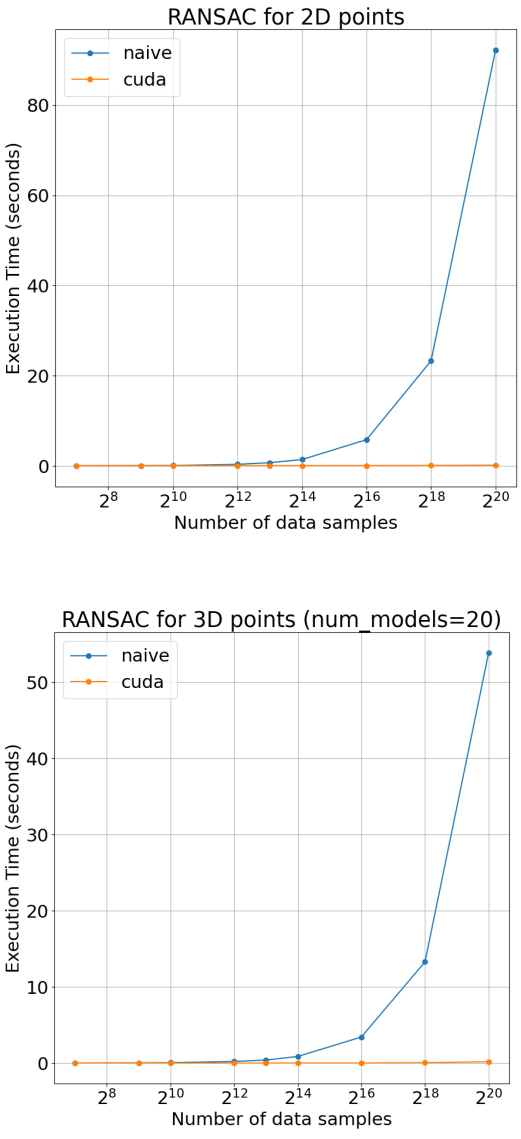

Figure 5: Constant number of models at 20 but increasing data points. x-axis is scaled logarithmically. 2D (up) and 3D (bottom)

Figure 5 shows the outcome when keeping models constant to 20 but increasing the data points to 1 million. Our Python implementation, which we made sure to time at the exact same points as our CUDA implementation, performed much worse with more data, as expected. CUDA time remains nearly constant in comparison. In the maximum case, CUDA time is 1.2s versus 80s in Python, which is a 67x speedup.

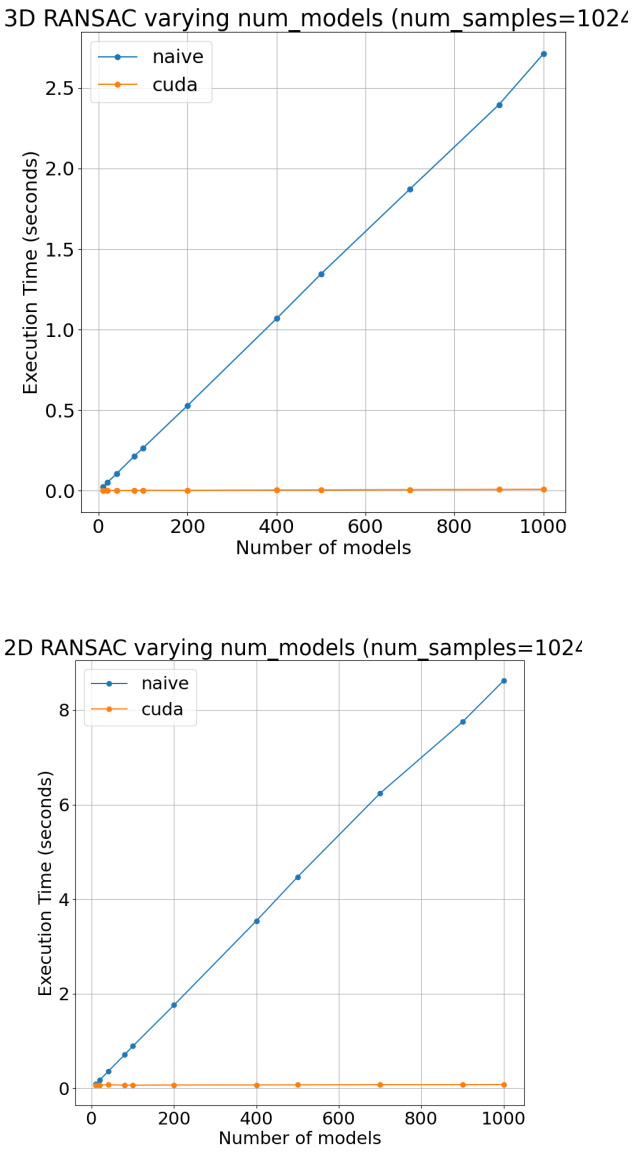

Figure 6: Increasing number of models but constant dataset size at 1024. 3D (up) and 2D (bottom)

Figure 6 shows outputs from varying numbers of models between 20 and 100 while leaving the number of points

constant at 1024. Again, the CUDA remains constant, while Python runtime blows up.

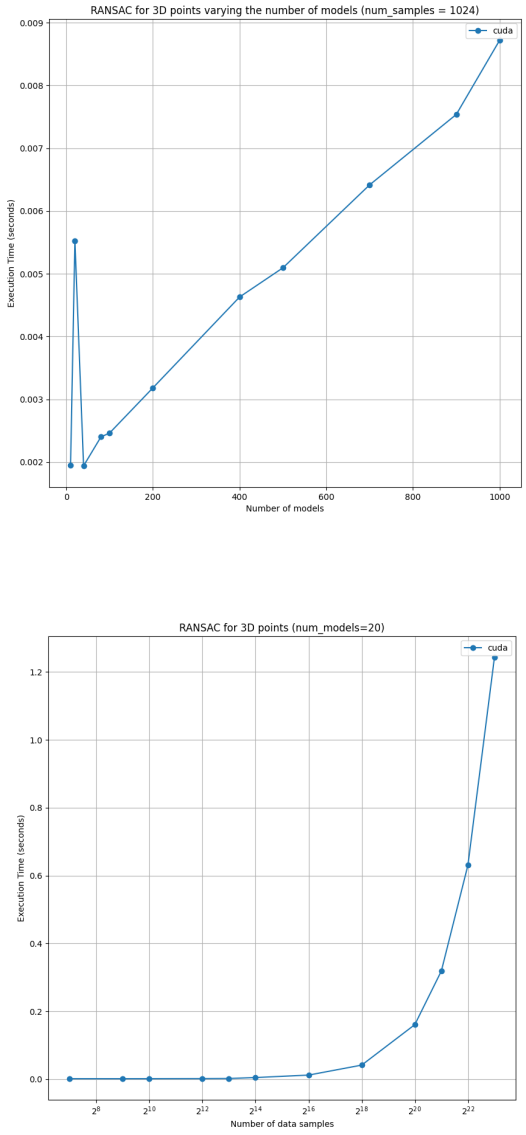

Figure 7: CUDA only runtime plots for model increase and dataset increase. Number of models (up) and Number of points (bottom)

Both shows increase but it's only relative. For the dataset increase, there is a 600x increase in runtime from 0.002s to 1.2s while for the model increase, it remains more modest with an 8x increase from 0.001s to 0.008s, which is quite insignificant compared to Python as well as the datapoint benchmark.

All our results were verified because we use simulated data to achieve all these results. We manually verified several parameters: the model ratio, the model parameters and final plot output using print statements.
Lastly, to understand how our step by step approach was performing, we broke down the performance times as shown in the following two figures.

Level 4 parallel version	Execution time (sec)
Model parameter calculation	0.003
Inlier Distance calculation	0.64
Inlier count calculation	0.78
Total	1.44

Figure 8: Execution times for Level 4 parallelization

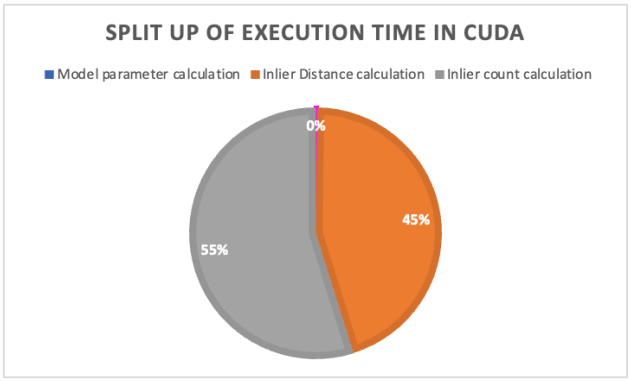

Figure 9: Pie chart for better visualization of execution times.

They show which stages of the pipeline from Figure 1 are contributing the most to the total time taken to execute. Furthermore, we used NSight to profile the compute (SM) and memory utilization for the distance calculation kernel of the 3D test cases since it takes up 45% of total runtime. We did this for different block sizes, (32, 64, 128, 256, 1024) and is displayed in the figures below.

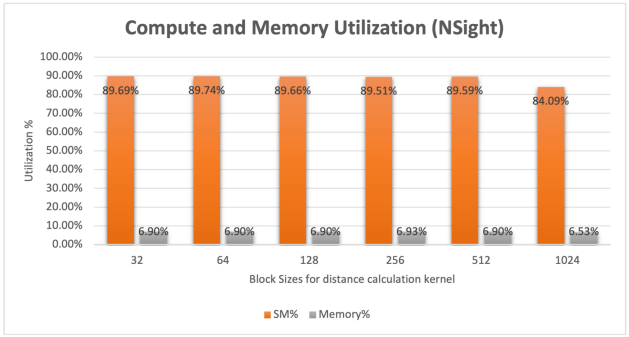

Figure 10: NSight Compute and Memory Utilization of distance calculation for 3D cases

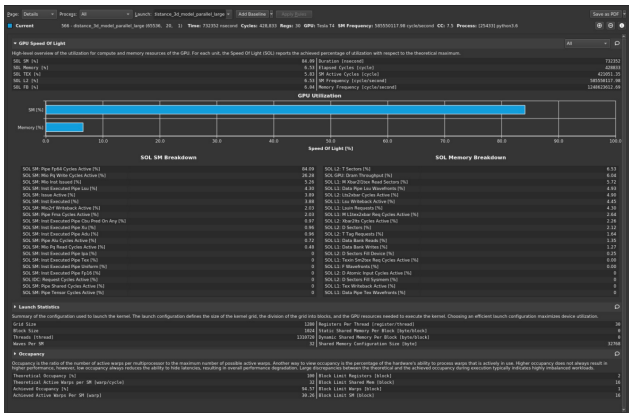

Figure 11: NSight statistics of distance calculation for 3D cases

Further analysis of these two images are in the discussion section and backed up by Figure 12 below, which also proves that compute time dominates the total computational cost.

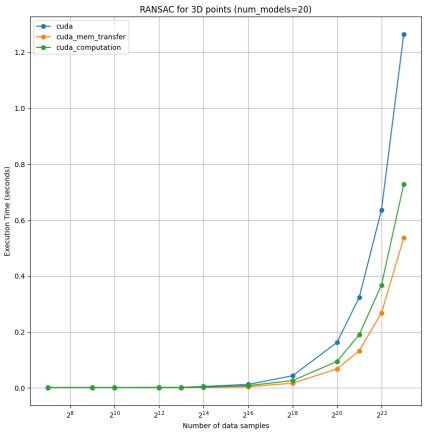

Figure 12: Execution time with and without memory transfer

One point to note is that the x-axis is logarithmic, so the exponential trend seen is actually linear with an increasing number of data samples.

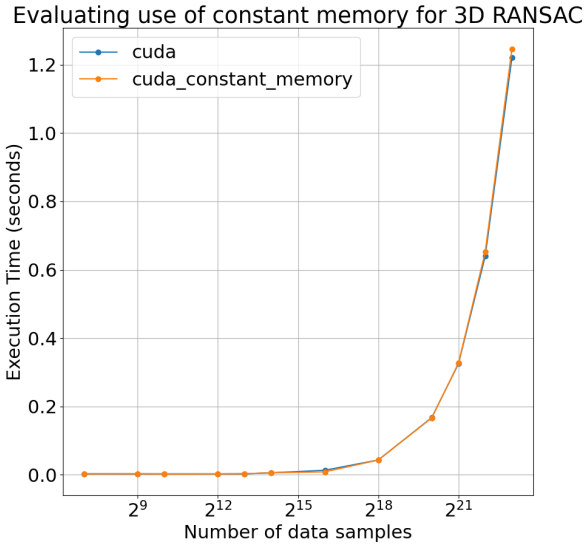

Figure 13: Testing with and without constant memory for model parameters

The figure above also shows execution time results when using constant memory to store the model parameters. Justification for this choice is mentioned in the discussion section but clearly from the plot, it makes virtually no difference.

4. Demonstration

Our real world test involved using a dataset that could verify our runtime benchmarks and proof of concepts in the 3D space. We used the KiTTI benchmark dataset that includes LiDAR point cloud data from a front facing Velodyne sensor. The goal here is to identify the ground plane from all objects in the point cloud. The model plane is fit in 3D and then all points below it are set to a different colour to show visually whether the output was correct. Unfortunately, we were not able to get accurate error metrics for this since ground truth was not available in this raw dataset.

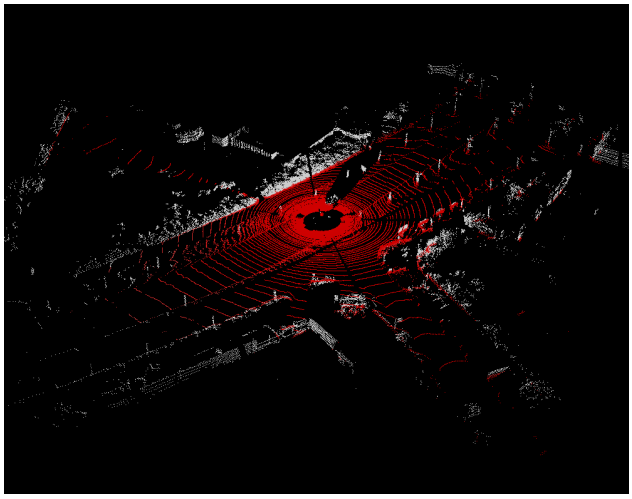

Figure 14: RANSAC on LiDAR point cloud data.

Clearly, there is some amount of error visible in Figure 14 since the lower portions of cars are also included in this ground plane. When timed, we made sure to exclude any plotting, dataset loading and dataset pre-processing. The timings are for this particular dataset, which included frames that contained ~80,000 points and had a total duration of 23s with 232 frames in total.

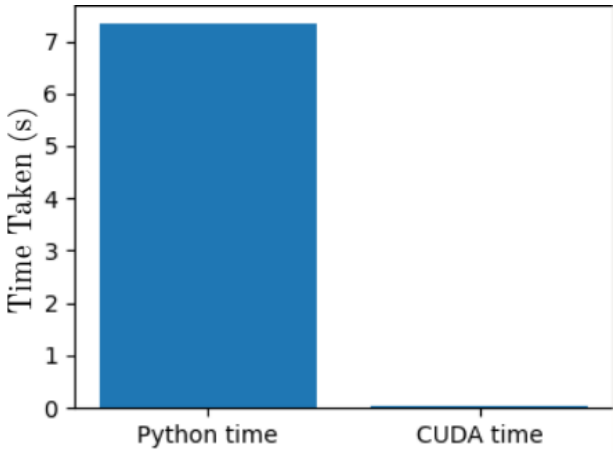

Figure 15: Time taken per point cloud frame

From Figure 15, our CUDA performs much faster with a 0.036s runtime versus our Python implementation with 7.03s runtime, which is approximately a 195x speedup.

5. Discussion and Further Work

Our results definitely show a significant difference in using our CUDA kernels. At each level of parallelization, we were able to achieve speed ups over the Python implementation. One important point to note is that the Python code was not written for optimum performance. A lot more can be vectorized and Numpy arrays along with other Numpy specific functions could be used to speed up performance. We also could use more list comprehension, since that is a more Pythonic way of handling lists. This could reduce the large gap in time taken for LiDAR data and our test data. Particularly for LiDAR data, there are libraries such as OpenCV and Point Cloud Library (PCL) that have highly optimized versions of RANSAC written in C++ that would most likely be comparable or even better than our CUDA versions.

In terms of completeness of results, our results prove that large boosts are achievable, however Figure 8 and 9 show one major area that we did not expect to take up so much time. The inlier checking and count takes up 55% of total runtime and was a surprising result because we intentionally kept it always on the CPU. It only involves a conditional check on the output array elements and checks for a ratio threshold. Additionally, we perform this operation using an optimized function, which is the logical and operator, so we did not expect it to be so significant. The parallelization of this inlier checking is definitely a next step that can be taken, and can be done using a scan kernel that only returns a single total count value that can be used for the ratio calculation. This would be ideal since we then would not even have to move the final output array back to the CPU, only the final count. Furthermore, the rest of the constants for verifying could just be stored in constant memory.

NSight did also offer some useful insights. A block size of 1024 achieved the best SM% and Memory% at 84.09% and 6.53% respectively, based on Figure 10. SM% and Memory% remained almost the same for all other block sizes though. Significantly, for all block sizes, the memory utilization is quite low at around 7% compared to a high SM% of around 89%. This could explain why the compute time dominates memory transfer time in CUDA. This also plays a role in understanding why we made certain design decisions for our memory store as outlined below.

In terms of memory utilisations, one of the big considerations was using shared memory or not. We made the decision to not use it since it did not make any practical sense. The entire dataset would be loaded into global memory first and then each point is only used once per model for the distance calculation. This is important because if we were to move to shared memory, it would mean adding an unnecessary memory read: 1 from global and then one from shared. Keeping it in global memory instead keeps it at only 1 read from global, which is better. Figure 13 also shows an example of a test using constant memory to store model parameters because these are the only variables that are being read n times where n is the number of points in the dataset. However, no noticeable execution time difference was observed, which

was again strange to us since there are so many reads and they should be faster if from constant memory over global. Maybe it is only a very small difference due to minimal reads from the same memory location.

Lastly, in terms of prior works to compare with, there is unfortunately none of significance that we could find. There is one github [5] that attempts to do this as part of a project but it was not documented well and so was quite difficult to follow along and interpret. We, however, did find several Python implementations that were all similar enough that we referenced in our serial implementation [6].

6. Conclusion

We proposed CUDA kernels to speed up several stages of the RANSAC algorithm from model estimation to error checking to the final inlier checking. One of the challenges was implementing this for large data sizes in the range of millions. We achieved speed ups of ~200x over serial Python implementations on LiDAR test data. We were able to achieve our stated goals and all the intermediate steps on the way. Importantly, we learned about the importance of thread block organization especially during our Level 4 implementation as well timing correctly. The main way to further this work would be to improve the inlier counting through the scan kernels that we proposed in the Discussion section.

7. Acknowledgements

We would like to thank Professor Kostic and TA Arjun for their support and feedback throughout the class as well as during our proposal and final presentations.

8. References

[1] “Fitting with Applications to - cs.ait.ac.th.” [Online]. Available: https://www.cs.ait.ac.th/~mdailey/cvreadings/Fischler-RANSAC.pdf.

[2] Q. X. Y. C; “NCC-RANSAC: A fast plane extraction method for 3-D range data segmentation,” *IEEE transactions on cybernetics*. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/24771605/.

[3] A. J. RaJ, “3D RANSAC algorithm for LIDAR PCD segmentation,” *Medium*, 05-Jun-2020. [Online]. Available: https://medium.com/@ajithraj_gangadharan/3d-ransac-algorithm-for-lidar-pcd-segmentation-315d2a51351.

[4] “10.3 2d Alignment - RANSAC.” [Online]. Available: http://cs.cmu.edu/~16385/s17/Slides/10.3_2D_Alignment__RANSAC.pdf.

[5] Alehdaghi, “Alehdaghi/PyPRANSAC: Python Parallel Ransac with Numba (cuda+python),” *GitHub*. [Online]. Available: https://github.com/alehdaghi/PyPRANSAC.

[6] “Robust linear model estimation using RANSAC – python implementation,” *On the way to vision... computer vision*, 27-Jun-2014. [Online]. Available: https://salzis.wordpress.com/2014/06/10/robust-linear-model-estimation-using-ransac-python-implementation/.

8. Appendices

Individual Student Contributions (in %)

Task	% Arvind	% Kyle
Overall	**50**	**50**
Python	**25**	**75**
Level 1	**25**	**75**
Level 2	**75**	**25**
Level 3	**75**	**25**
Level 4	**75**	**25**
LiDAR	**25**	**75**