ACCELERATING POINT SET REGISTRATION FOR AUTOMATED AERIAL REFUELING

THESIS

Ryan M Raettig, 2d Lt, USAF

AFIT-ENG-MS-21-M-075

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-21-M-075

Accelerating Point Set Registration for Automated Aerial Refueling

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Ryan M Raettig, B.S.EC.E.

2d Lt, USAF

March 19, 2021

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-21-M-075

Accelerating Point Set Registration for Automated Aerial Refueling

THESIS

Ryan M Raettig, B.S.EC.E.
2d Lt, USAF

Committee Membership:

Scott L Nykl, Ph.D
Chair

Laurence D Merkle, Ph.D
Member

Clark N Taylor, Ph.D
Member

AFIT-ENG-MS-21-M-075

Abstract

The goal of Automated Aerial Refueling (AAR) is to control the tanker boom to safely refuel a receiving aircraft with no input or aid from the boom operator. To move the boom into the refueling receptacle, the position and orientation (pose) of the receiver relative to the tanker must be known. In computer vision and robotics, point set registration is a fundamental issue used to estimate the relative pose of an object in an environment. In a rapidly changing scene, this method must be executed frequently and in a timely fashion, or the pose becomes outdated. One problem in AAR is the point registration method is currently a computational bottleneck of the vision processing pipeline. In addition, the matching of each sensed point with a closest truth point, nearest neighbor matching, is the most costly portion of the point set registration process. For this reason, this research focuses on speeding up a widely used point registration method, the Iterative Closest Point (ICP) algorithm and related nearest neighbor algorithms. This research lays out novel nearest neighbor matching algorithms based on the Delaunay Structure with a reduced cost compared to conventional nearest neighbor matching algorithms. Additionally, the ICP algorithm is transformed into a massively parallel algorithm and mapped onto a vector processor to realize a speedup of approximately 2 orders of magnitude. Lastly, this thesis presents algorithmic and runtime analysis with synthetic, virtual, and real experiments.

Acknowledgements

I thank my adviser Dr. Nykl for his invaluable mentorship and guidance throughout this process. He is fully dedicated to helping his students and progressing the project. He was always available day or night and I'm truly grateful to have him as my advisor. I would like to thank my committee, Dr. Merkle and Dr. Taylor, for always helping out and or explaining topics.

I would like to thank my wife Melissa for selflessly putting the family first and encouraging me to focus on research. I would like to thank my sons Ryan and Matthew for watching over Mom and understanding when I had to work countless hours.

I wish to thank Dan Schreiter, and the rest of AFRL/RQ Aerospace Systems Directorate for their support and feedback throughout this research.

Ryan M Raettig

Table of Contents

Page

Abstract iv

Acknowledgements v

List of Figures viii

List of Tables xiii

I. Introduction 1

1.1 Problem Background 1
1.2 Research Objectives 3
1.3 Document Overview 4

II. Background and Literature Review 5

2.1 Iterative Closest Point Algorithm 5
2.2 Accelerated Point Set Registration 9
2.2.1 Parallel ICP 9
2.2.2 GPU Point-Cloud Registration 9
2.2.3 Softassign EM-ICP 10
2.3 CUDA and GPUs 10
2.4 Nearest Neighbor 11
2.4.1 Delaunay Triangulation 13

III. Methodology 16

3.1 Preamble 16
3.2 Definition of Algorithm 16
3.2.1 Target Parallel Platform 17
3.3 Algorithm Decomposition and Mapping 21
3.3.1 Decomposition 21
3.3.2 Mapping 30
3.4 Optimization 36
3.4.1 Find Correspondences 36
3.4.2 Reduction 37
3.5 Algorithm Analysis 37
3.5.1 Thread Communication 37
3.5.2 Interaction Overheads 38
3.5.3 Isoefficiency Function 40
3.5.4 Scalability 41
3.6 Delaunay Traversal 42
3.6.1 Delaunay Creation 42

Page

3.6.2 Notation 42
3.6.3 Traversal 43
3.6.4 Space and Time Complexity 44
3.7 Delaunay Walk Variations 45
3.7.1 Zero Delaunay Walk 45
3.7.2 KD Approximate Delaunay Walk 45
3.7.3 Previous Nearest Neighbor Delaunay Walk 46
3.7.4 PNN Optimized Delaunay Walk 46
3.8 ICP Filter 46
3.9 Experimental Design 47
3.9.1 3D Models 49
3.9.2 Virtual and Real Stereo Block Matching 50

IV. Results and Analysis 55

4.1 Preamble 55
4.2 Performance 55
4.3 3D Models 63
4.4 Virtual and Real Stereo Block Matching 84

V. Conclusions 92

5.1 Future Work 92

Appendix A. Additional Results 94

Appendix B. Code Implementation 101

Bibliography 102
Acronyms 111

List of Figures

Figure Page

1. Steps in each ICP iteration directly from Besl's paper *A Method for Registration of 3-D Shapes* [1]. 7

2. This shows ICP broken down by steps when executing CPU *KD Tree* on the Aircraft A model with 5k points. The nearest neighbor portion of ICP is taking over 97% of the total runtime. 8

3. This shows the point cloud of a bunny model and the corresponding Delaunay triangulation connections for a single point. 15

4. Task dependency graph of accelerated ICP Steps from Figure 1 with critical path in solid red. 18

5. Task interaction graph of accelerated ICP Supertasks from Figure 1 with supertask 5 consolidated. 19

6. Task dependency graph of supertask *1. Find Correspondences* outlined in Algorithm 1. 22

7. Task dependency graph of supertask *2. Calculate Centers of Mass* 8 point reduction example outlined in Algorithm 2. 24

8. Task dependency graph of each task in supertask *3. Calculate* $\mathbf{\Sigma_{px}}$, a 1 to 1 map and then a reduction for each of the 9 values in the $\mathbf{\Sigma_{px}}$ matrix used in Algorithm 3. 24

9. Task dependency graph of supertask *4. Calculate* $\mathbf{Q}$ with critical path in solid red outlined in Algorithm 4. 26

10. Task dependency graph of supertask *5. Find Eigenvalues and Eigenvectors and Calculate* $\mathbf{R}$ *and* $\vec{t}$ outlined in Algorithm 5. 28

11. Task dependency graph of *6. Apply* $\mathbf{R}$ *and* $\vec{t}$ outlined in Algorithm 6. 28

12. Task dependency graph of *7. Calculate Error* outlined in Algorithm 7. 29

Figure Page

13. Virtual Aircraft A sensed object 48

14. Virtual Aircraft A red 5053 point truth model based on sensed object 48

15. Left and right virtual stereo cameras images showing sensed object 48

16. ICP running on yellow sensed points and red truth model points 49

17. This shows ICP running on the teapot model point cloud with 8k points. 51

18. This shows ICP running on the sports car model with 27k points. 52

19. This shows ICP running on the dragon model with 31k points. 53

20. This shows the Aircraft B(21k) model with noise added to the yellow sensed points. The truth model in red is being fitted to the yellow points with ICP. 54

21. This shows ICP executed with real and virtual and real stereo block matching. ICP is fitting the red truth model to the sensed points. The real images are in the upper right and left. The virtual images are in the lower right and left. The model being fitted is Aircraft B(21k). 54

22. This shows an AAR scenario in which the receiving aircraft is Aircraft A and it is approaching the tanker. The two virtual images in the lower left and right are from the virtual cameras on the tanker. Stereo block matching is being run on these images to produce the yellow sensed points. Then ICP is executed to fit the red truth model to the yellow points. In return, the pose of Aircraft A relative to the tanker is produced. 58

23. Graph of CPU versus GPU ICP overall runtimes versus number of sensed points. 59

24. Graph of GPU speedup over CPU versus number of sensed points. 59

Figure Page

25. Graph of nearest neighbor kernel real and theoretical runtimes versus number of sensed points. 60

26. Graph of reduction kernel real and theoretical runtimes versus number of sensed points. 61

27. CPU Nearest Neighbor Algorithm Runtime vs Model 66

28. GPU Nearest Neighbor Algorithm Runtime vs Model 67

29. ICP Runtime CPU *PNN Delaunay* vs GPU *PNN Optimized Delaunay* 68

30. ICP GPU *PNN Optimized Delaunay* Speedup over CPU *PNN Delaunay* 69

31. This graph shows an nearest neighbor algorithmic comparison of the runtime per iteration on the CPU running ICP on the Teapot(8k) model. 70

32. This graph shows an nearest neighbor algorithmic comparison of the runtime per iteration on the GPU running ICP on the Teapot(8k) model. 71

33. This shows the total ICP runtime when using the CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* nearest neighbor algorithms. 72

34. This shows the ICP speedup when using GPU *PNN Optimized Delaunay* nearest neighbor algorithms over CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* nearest neighbor algorithms. 73

35. CPU ICP with *PNN Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(5k) model. 74

36. GPU ICP with *PNN Optimized Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(5k) model. 75

37. CPU ICP with *PNN Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(63k) model. 76

Figure | Page

38. GPU ICP with *PNN Optimized Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(63k) model. 77

39. This shows the frequency of how many walks are taken by the algorithms for the 4k Teapot model. 78

40. This shows the frequency of how many walks are taken by the algorithms for the 8k Teapot model. 79

41. This shows the frequency of how many walks are taken by the algorithms for the 31k Dragon model. 80

42. This shows the frequency of how many walks are taken by the algorithms for the 62k Dragon model. 81

43. This shows the nearest neighbor runtime between CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* with noise, virtual images, and real images all on the Aircraft B(21k) model. 82

44. This shows the total ICP runtime when using the CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* nearest neighbor algorithms with noise, virtual images, and real images all on the Aircraft B(21k) model. 83

45. Errors in translation and rotation from ICP on virtual imagery on the CPU 86

46. Errors in translation and rotation from ICP on virtual imagery on the GPU 87

47. Errors in translation and rotation from ICP on real imagery on the CPU 88

48. Errors in translation and rotation from ICP on real imagery on the GPU 89

49. Errors in translation and rotation from ICP on real imagery on the GPU with filter 90

50. This shows the AAR vision processing pipeline runtime from image acquisition with real 4k images using ROI and ICP with with GPU *PNN Optimized Delaunay* on the Aircraft B(21k) model. 91

Figure Page

51. This shows the CPU *KD Tree* and GPU *PNN Optimized Delaunay* nearest neighbor runtimes. 94

52. This shows GPU *PNN Optimized Delaunay* over CPU *KD Tree* nearest neighbor speedup. 95

53. This shows the CPU *KD Tree* and GPU *PNN Optimized Delaunay* total ICP runtimes. 96

54. This shows GPU *PNN Optimized Delaunay* over CPU *KD Tree* total ICP speedup. 97

55. This shows the AAR vision processing pipeline runtime from image acquisition with real 4k images using ROI and ICP with with CPU *KD Tree* on the Aircraft B(21k) model. 98

56. This shows the AAR vision processing pipeline runtime from image acquisition with real 4k images without ROI and ICP with with GPU *PNN Optimized Delaunay* on the Aircraft B(21k) model. 99

57. This shows the AAR vision processing pipeline runtime from image acquisition with real 2k images without ROI and ICP with with GPU *PNN Optimized Delaunay* on the Aircraft B(21k) model. 100

List of Tables

Table Page

1. Kernel portion of runtime and kernel runtimes reported by *nvprof*. 62

2. CPU and GPU runtimes and number of points. 62

3. GPU speedup over CPU and number of points. 62

4. Nearest neighbor kernel real and theoretical runtimes versus number of sensed points. 62

5. Reduction kernel real and theoretical runtimes versus number of sensed points. 62

6. This shows the positional mean magnitude error in meters as well as the 99.5% confidence interval. 85

7. This shows the rotational mean magnitude error in degrees as well as the 99.5% confidence interval. 86

8. This shows the SBM/Reprojection mean magnitude runtime in seconds as well as the 99.5% confidence interval. 86

Accelerating Point Set Registration for Automated Aerial Refueling

I. Introduction

1.1 Problem Background

Automated Aerial Refueling (AAR) intends to supplement United States Air Force (USAF) refueling missions by achieving the capability to control the boom and safely refuel, independent of input from the boom operator. Using the tanker's stereo-vision cameras and a computer vision processing pipeline, the receiving aircraft's relative position and orientation (pose) can be estimated. From this information, the tanker's boom can be safely moved into the receiver's refueling receptacle.

Biological systems provide the ability for humans to rapidly process imagery, detect objects, and determine their pose relative to the scene. These biological systems contribute to a human's perception. This process happens quickly, enabling humans to make responsive decisions in a rapidly changing environment. In contrast, with respect to robotics, self-driving vehicles, and autonomous systems, these biological systems must be replaced by hardware and software using computer vision. This process utilizes image and vision processing algorithms to enable these inorganic systems to sense their surroundings in dynamic environments.

One method employed in vision systems is to capture images of the environment and generate *sensed point* clouds through feature detection and stereo block matching [2] between pairs of images. These sensed points must then be processed to create a semantic interpretation of an environment. *Point set registration* [3, 4] is a method that can extrapolate information from these sensed points. Point set registration

aligns point sets via a rigid transformation [5]. This rigid transformation gives relative pose information. It enables the autonomous system to align these sensed points with a known truth model. In this fashion, the autonomous system can estimate the sensed object's pose from the camera.

The human retina has a higher concentration of cells at the center of the retina called the fovea [6]. Thus, humans have a higher resolution at center of their field of view, as more detail provides more information for perception. This same concept can be seen in computer vision systems. Higher resolution images provide a more accurate representation of objects. However, this comes at the cost of an exponential computational growth based on pixel density. Thus, processing times increase when the point set registration method saturates or uses all available hardware processors. This issue inhibits the autonomous system from rapidly processing imagery and quickly making decisions or actions on it. For this reason, either reducing the number of sensed points or accelerating the point registration method is required to achieve real-time object registration. However, reducing the number of sensed points reduces the information gained from the imagery, leading to a less accurate perception of objects' classification and pose. This concept illuminates the value and the problem solved by a parallel point set registration method.

In the Iterative Closest Point (ICP) [1] registration method, *nearest neighbor* matching is the first and most costly step. In order to accelerate ICP, the nearest neighbor matching must also be accelerated. Nearest neighbor matching refers to assigning the most similar points based on a criterion [7]. Practically, it's assigning a closest point for each point between the two point clouds. In addition to ICP, the nearest neighbor algorithm is used in machine learning [8], robotics [9] and pattern recognition [10]. For this reason and real-time capabilities, this research focuses on accelerating the Euclidian distance pairwise point set registration method of ICP as a

whole. Additionally, specifically in ICP, the nearest neighbor matching process is accelerated by leveraging previous nearest neighbor matches. Lastly, these accelerations can be applied in any application which utilizes the nearest neighbor algorithm.

1.2 Research Objectives

Currently in the Air Force Institute of Technology (AFIT) AAR computer vision processing pipeline, the point set registration process is a bottleneck. This research aims to accelerate the point set registration in the computer vision processing pipeline to calculate the pose of the receiving aircraft in AAR. This research contributes the following.

1. The point set registration method of the ICP algorithm is transformed from a serial algorithm to a massively parallel algorithm that executes efficiently on a vector processor such as a graphics processing unit (GPU).

2. The parallel and novel algorithms are analyzed and theoretical and real runtimes are compared.

3. A novel algorithm for nearest neighbor matching is introduced based on the Delaunay triangulation.

4. Additionally, this research implements and tests different combinations of nearest neighbor matching algorithms in the ICP algorithm to realize a speedup of approximately 2 orders of magnitude.

5. Registration of augmented, virtual, and real sensed points are compared for robustness.

6. Lastly, the AAR aspects of the tanker and receiver trajectory are simulated in a virtual 3D world and a real-world Vicon 3D motion capture system [11].

1.3 Document Overview

Chapter II presents a literature review and background on ICP, accelerating point set registration, and the aspects to the novel nearest neighbor algorithm. Chapter III presents the methodology and includes: the definition of the accelerated ICP algorithm, target parallel platforms, decomposition of the algorithm, optimizations, analysis of the algorithm, the Delaunay nearest neighbor algorithm, and experiments. In Chapter IV, results are presented. Lastly, Chapter V concludes this research. Portions of this thesis have been submitted as journal papers that are under review [12, 13].

II. Background and Literature Review

This chapter provides the background and related work for this thesis. Section 2.1 covers the Iterative Closest Point (ICP) algorithm and other point set registration methods. Section 2.2 discusses various accelerated point set registration methods. Section 2.3 overviews Compute Unified Device Architecture (CUDA) and GPUs. Section 2.4 outlines background on the nearest neighbor problem. Finally, Section 2.4.1 introduces the Delaunay trianglution.

2.1 Iterative Closest Point Algorithm

As originally discussed in [12], the ICP algorithm registers a sensed 3-D point cloud onto a reference truth 3-D point cloud. In Automated Aerial Refueling (AAR) and this research, ICP is used to transform a truth model onto a sensed point cloud to recover a rotation and translation from the stereo-vision cameras on the tanker aircraft. ICP can be used to align a plethora of shapes as long as they are approximated as point sets. Besl [1] discusses the parameters to which the ICP algorithm can be applied: "1) sets of points, 2) sets of line segments, 3) sets of parametric curves, 4) sets of implicit curves, 5) sets of triangles, 6) sets of parametric surfaces, and 7) sets of implicit surfaces." Besl refers to the shape the ICP algorithm is trying to align as P. In the real world, P inherently comes from a sensor like a high definition camera. To align P, ICP must have prior knowledge of the known truth model. Besl refers to this truth model as X. In this fashion, the ICP algorithm aligns P onto X.

The following list explains the ICP algorithm from Besl [1].

1. Assign nearest neighbor correspondences between P and X and assign the correspondences to Y.

2. Compute the registration or the rotation and translation that transforms P onto X.

3. Apply the registration to P and calculate the error between P and X.

4. If the error is lower than a set threshold, the algorithm exits returning the rotation and translation. Otherwise, the algorithm iterates again to find a more accurate rotation and translation by minimizing this error.

Figure 1 shows these steps from Besl [1].

Several variants exist for each of these steps to improve accuracy and speed, particularly the second step. Where Besl utilized point-to-point correspondences, Chen modifies the algorithm by generating point normals for the truth model and matching the query point to the plane defined by the truth point and its normal vector, typically cited as “point-to-plane” [14]. Since this method registers source points to the area around the target point instead of the target point itself, the algorithm’s sensitivity to noise can be reduced [15]. A downside of point-to-plane is the correspondence calculation will cause each iteration to take longer than the point-to-point approach; however, point-to-plane converges in fewer iterations, thus the timing for each algorithm ends up being nearly equal [16]. Extending point-to-plane, generalized ICP describes a plane-to-plane method [17]. In plane-to-plane, normal vectors are calculated on both target and source point clouds. This method can be shown to be more robust; however, since surface normals need to be calculated, point-to-plane is typically not utilized for real-time ICP applications.

Whereas the methods in [14, 17] focus on the accuracy of the second step of Besl’s ICP, these components of the algorithm execute relatively quickly when compared to the nearest neighbor step. As seen in Figure 2, a large portion of an ICP iteration’s runtime is spent generating nearest neighbor correspondences. An approach to speeding up the nearest neighbor phase of ICP is to cache the correspondences found in the

a. Compute the closest points: $Y_k = \mathcal{C}(P_k, X)$ (cost: $0(N_p N_x)$ worst case, $0(N_p \log N_x)$ average).

b. Compute the registration: $(\vec{q}_k, d_k) = \mathcal{Q}(P_0, Y_k)$ (cost: $O(N_p)$).

c. Apply the registration: $P_{k+1} = \vec{q_k}(P_0)$ (cost: $O(N_p)$).

d. Terminate the iteration when the change in mean-square error falls below a preset threshold $\tau > 0$ specifying the desired precision of the registration: $d_k - d_{k+1} < \tau$.

Figure 1: Steps in each ICP iteration directly from Besl's paper *A Method for Registration of 3-D Shapes* [1].

previous iteration. A cached kd tree, which executes in constant time, is presented in [18], where a pointer to the node within the tree is utilized as the starting node of the kd tree search after the first iteration. In [19], a similar approach is utilized where the previous correspondence is utilized as an estimated neighbor, and the neighbor calculation is expedited if this estimation matches certain geometric constraints.

The accelerated ICP algorithm described in this thesis is based on the "classic" *point-to-point* ICP algorithm by Besl. A method which utilizes a point-to-point correspondence is generally more accurate but can be time consuming because of the nearest neighbor search task [20]. If the nearest neighbor search is accelerated, this approach's accuracy and timeliness makes it a viable candidate for real-time applications like AAR.

In contrast to the previous methods, the *point-to-projection* registration approach eliminates the need for finding the closest point or correspondences but generally gives a less accurate pose estimation [20]. From Park [20]: "...searching the closest point in the ICP algorithm is a computationally expensive task." Park explains how this task is replaced by point-to-projection [20]: "...*point-to-projection* approach finds

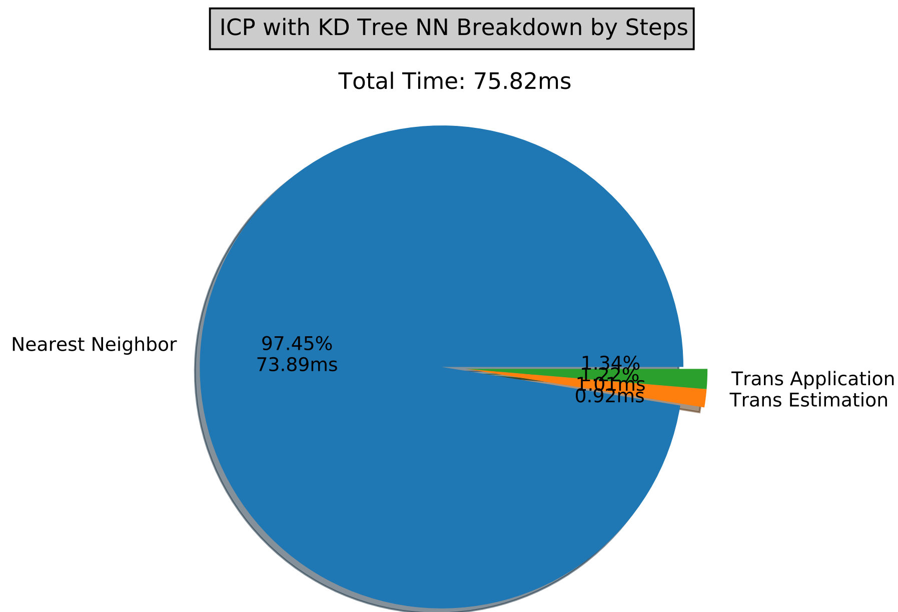

Figure 2: This shows ICP broken down by steps when executing CPU *KD Tree* on the Aircraft A model with 5k points. The nearest neighbor portion of ICP is taking over 97% of the total runtime.

the correspondence of a source control point by projecting the source point onto a destination surface..."

Point-to-(tangent)-plane [21] is the most accurate approach but most computationally expensive [20]. In this approach, both the nearest neighbor correspondences and the intersection surface needs to be found [20]. An approach of brute force [22], kd tree [23][24][25], Delaunay traversal [26][13], or some other variation can be used to execute nearest neighbor [27] correspondences. Presented in this thesis, the same parallel mapping and optimizations strategies can be used as a guidelines regardless the preferred ICP variation or nearest neighbor approach.

2.2 Accelerated Point Set Registration

2.2.1 Parallel ICP

Langis [28] implemented a parallel ICP algorithm in a parent-child model [29]. The parent process spawns child processes to compute nearest neighbor correspondences. The children report these correspondences to the parent. The parent then calculates the rotation and translation based on these correspondences and applies the rotation and translation to P. From Langis on explaining the parallel iterations [28]: "Each child concurrently computes the correspondences between points in I_f, and the points of I_r. The resulting correspondences are then sent back to the parent process." I_f and I_r represent the sensed and truth model in this example. The parent process becomes the bottleneck in this model with potentially many children attempting to report concurrently.

2.2.2 GPU Point-Cloud Registration

Rahman [30] implemented a fast graphics processing unit (GPU) point-cloud registration algorithm mapped into 4 blocks on the GPU. In this example, a block is

a separate process mapped on a GPU. The first block finds the centers of mass of the point clouds. The second block transforms the point clouds to the origin. The third block executes the Singular Value Decomposition (SVD) [31] technique, a very small computational process. The fourth block finds the rotation and translation. In contrast to Rahman's algorithm based on SVD to align two point clouds, the implementation presented in this thesis utilizes Besl's ICP algorithm.

2.2.3 Softassign EM-ICP

Tamaki [32] implemented an accelerated Softassign [33] Expectation-Maximization (EM)-ICP [34] algorithm based in CUDA. The algorithm accelerations were predicated on assigning estimated nearest neighbor correspondences. Based on these estimated correspondences, an estimated rotation and translation was calculated. In contrast, the accelerated ICP algorithm in this thesis computes exact nearest neighbor correspondence and thus gives a more accurate rotation and translation transformation.

2.3 CUDA and GPUs

GPUs were originally made for computer graphics processing to concurrently execute vertex and pixel pipelines. Thus, GPUs have many cores for processing. For this reason, developers aimed to utilize these cores for not just graphics, but also general purpose computing. Nvidia introduced CUDA for developers to do just that through one application programming interface (API).

Nvidia released the first version of CUDA, 1.0, on 23 June 2007 [35]. CUDA presents a single program multiple data (SPMD) [29] model to the GPU. CUDA is specifically designed for Nvidia GPU architectures [35]. As of writing this thesis, the current version of CUDA is 11.2. As Nvidia GPU architectures advanced, various

features have been added to CUDA. The compute capability of an Nvidia architecture correlates to the supported features of CUDA on said architecture [36].

Oden [37] goes over an interesting discussion of Python packages that utilize pre-compiled and just-in-time compiled C-CUDA for GPU implementations. Numba-CUDA [38] is a Python package that utilizes just-in-time compiled C-CUDA. In every practical sense, the CUDA presented in this thesis is native C-CUDA. Oden explains that using Python for GPU implementations is not a preferable avenue for performance because the user is limited to these libraries. Oden covers metrics that show C-CUDA libraries are faster than Numba-CUDA libraries. Oden shows C-CUDA Python packages reaching 85% performance and Numba-CUDA Python packages reaching 50% performance [37]. With this in mind, using the lowest level of native C-CUDA programming language directly yields the best performance increases because the most control is given to the user. For this reason, native C-CUDA will be used in this research to implement an accelerated ICP algorithm.

In terms of utilizing a GPU to accelerate nearest neighbor searches, many methods take advantage of the massive parallelism available. For example, [39] utilizes the CUDA and CUBLAS libraries to do high dimensional feature matching. One method utilizing the GPU loads the point data into texture memory and shader programs calculate distances and determine nearest neighbor [40]. Another method runs two brute force searches in parallel along lists where each element owns a subset of the search space, proven to work $\mathcal{O}(\sqrt{n})$ [41].

2.4 Nearest Neighbor

The search for the most similar matches between sets of vectors or points is nearest neighbor matching [42]. Research into the nearest neighbor problem has spanned numerous fields, including pattern recognition [10], machine learning [8], and robotics

[9]. Some of these require an ordered list of similar data, while others only require a closest point. Depending on the metric used for similarity, the closest point may or may not be unique. Typically, some distance metric such as Euclidean distance, Manhattan distance, or squared distance is utilized depending on the application. For this research, Euclidian distance-based pairwise nearest neighbor matching is used in ICP and assigns a closest point for each point between the two point clouds.

The generic method for determining nearest neighbors is commonly referred to as *brute force*. With this method, the distance between the source point and every other point in the target set is computed and the closest point is returned. If multiple nearest neighbors are required, the algorithm can be modified such that the distance from source to each target point is stored, then the target points are sorted based on that distance. Assuming equal size point sets, the nearest neighbor algorithm results in a time complexity of $\mathcal{O}(n)$ to match a single point and $\mathcal{O}(n^2)$ to match all points.

In order to reduce the time complexity for a nearest neighbor search, applications typically utilize a kd tree structure. A kd tree is a space partitioning k-dimensional binary tree that organizes a data set into k dimensions [43]. A kd tree is built by splitting the space into a hyperplane created at the median of the given dimension. The points are split along this partition and are subsequently divided based on the other dimensions. To query the kd tree for the nearest neighbor to a given point, the algorithm begins at the root node and traverses down the tree choosing which child node to visit based on which side of the hyperplane the query point exists [44]. After reaching a leaf, the algorithm back-traces back through the tree checking the distance to each hyperplane to determine if the adjacent space needs to be checked for a closer point. Since a binary tree has $log_2(n)$ levels, the average query time is proportional to $\mathcal{O}(log(n))$ with a worst cast of having to search the entire tree being $\mathcal{O}(n)$.

2.4.1 Delaunay Triangulation

As originally discussed in [13], Delaunay triangulation has shown promising aspects to accelerate nearest neighbor searches. In two-dimensions, a Delaunay triangulation of a set of points is a set of triangles such that no points are located within the circumcircles created by the vertices of each triangle. When extended to three-dimensions, the triangles become tetrahedrons, and the circumcircles become circum-spheres, and the condition of no points being located in each circum-sphere holds. The concept can be extended to d-dimensions [45]; however, the work presented in this thesis is limited to three-dimensions.

The creation of the Delaunay triangulation structure can be completed offline. Lee [46] goes over a divide-and-conquer and an iterative algorithm to create the Delaunay triangulation structure. In this thesis, the Open3D library [47] is used to create the Delaunay Triangulation. Open3D uses the Qhull library [48] to compute the convex hull of the model and then stores the triangulation as a series of tetrahedrons.

Delaunay triangulations allow for many applications including path planning [49, 50, 51] and surface reconstruction [52, 53, 54]. Here is an application uses Delaunay triangulation to generate nearest neighbors within a data set with the goal of calculating strain [55]. Mulchrone removes the edges that form the convex hull and the remaining edges connect the nearest neighbors. Similarly, in [56] Delaunay triangulations are utilized to cluster points by removing undesired and redundant edges. These examples show Delaunay triangulation can be used to efficiently assign a variety of point correspondences.

While these methods identify neighbors within a data set, the Full Delaunay Hierarchies (FDH) [57] algorithm utilizes a Delaunay traversal to determine the nearest neighbor of a source point to a target dataset. The traversal in FDH only moves from vertices of lower index to higher index, disallowing traversal in both directions.

This detail increases the complexity of implementing the algorithm. Additionally, the authors state their method cannot directly extend to greater than two dimensions.

Another method for searching a dataset for nearest neighbors is an octree constructed utilizing Voronoi cells. Since a Delaunay triangulation is the mathematical dual of a Voronoi diagram [45], this structure [58] is intrinsically connected to a Delaunay triangulation. The octree described in [58] is built not on the individual points of the dataset, but on the Voronoi cells computed from the original points. Voxels of the tree contain at most M_{max} intersecting Voronoi cells. These voxels are then accessed through a hash table where each entry is indexed through its level in the tree.

While some applications require an exact nearest neighbor, some only require an approximate match. Typically, an approximate neighbor search returns a result quicker than an exact search. A straightforward approach utilizing a kd tree is to simply traverse to the leaf node corresponding to the query point [19]. Additionally, this method lessens the space complexity for storing the data structure, as only the leaf nodes and median values for each hyperplane axis needs to be stored. In [59], a method is presented that can configure itself to the desired degree of accuracy required for the application, allowing the user to balance between precision and speed. In [60], the k-nearest approximate neighbors are calculated in constant time by utilizing locally sensitive hashing to partition datasets into clusters.

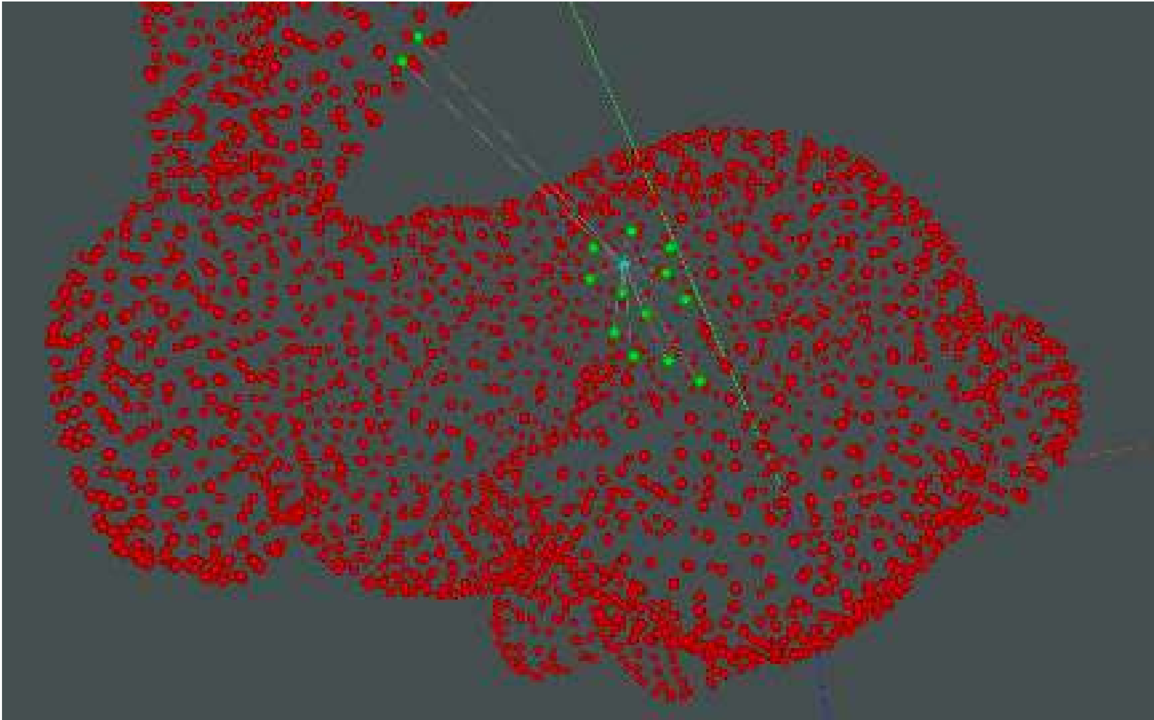

Figure 3: This shows the point cloud of a bunny model and the corresponding Delaunay triangulation connections for a single point.

III. Methodology

3.1 Preamble

This thesis aims to accelerate point set registration for Automated Aerial Refueling (AAR). This is accomplished by accelerating the Iterative Closest Point (ICP) algorithm. Additionally, a fast nearest neighbor search is implemented in ICP based on the Delaunay triangulation.

Section 3.2 defines, decomposes, and maps the accelerated ICP algorithm. Section 3.5 analyzes the algorithm. Section 3.6 covers the Delaunay triangulation creation, notation, and traversal. Section 3.7 presents the novel Delaunay walk variations. Section 3.9 outlines experiments to compare a central processing unit (CPU) and graphics processing unit (GPU) ICP implementation as well as various nearest neighbor algorithms.

3.2 Definition of Algorithm

As originally discussed in [12], the goal of this research is to accelerate the ICP algorithm through parallelism to quickly align sensed points to a known reference model via a rotation and translation that minimizes the absolute pairwise Euclidean distance, or error, between these sets. Thus, the steps in Figure 1 are parallelized by the accelerated ICP algorithm. Figure 4 shows the task dependency graph. Figure 5 shows the task interaction graph. This graph shows privatization [29, 61] will work as an optimization strategy by replicating data and reducing data contention amongst tasks.

The accelerated ICP algorithm must execute sequentially through adjacent tasks in the solid red loop in Figure 4. This solid red loop represents the critical path of the accelerated ICP algorithm. In this way, the critical path is the combination of

the critical paths of each ordered task. Supertasks *2. Calculate Centers of Mass* and *3. Calculate* $\mathbf{\Sigma_{px}}$ may be executed in parallel, but *3. Calculate* $\mathbf{\Sigma_{px}}$ is in the critical path. Besides tasks with all-to-one reductions, the critical path within each task is similar. In tasks with reductions, the critical path is $log_2(n)$, which is the number of steps or levels in the reduction.

For the remainder of the thesis, the following symbols and rules are used. X is the reference or truth model. P_0 represents the original sensed points. P_k is the kth iteration of the registration applied to P_0. Y is the list of nearest neighbors or correspondences of P_k and X. The sets: X, P_0, P_k, and Y contain 3x1 column vectors of 3-D points. $\mathbf{\Sigma_{px}}$ is a cross-covariance matrix generated between P_0 and Y. $\mathbf{Q}$ is a 4x4 symmetric matrix from which the unit eigenvector corresponding to the maximum eigenvalue represents the axis of rotation. Matrices are column-major and indexed by a $(row, column)$ subscript in the algorithms. Lastly, the optimal rotation and translation are represented by $\mathbf{R}$ and $\vec{t}$ respectively.

In the sections and ideas that follow the accelerated ICP algorithm has been implemented via Compute Unified Device Architecture (CUDA). Because the single processing units in CUDA are threads, the term *thread* is used throughout this thesis. However, *thread* and *process* [29] encapsulate the same idea of a single processing unit.

3.2.1 Target Parallel Platform

In the first set of experiments, the target parallel platform in this research is an Nvidia Titan V GPU. This is compared against the target platform for the serial implementation of ICP, which is a machine with an Intel Xeon CPU with 3.10 GHz clock speed and 128 GB of RAM.

The Nvidia Titan V GPU has 5120 CUDA cores and delivers up to 110 teraflops [62]. The Titan V has a core clock speed of 1200 MHz and a memory clock speed of

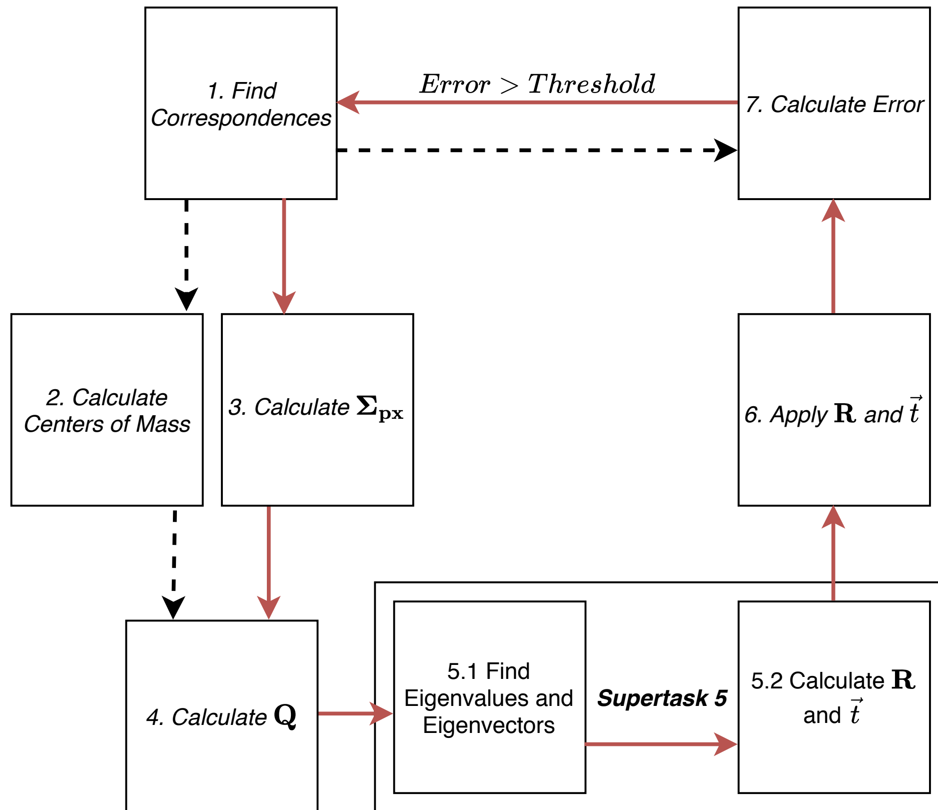

Figure 4: Task dependency graph of accelerated ICP Steps from Figure 1 with critical path in solid red.

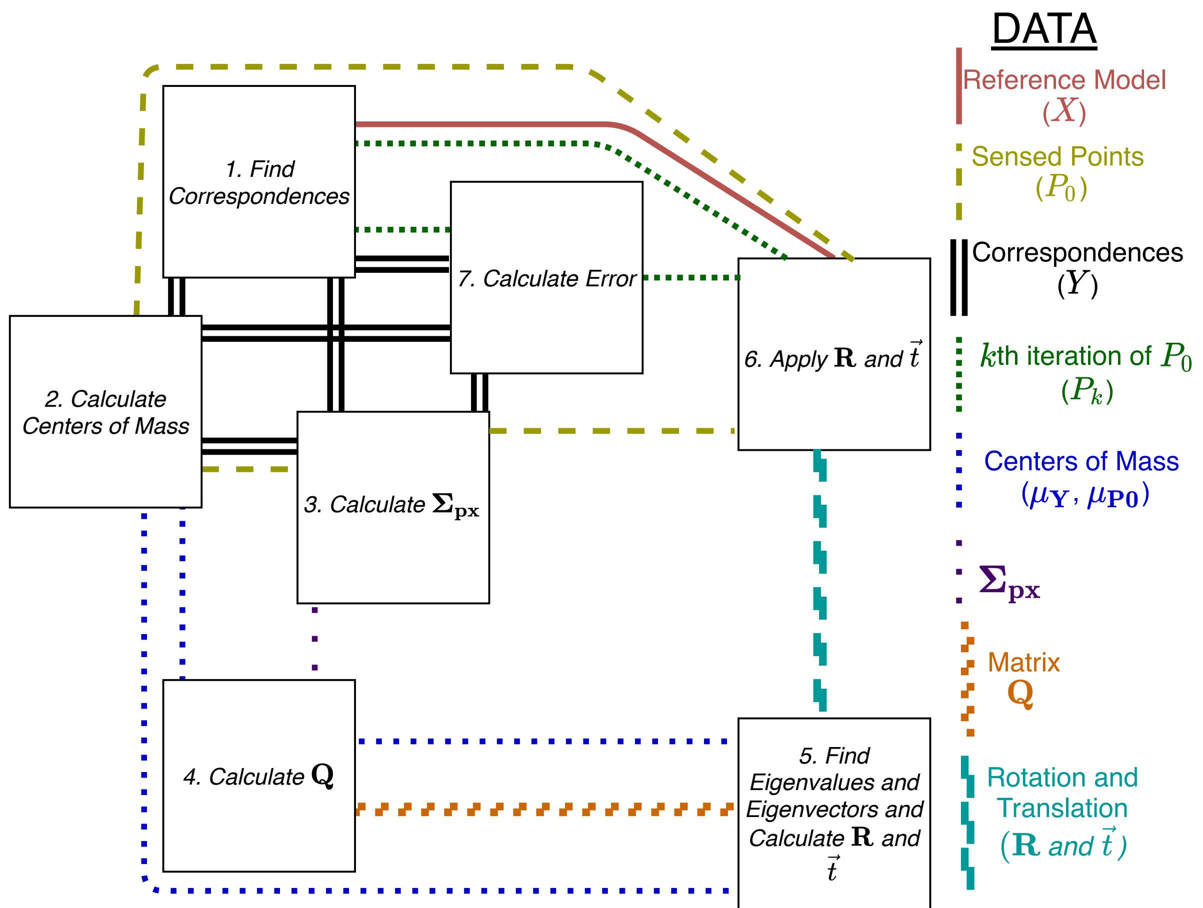

Figure 5: Task interaction graph of accelerated ICP Supertasks from Figure 1 with supertask 5 consolidated.

850 MHz. The memory bandwidth is 652.8 GB/s and the L2 Cache Size is 4608K [62]. The Titan V has an Nvidia Volta architecture with a compute capability of 7.0 [63]. The Volta boasts the NVLink, an interconnect between GPUs with a bandwidth of 300GB/s [63]. However, this feature is only active when multiple GPUs are used in an application.

In the second set of experiments, the CPU being used in these experiments is the AMD Ryzen Threadripper 3970X 32-Core processor [64] at 3.9 GHz with 64 Gb of RAM. The GPU is the Nvidia GeForce RTX 3080 [65]. Both of these pieces of hardware are relatively top-of the-line per the time of writing this thesis.

The control structure used through CUDA is the single program multiple data (SPMD) [36]. CUDA version 11.2 is used in this accelerated ICP algorithm implementation [36]. In this fashion, one program is compiled and executed on the GPU, but each CUDA core has a specific control sequence mapped to it by the program. CUDA cores in the same block can communicate via shared memory. CUDA cores in different blocks communicate via global memory in distributed memory.

The accelerated ICP algorithm employs the data-parallel model [29]. The tasks are decomposed and mapped in such a way to take advantage of data-parallelism at the thread level inside CUDA kernels in the GPU. Each task takes advantage of block distributions of the data into the thread blocks, increasing the locality of interaction within the thread block and splitting the computation between threads. The CPU-GPU CUDA setup can be thought of as a manager-worker setup. Here, the CPU is the manager and the GPU is the worker. In this setup, multiple GPUs could be added as workers. However, in this research, only one GPU is working.

3.3 Algorithm Decomposition and Mapping

The data decomposition technique [29] is used throughout the accelerated ICP algorithm. The nature of CUDA enables for a very fine-grain task decomposition. For this reason, the granularity of the decomposition leverages any possible parallelism in the algorithm amongst threads.

3.3.1 Decomposition

The following list contains the supertasks from the decomposition of the accelerated ICP algorithm.

1. *Find Correspondences*
2. *Calculate Centers of Mass*
3. *Calculate* $\mathbf{\Sigma_{px}}$
4. *Calculate* $\mathbf{Q}$
5. *(5.1) Find Eigenvalues and Eigenvectors and (5.2) Calculate* $\mathbf{R}$ *and* $\vec{t}$
6. *Apply* $\mathbf{R}$ *and* $\vec{t}$
7. *Calculate Error*

1. Find Correspondences This is the first supertask in the algorithm and it finds the nearest neighbor matches between the sensed points and the reference model. Algorithm 1 shows the pseudo code for each thread. Squared Euclidean distance is used to eliminate the expensive computational need of a square root. Each thread i takes in X and P_k and assigns $Y[i]$ to the point in X closest to $P_k[i]$. In this manner, a thread is launched for each member in P_k. In the real world, P_k can vary from a couple hundred points to thousands of points. The number of

points depends on sensor configuration and the accuracy needed. The accelerated ICP implementation has been tested on point cloud sizes from 1k to 40k and usually iterates about 30 times to achieve an error below 1×10^{-6}. Figure 6 shows the task dependency graph for tasks inside *1. Find Correspondences*. Because this is a one-to-one mapping, this supertask has a degree of concurrency equal to the number of sensed points.

Algorithm 1 Supertask: *1. Find Correspondences* pseudo code.

1: **function** NEARESTNEIGHBOR($P_k[i]$, X, $Y[i]$)
2: $\quad d_{min} = \max(float)$
3: $\quad j = -1$
4: $\quad counter = 0$
5: $\quad$**for each** $\mathbf{x} \in X$ **do**
6: $\quad\quad$**if** $squaredDistance(\mathbf{x}, P_k[i]) < d_{min}$ **then**
7: $\quad\quad\quad j = counter$
8: $\quad\quad\quad d_{min} = squaredDistance(\mathbf{x}, P_k[i])$
9: $\quad\quad$**end if**
10: $\quad\quad counter = counter + 1$
11: $\quad$**end for**
12: $\quad Y[i] = X[j]$
13: **end function**

2. Calculate Centers of Mass This step maps to the beginning of Besl's "Compute the registration" [1]. This step computes the estimated rotation ($\mathbf{R}$) and translation ($\vec{t}$) between the sensed points and the reference model. The centers of mass

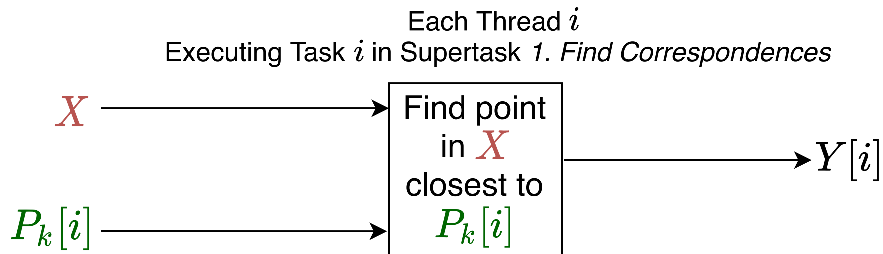

Figure 6: Task dependency graph of supertask *1. Find Correspondences* outlined in Algorithm 1.

of Y and P_0 are calculated in this supertask. Algorithm 2 shows the pseudo code for this supertask. In this example, $\mu_{\mathbf{Y}}$ is the center of mass of Y. $\mu_{\mathbf{P0}}$ is the center of mass of P_0. This supertask consists of two all-to-one reductions. Figure 7 shows the task dependency graph of a single reduction for tasks inside *2. Calculate Centers of Mass*. Because this is a binary tree reduction, this supertask has a maximum degree of concurrency of half the number of sensed points. However, this supertask has an average degree of concurrency of the number of threads active throughout the binary tree reduction divided by the number of levels in the tree. Let n be the number of sensed points. The average degree of concurrency is $\frac{2n-1}{log_2(n)+1}$.

Algorithm 2 Supertask: *2. Calculate Centers of Mass* of P_0 and Y pseudo code.

1: **function** CenterOfMass(C, μ)
2: $\quad \mu = \frac{\sum C}{|C|}$
3: **end function**
4: $CenterOfMass(P_0, \mu_{\mathbf{P0}})$
5: $CenterOfMass(Y, \mu_{\mathbf{Y}})$

3. Calculate $\mathbf{\Sigma_{px}}$ In this supertask a $\mathbf{\Sigma_{px}}$, a 3x3 matrix, is generated from the point correspondences. Algorithm 3 shows the pseudo code to calculate $\mathbf{\Sigma_{px}}$. In this algorithm, $\mathbf{I_3}$ represents a 3x3 identity matrix. Figure 8 demonstrates the first step of this supertask is a one-to-one map where each thread computes a dot product. The degree of concurrency of the map is the number of sensed points. In the final portion of this supertask, summing the dot products is an all-to-one binary tree reduction. As stated previously, if n is the number of sensed points, the average degree of concurrency for this supertask is $\frac{2n-1}{log_2(n)+1}$.

4. Calculate $\mathbf{Q}$ This supertask generates $\mathbf{Q}$, a 4x4 matrix, from which $\mathbf{R}$ and $\vec{t}$ will eventually be calculated. Algorithm 4 shows the pseudo code for *4. Calculate* $\mathbf{Q}$. The centers of mass and $\mathbf{\Sigma_{px}}$ were calculated in the previous two supertasks.

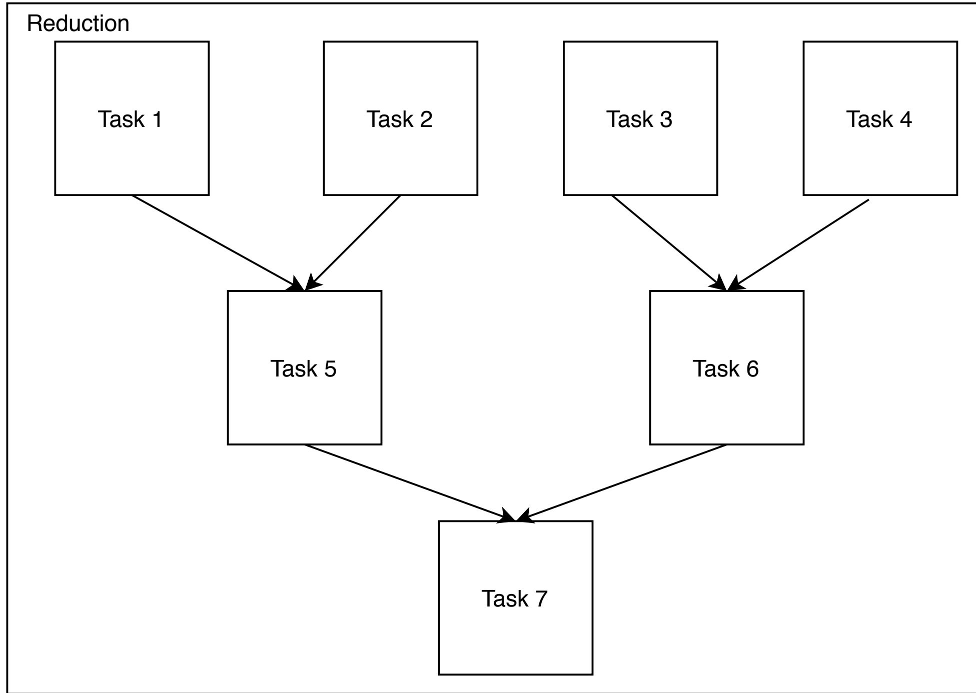

Figure 7: Task dependency graph of supertask *2. Calculate Centers of Mass* 8 point reduction example outlined in Algorithm 2.

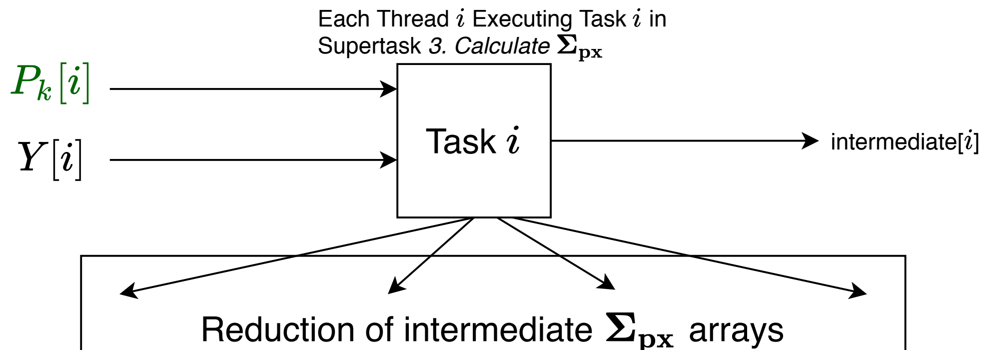

Figure 8: Task dependency graph of each task in supertask *3. Calculate* $\mathbf{\Sigma_{px}}$, a 1 to 1 map and then a reduction for each of the 9 values in the $\mathbf{\Sigma_{px}}$ matrix used in Algorithm 3.

Algorithm 3 Supertask: *3. Calculate* $\mathbf{\Sigma_{px}}$ pseudo code.

```
1: function DOT_Σpx(P0, Y, Σpx)
2:     F = I3
3:     for each x, y ∈ P0, Y do
4:         B = xᵀ · y
5:         F = F + B
6:     end for
7:     Σpx = F / |P0|
8: end function
```

Figure 9 shows the task dependency graph for the tasks within *4. Calculate* **Q**. The maximum degree of concurrency for this supertask is 16.

Algorithm 4 Supertask: *4. Calculate* **Q** pseudo code.

1: **function** CALC_Q($\mathbf{\Sigma_{px}}$, $\mu_{\mathbf{P0}}$, $\mu_{\mathbf{Y}}$, $\mathbf{Q}$)

2: $\mathbf{M} = \mu_{\mathbf{P0}} \cdot \mu_{\mathbf{Y}}$

3: $\mathbf{D} = \mathbf{\Sigma_{px}} - \mathbf{M}$

4: $\mathbf{E} = \mathbf{D}^{\intercal}$

5: $\mathbf{A} = \mathbf{D} - \mathbf{E}$

6: $\mathbf{\Delta} = \begin{bmatrix} a_{2,3} & a_{3,1} & a_{1,2} \end{bmatrix}$

7: $trace_D = tr(\mathbf{D})$

8: $\mathbf{S} = \mathbf{D} + \mathbf{E}$

9: $s_{1,1} = s_{1,1} - trace_{\mathbf{D}}$

10: $s_{2,2} = s_{2,2} - trace_{\mathbf{D}}$

11: $s_{3,3} = s_{3,3} - trace_{\mathbf{D}}$

12: $\mathbf{Q} = \begin{bmatrix} trace_D & \Delta_1 & \Delta_2 & \Delta_3 \\ \Delta_1 & s_{1,1} & s_{1,2} & s_{1,3} \\ \Delta_2 & s_{2,1} & s_{2,2} & s_{2,3} \\ \Delta_3 & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$

13: **end function**

5. Find Eigenvalues and Eigenvectors and Calculate R and $\vec{t}$ In this supertask, **R** and $\vec{t}$ are calculated based on the eigenvalues and eigenvectors of **Q** and both point clouds' centers of mass. **Q** was calculated in the previous supertask, and the centers of mass were generated in supertask *2. Calculate Centers of Mass.* After finding the eigenvalues and eigenvectors of **Q** in supertask *5.1 Find Eigenvalues and Eigenvectors*, the eigenvector corresponding to the max eigenvalue is assigned to the

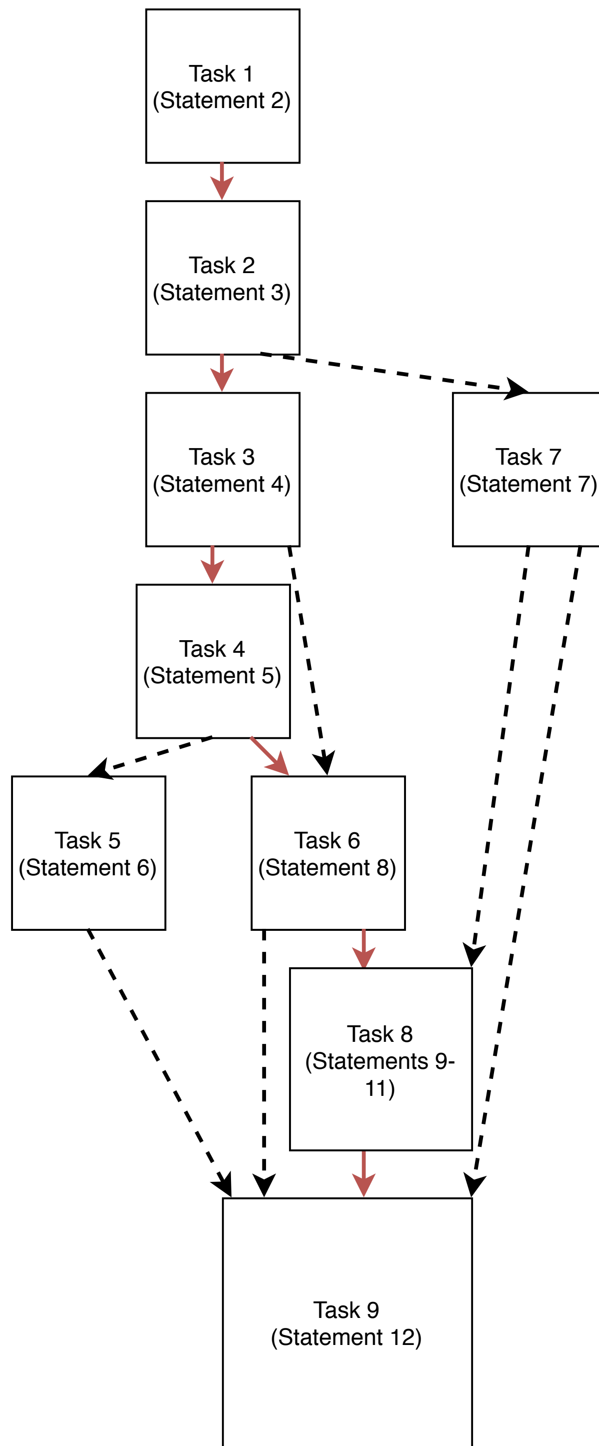

Figure 9: Task dependency graph of supertask *4. Calculate* $\mathbf{Q}$ with critical path in solid red outlined in Algorithm 4.

quaternion $\hat{q}$. From this quaternion, supertask *5.2 Calculate* $\mathbf{R}$ *and* $\vec{t}$ builds $\mathbf{R}$ and $\vec{t}$. Figure 10 shows the supertask dependency graph for the tasks within *5. Find Eigenvalues and Eigenvectors and Calculate* $\mathbf{R}$ *and* $\vec{t}$. Algorithm 5 shows the pseudo code for this supertask. The maximum degree of concurrency for this supertask is again 16.

Algorithm 5 Supertask: *5. Find Eigenvalues and Eigenvectors and Calculate* $\mathbf{R}$ *and* $\vec{t}$ pseudo code.

```
1: function CALC_RT(Q, μ_P0, μ_Y, R, t)
2:     W = eigenValues(Q), V = eigenVectors(Q)
3:     j = 0
4:     w_max = W[0]
5:     j_max = 0
6:     for each value ∈ W do
7:         if value > w_max then
8:             w_max = value
9:             j_max = j
10:        end if
11:        j = j + 1
12:    end for
13:    q = V[j_max]
14:    q̂ = normalize(q)
15:    R = rotationFromQuaternion(q̂)
16:    t = μ_Y − R · μ_P0
17: end function
```

6. Apply R and $\vec{t}$ This supertask applies the estimated rotation and translation to the sensed points. Algorithm 6 shows the pseudo code for this supertask. P_k is assigned with the points from P_0 with $\mathbf{R}$ and $\vec{t}$ applied. Figure 11 shows the task dependency graph for the tasks within *6. Apply* $\mathbf{R}$ *and* $\vec{t}$. Because this is a one-to-one mapping, the degree of concurrency for this supertask is the number of sensed points.

7. Calculate Error This is the final supertask that calculates the error between P_k and X. Algorithm 7 shows the pseudo code for this supertask. Figure

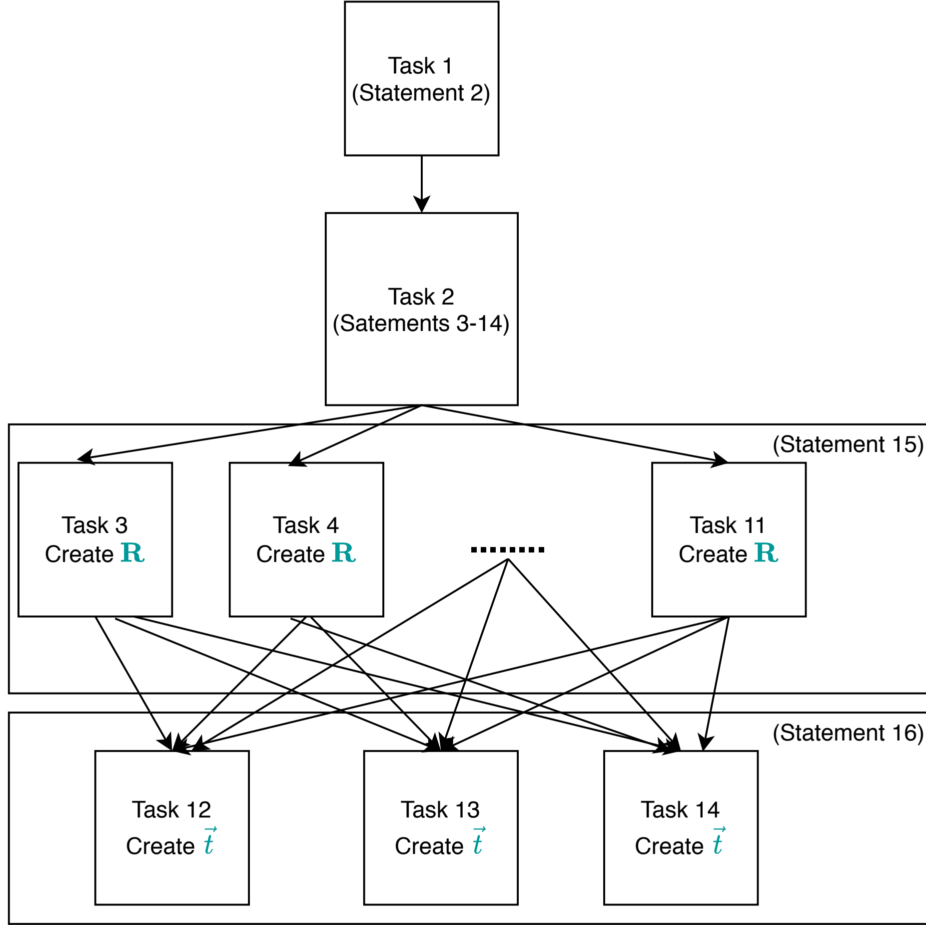

Figure 10: Task dependency graph of supertask *5. Find Eigenvalues and Eigenvectors and Calculate* $\mathbf{R}$ *and* $\vec{t}$ outlined in Algorithm 5.

Algorithm 6 Supertask: *6. Apply* $\mathbf{R}$ *and* $\vec{t}$ pseudo code.

1: **function** APPLY_RT(P_0, $\mathbf{R}$, $\vec{t}$, P_k)
2:     **for each** $\mathbf{p_0}, \mathbf{p_k} \in P_0, P_k$ **do**
3:         $\mathbf{p_k} = \mathbf{R} \cdot \mathbf{p_0} + \vec{t}$
4:     **end for**
5: **end function**

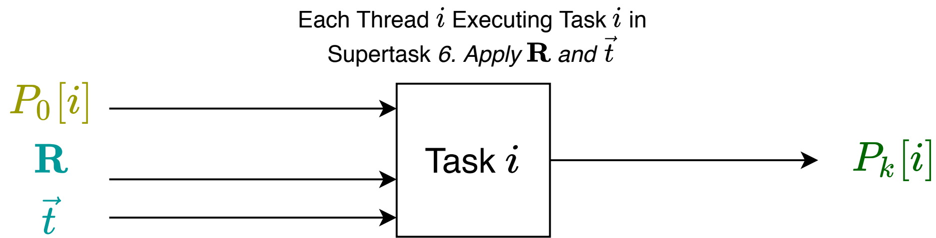

Figure 11: Task dependency graph of *6. Apply* $\mathbf{R}$ *and* $\vec{t}$ outlined in Algorithm 6.

12 shows the task dependency graph for the tasks within *7. Calculate Error*. The one-to-one mapping part of this supertask has a degree of concurrency of the number of sensed points. The average degree of concurrency of the binary tree reduction is again $\frac{2n-1}{log_2(n)+1}$ with n as the number of sensed points.

Algorithm 7 Supertask: *7. Calculate Error* pseudo code.

1: **function** CALC_ERROR(P_k, Y, $error_{rms}$)
2: $\quad error_{rms} = 0$
3: $\quad$ **for each** $\mathbf{p_k}, \mathbf{y} \in P_k, Y$ **do**
4: $\quad\quad error_{rms} = error_{rms} + |(\mathbf{p_k} - \mathbf{y})|^2$
5: $\quad$ **end for**
6: $\quad error_{rms} = \frac{error_{rms}}{|P_k|}$
7: **end function**

The ICP algorithm then iterates through this process until error drops lower than a set threshold. The user can set the error threshold to a desired value. The larger the error threshold, the faster the algorithm converges with a less accurate estimated pose. The smaller the error threshold, the longer the algorithm will take to converge, but will typically yield a more accurate pose.

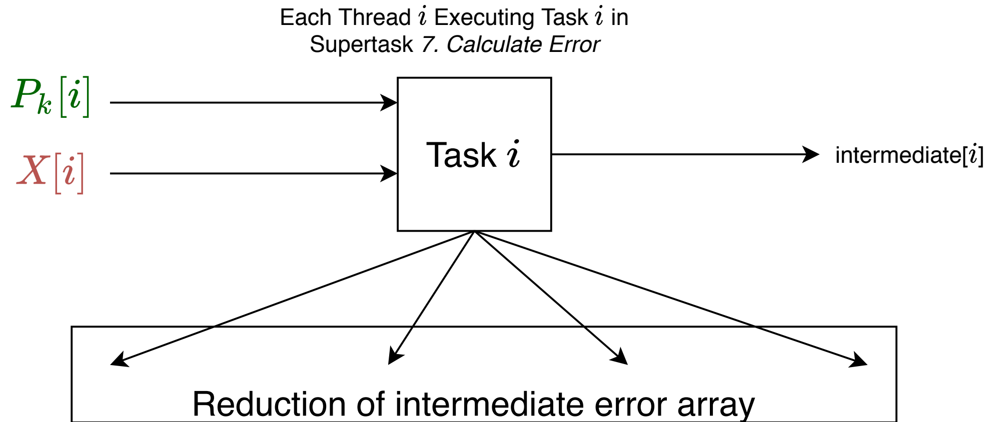

Figure 12: Task dependency graph of *7. Calculate Error* outlined in Algorithm 7.

3.3.2 Mapping

Each supertask is statically mapped with respect to mapping the ICP algorithm onto the GPU with CUDA. The ICP algorithm is inherently decomposed to reduce thread interaction and idleness inside supertasks.

1. Find Correspondences This supertask is statically mapped to a kernel [66] launching with n threads. Again, n is the number of sensed points. Each thread is completing a part of this task. Algorithm 1 and Figure 6 shows how each thread i calculates the associated nearest neighbor of $P_k[i]$ to X. A mapping based on a data partitioning works best for *1. Find Correspondences.* Because each thread reads X, X is partitioned in a block distribution to take advantage of shared memory and the locality of interaction. This mapping reduces the amount of time each thread spends reading X. Loading X into shared memory requires each thread to interact and synchronize after loading X from global to shared memory. This process reduces the contention on global memory banks and minimizes the size of global data exchange. Lastly, this organization completely eliminates interaction between threads and idleness within threads.

2. Calculate Centers of Mass This supertask is statically mapped to reduction kernel, launching with $\frac{n}{2}$ threads. This calculates a single centers of mass. This kernel is executed twice and the instances can be run in parallel to calculate the centers of mass of P_0 and Y. However, the center of mass of P_0 only needs to be run the first ($k = 0$th) iteration of the accelerated ICP algorithm. In this kernel, threads in each block reduce the data associated with that thread block to a block sum. Then, each block sum is summed and the final sum is divided by the number of elements of the array. This must take advantage of a block distribution and utilization of shared memory.

3. Calculate $\mathbf{\Sigma_{px}}$ This supertask is first statically mapped to a kernel, launched with n threads. Each thread calculates each dot product of the P_k and Y pair. This is then stored in 9 intermediate arrays. These intermediate arrays are summed in reduction kernels launched with $\frac{n}{2}$ threads, which can be run in parallel. These are then averaged into $\mathbf{\Sigma_{px}}$. This mapping substantially reduces interaction and idleness between threads in the supertask.

4. Calculate Q This supertask is statically mapped to a kernel which spawns 16 warps of 32 threads. To differentiate execution of statements, the SPMD is written in multiple branch statements. However, this introduces the problem of branch divergence amongst threads in the same warp [67]. To fix this problem, each statement that can be executed in parallel is assigned to a warp that executes independently with an independent instruction scheduler. Additionally, threads use shared memory to interact. In Algorithm 4 and Figure 9, statements 2 and 3 can be mapped to warp 1. Once those finish and synchronize, warp 1 can execute statements 4 and 5. Concurrently, warp 2 can execute statement 7. After both those end and synchronize, statement 6 can execute on warp 1 and statement 8 on warp 2. After those finish and synchronize, warp 1 can execute statements 9-11. Finally after those finish and synchronize, all 16 warps can execute statement 12. This supertask does not map very well but using shared memory for interaction reduces the communication overhead. Unavoidably, because a single warp contains 32 threads, each warp may have idle threads throughout the supertask.

5. Find Eigenvalues and Eigenvectors and Calculate R and $\vec{t}$ The eigenvalues and eigenvectors portion is statically mapped to a *cuSolver* [36] kernel launched with **Q**. After this, the rest of the supertask is statically mapped to a single kernel with 9 warps of 32 threads each. The quaternion, $\hat{q}$, is calculated by warps 0-3

and placed into shared memory. After synchronization, warps 0-8 assign **R**. Then after synchronization, warps 0-2 assign $\vec{t}$. This supertask also does not map well because some threads in the warps are idle. However, shared memory interaction is used for working threads.

6. Apply R and $\vec{t}$ This supertask is statically mapped to a kernel, launching with n threads. A mapping based on data partitioning works best for this supertask. Each thread i inside the kernel calculates its own $P_k[i]$ based on its **R**, $\vec{t}$, and $P_0[i]$. Besides memory bank contention, this mapping prohibits thread interaction and thread idleness.

7. Calculate Error The supertask is statically mapped to a kernel, launching with n threads. A mapping based on a data partitioning works best for this supertask. Additionally, a block distribution works for shared memory interactions between threads during the reduction phase of this supertask. Each thread i inside the kernel calculates an intermediate error based on $P_k[i]$ and $X[i]$. These intermediate errors are placed into shared memory and then summed in a reduction kernel. This mapping reduces interaction and idleness in threads.

3.3.2.1 Supertask Breakdowns

All tasks in a single ICP iteration are static in nature. It is known which tasks must be executed to complete each iteration. What is not known, however, is how many iterations must be executed to converge to the error threshold. Because error is not computed until the end of an iteration, the number of iterations executed in the ICP algorithm is dynamic in nature. The iterative part of the algorithm leads to dynamic task generation. The ICP algorithm has non-uniform tasks. *1. Find Correspondences* takes the most amount of time to complete and usually takes up

over 90% of the computational time [13].

A discussion follows about each task required in each supertask, because each supertask executes in sequence. In the ICP algorithm, supertask interactions are irregular[29], static[29], and one-way[29]. However, each supertask has threads that exhibit inter-task interactions that are regular[29].

1. Find Correspondences This supertask is split perfectly between each thread. Each thread is doing the same amount of work to find the nearest neighbor associated with their thread index. Because of this partitioning, this supertask is split into uniform tasks between the threads. This supertask does have a large data size associated with it. Each thread needs to access X, single member of P_k, and a write in Y.

In this supertask, the interaction is static and P_k and X are read-only. Y is read-write. Because threads do not interact besides memory bank conflicts, interactions are regular.

2. Calculate Centers of Mass This supertask is split into a multitude of non-uniform tasks. In the first part, the task of calculating both centers of mass could be mapped uniformly amongst threads. However, inside calculating each center of mass, the distribution of work is non-uniform amongst threads because it is an all-to-one reduction. The all-to-one reduction is mapped such that at the end of the reduction a single thread is reporting the sum and all the other threads are idle. The data size associated with this supertask is also large as it includes P_k and Y.

In this supertask, P_k and Y are read-only. The centers of mass' intermediate arrays reduction is read-write. Because this supertask includes all-to-one reductions, it inherently has threads interacting.

3. Calculate $\mathbf{\Sigma_{px}}$ In this supertask, the dot product is mapped to threads uniformly in a kernel, and the reduction can be mapped to threads non-uniformly. The data size associated with this supertask is also large as it includes P_0 and Y.

In this supertask, the interaction is static and P_0 and Y are read-only. The intermediate arrays are read-write. The reduction aspect of this supertask inherently has threads interacting.

4. Calculate Q This supertask can be mapped non-uniformly to threads in a kernel. There are many mini-tasks happening in this supertask that each require different amounts of time. The data size for this supertask is small as it only takes in $\mathbf{\Sigma_{px}}$ and the centers of mass and only outputs **Q**, a 4x4 matrix.

In this supertask, the interaction is static and $\mathbf{\Sigma_{px}}$ and the centers of mass are read-only. **Q** matrix is read-write. Interactions between threads are regular, but require finesse to mask with helpful computations. Interactions between threads should be timed so synchronization does not come at a high price.

5. Find Eigenvalues and Eigenvectors and Calculate R and $\vec{t}$ This supertask is a smaller task in the accelerated ICP algorithm. Its data size is small as it only takes in a 4x4 matrix and outputs a 3x3 matrix **R** and a 3x1 vector $\vec{t}$.

In this supertask, the interaction is static and **Q** read-only. **R** and $\vec{t}$ are read-write. Interactions between threads are regular, but require finesse to mask. Interactions between threads should be timed so synchronization does not come at a high price.

6. Apply R and $\vec{t}$ This supertask is mapped uniformly to threads in a kernel. Each thread i applies **R** and $\vec{t}$ to $P_0[i]$ and assigns it to $P_K[i]$. The data size for this supertask is small as each thread only takes in a 3x3 matrix and a 3x1 vector and outputs to a single member in P_k.

In this supertask, the interaction is static and P_0, $\mathbf{R}$, and $\vec{t}$ are read-only. P_K is read-write. Because threads do not interact besides memory bank conflicts, interactions are regular.

7. Calculate Error This supertask is mapped non-uniformly to threads in a kernel as the last part of finding error is an all-to-one reduction. The data size for this supertask is large as it takes in both P_k and X but only outputs a single value.

In this supertask, the interaction is static and P_k and X are read-only. The reported error is read-write. Interactions between threads are again regular. The reduction aspect of this supertask has the highest thread interaction.

3.3.2.2 Processing Models

The way the accelerated ICP algorithm is decomposed into tasks is very similar to the pipeline model [29]. However, this pipeline is not being preemptively filled until the error is found. For this reason, the accelerated ICP algorithm employs the manager-worker model [29]. Where the CPU is the manager and the GPU is the worker. After each iteration the CPU receives the error from the GPU and dynamically decides whether to run another iteration. Additional worker GPUs can be added to the model. Because the Nvidia Titan V GPU has 5120 CUDA cores, each CUDA core can be considered a worker. Like stated previously, in this research, only one GPU is considered. However, each step within the ICP algorithm exhibits its own model.

Supertasks *1. Find Correspondences* and *6. Apply* $\mathbf{R}$ *and* $\vec{t}$ exhibit the data-parallel model [29]. Figure 6 shows how each thread i is performing the same set of code to calculate the nearest neighbor of X for their respective $P_k[i]$. Figure 11 shows this same idea.

Supertask *2. Calculate Centers of Mass* exhibits a task graph model [29] and a

data-parallel model. Threads run in parallel based on data, but are also dependent on the reduction process between threads in shared memory space.

A task graph model describes supertasks: *3. Calculate* $\mathbf{\Sigma_{px}}$, *4. Calculate* $\mathbf{Q}$, *5.1 Find Eigenvalues and Eigenvectors*, and *5.2 Calculate* $\mathbf{R}$ *and* $\vec{t}$. These supertasks have many dependencies between threads. However, these supertasks allow for some data-level parallelism between threads.

Supertask *7. Calculate Error* exhibits the data-parallel model. Each thread i within the kernel is performing the code on its own $P_0[i]$. This supertask also exhibits a data-parallel model where each thread calculates the error associated with each point between X and P_k. However, the reduction part of this task is more of a task graph model where a multitude of threads are reducing down to a single thread at the end.

3.4 Optimization

3.4.1 Find Correspondences

The nearest neighbor kernel completing the supertask of *1. Find Correspondences* utilizes the privatization optimization. All threads inside each thread block need to read the same global memory many times. To avoid this, this global memory is replicated amongst all thread blocks through privatization. From Stratton [61]: "Privatization is the transformation of taking some data that was once common or shared among parallel tasks and duplicating it such that different parallel tasks have a private copy on which to operate." In this manner, global memory reads are reduced and transformed into shared memory reads. The timing hit from a shared memory access is less than a global memory access. This optimization reduces the runtime of the nearest neighbor kernel in the accelerated ICP algorithm implementation.

3.4.2 Reduction

The all-to-one reduction [29] kernel that uses a binary tree-type reduction is used 12 times throughout the accelerated ICP algorithm. For this reason, optimization of this kernel is of the utmost importance. The implementation of the reduction kernel is based on Ritcher's reduction kernel optimizations ideas [68].

First, each thread in the reduction kernel reads 2 elements from global memory, adds them together, and assigns them to shared memory in the thread block. Thread blocks then perform the remaining reduction in shared memory. This method reduces future memory access times in the reduction process. Threads out of bounds of the array assign a zero into shared memory. Additionally shared memory bank conflicts are reduced through special indexing. From Ritcher [68]: "Zero padding reduces thread divergence by allowing all the threads to participate in calculations. Shared memory indexing was implemented in a way to reduce shared memory bank conflicts and also reducing thread divergence." The final 32 threads of the reduction can reduce without memory bank conflicts by taking advantage of warp properties on the GPU. Ritcher [68] also explains unrolling any additional loops left over in the reduction to achieve maximum performance. This reduces the instruction count of the kernel by removing the loop check and loop iterator. Ritcher's ideas exemplify many CUDA optimization techniques that have been applied to the reduction kernel and other kernels in accelerated ICP algorithm.

3.5 Algorithm Analysis

3.5.1 Thread Communication

All-to-one reductions frequently appear in the design of this accelerated ICP. Reductions occur inside supertasks *2. Calculate Centers of Mass*, *3. Calculate* $\mathbf{\Sigma_{px}}$, and

7. Calculate Error. With respect to the GPU, because these communication patterns are implemented with threads communicating with each other, utilizing shared thread block memory presents opportunities for optimization. However, reduction is a challenging communication pattern with the bottleneck of a single thread reporting the result at the end of each supertask.

For this research, the assumption is made that every multiprocessor on the Titan V is involved in the all-to-one reduction. This research exploits several aspects of GPU architecture in the implementation of all-to-one reductions, including utilizing shared-memory amongst thread blocks, taking advantage of thread behavior within warps, and optimizing cache hits. From Grama, the cost for the all-to-one reduction is $T(p) = (t_s + t_w m)\log(p)$. T represents the time to run an all-to-one reduction with t_s as the startup time or overhead, t_w as the transfer time or bandwidth, m as the size of the message, and p as the number of processors or elements in the reduction [29]. This can be mapped to the Nvidia Titan V GPU. Again, the Nvidia Titan V has 5120 CUDA Cores and a memory bandwidth of 652.8 GB/s [62]. These can be substituted for a more accurate cost:
$T(5120) = (t_s + \frac{1}{652.8GB/s}m)\log(5120)$. To get a precisely accurate t_s, a profiling would need to be conducted. For this analysis, it is assumed that t_s is greater than GPU clock rate of 1200 MHz. This gives us our final equation of:
$T(5120) = (\frac{1}{1200MHz} + \frac{1}{652.8GB/s}m)\log(5120)$ for a lower bound execution runtime.

3.5.2 Interaction Overheads

In the supertask of *1. Find Correspondences*, an interaction overhead is the contention for X. Each thread may need all members of X. To reduce this overhead, each thread block can initially load a copy of X into shared memory. Then threads can pull from shared memory inside the thread block throughout the rest of the ker-

nel. This approach reduces the global reads of each X member to only the number of blocks. This method is also called privatization or the replication of data to take advantage of a "private copy" [61] amongst threads.

In the supertask of *2. Calculate Centers of Mass*, interaction overheads are the synchronization of threads during the reduction process as well as contention for the input array. The contention is reduced again replicating the input array into shared memory inside the thread blocks and also using memory accesses with striding. The synchronization overhead between threads can be reduced by taking advantage of warp synchronization amongst threads in the same warp.

In the supertask of *4. Calculate* $\mathbf{Q}$, interaction overheads are waiting for data-exchange and contention for data. The contention is reduced by loading thread data to shared memory. The data-exchange is masked with useful computations.

In the supertask of *5. Find Eigenvalues and Eigenvectors and Calculate* $\mathbf{R}$ *and* $\vec{t}$, interaction overheads are data contention and waiting for data-exchange. For instance, many threads need the max eigenvalue. Because of this, each thread calculates the max eigenvalue. This method replaces interactions with computations. Additionally, shared memory is used for the quaternion, eigenvalues, eigenvectors, and centers of mass. This technique reduces data contention in global memory banks.

For supertask *6. Apply* $\mathbf{R}$ *and* $\vec{t}$, an interaction overhead is the contention for P_0, $\mathbf{R}$, and $\vec{t}$. P_0, $\mathbf{R}$, and $\vec{t}$ are replicated amongst each thread block to reduce global memory bank contention.

In the supertask of *7. Calculate Error*, an interaction overhead is the contention for P_k and X during the reduction phase. This can be mitigated by reducing in shared memory and also using striding in memory accesses.

3.5.3 Isoefficiency Function

When building the isoefficiency function, it is assumed that $m \geq p$ and $n \geq p$. Here m is the number of truth points, n is the number of sensed points, and p is the number of processors. Because the accelerated ICP implementation and a serial ICP implementation run the same number of iterations, only a single ICP iteration is considered for the runtime for simplification purposes.

The parallel runtime of supertask *1. Find Correspondences* is

$$\frac{nlog_2(m)}{p} \tag{1}$$

The parallel runtime of the supertask *2. Calculate Centers of Mass* is

$$\frac{2n}{p}log_2(n) \tag{2}$$

The parallel runtime of supertask *3. Calculate* $\mathbf{\Sigma_{px}}$ is
$\frac{n}{p} + \frac{9n}{p}log_2(n)$. Assuming p is significantly greater than 16, the parallel runtime of the portion containing the supertasks *4. Calculate* $\mathbf{Q}$ and *5. Find Eigenvalues and Eigenvectors and Calculate* $\mathbf{R}$ *and* $\vec{t}$ is a constant C because the maximum degree of concurrency is 16. However, for the serial implementation, this needs to be a runtime of $16C$. The supertask of *6. Apply* $\mathbf{R}$ *and* $\vec{t}$ has a parallel runtime of $\frac{n}{p}$. Lastly, the supertask *7. Calculate Error* has a parallel runtime of $\frac{n}{p}log_2(n)$.

This leads to a parallel runtime of:
$\frac{nlog_2(m)}{p} + \frac{2n}{p} + \frac{12nlog_2(n)}{p} + C$ which also leads to a serial runtime of: $nlog_2(m) + 2n + 12nlog_2(n) + 16C$. However, an ideal serial algorithm does a single reduction in n. This makes our serial runtime now:
$nlog_2(m) + 2n + 12n + 16C$ or $nlog_2(m) + 14n + 16C$.

For the following analysis, it is assumed the runtime C adds is negligible compared

to $nlog_2(m)$, n, and $log_2(n)$ weighted terms. From Grama [29], the expected speedup by analysis is the serial runtime divided by the parallel runtime. With $C = 0$, this gives a speedup of:

$$\frac{p(nlog_2(m) + 14n)}{nlog_2(m) + 2n + 12nlog_2(n)} \quad (3)$$

Using Grama's isoefficiency equation of $\frac{speedup}{p}$ [29], this gives an isoefficiency function of $\frac{(nlog_2(m)+14n)}{nlog_2(m)+2n+12nlog_2(n)}$. Assuming like-weighted terms scale about the same with increased point sizes, there is an extra $nlog_2(n)$ term in the denominator. This term is a remnant of the serial reduction being more efficient than the parallel in a reduction.

3.5.4 Scalability

From the speedup function previously stated in Eq. (3), an increased number of sensed and truth points allows for more processors to be used. Consequently, larger point clouds generally achieve a larger speedup. Additionally, increasing the number of processors past the number of sensed points produces diminishing performance results. The reason for this effect is the maximum degree of concurrency mapped is equal to the number of sensed points, as found in many supertasks like *1. Find Correspondences.*

The accelerated ICP algorithm scales well with the number of processors as long as processors is less than the number of sensed and truth points. The number of sensed points has the most effect on the runtime and speedup of the algorithm. Increasing both the number of processors and the number of points achieves a justified speedup. For instance, a setup with 5k processors and 30k truth and sensed points is expected to achieve a speedup of 739.02x.

3.6 Delaunay Traversal

Now, the novel Delaunay nearest neighbor algorithms will be presented. These Delaunay algorithms and variations are implemented and used in the accelerated ICP algorithm in *1. Find Correspondences.*

3.6.1 Delaunay Creation

As originally discussed in [13], When registering an object with a known model, the Delaunay triangulation of points is pre-calculated offline. This step is done utilizing the Open3D library [47] discussed in subsection 2.4.1. In order to prepare the Delaunay graph for the nearest neighbor traversal, the edges of each tetrahedron are extracted and grouped according to the starting point. Thus, each edge is represented twice. Although this factor increases required memory space, the graph becomes bidirectional allowing for an arbitrary start-point.

3.6.2 Notation

As stated previously, given a point set $X = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ of n points of m dimensions, the nearest neighbor of a point $\mathbf{p} \in P$ is

$$NN(\mathbf{p}, X) = argmin_{\mathbf{x} \in X} |\mathbf{p} - \mathbf{x}_i|_2. \tag{4}$$

Because multiple points within X may be equidistant to $\mathbf{p}$, the result of the nearest neighbor function might not be unique.

A Delaunay triangulation of a point set is a graph $G = (1..k, E)$ consisting of edges from each point $\mathbf{x}_i$ to the Delaunay neighbors $\mathbf{x}_{j \neq i}$. To find the nearest neighbor of a source point within the target point set utilizing the Delaunay triangulation, the algorithm can start with an arbitrary start point, $\mathbf{x}_0$.

To store the full Delaunay triangulation, for each point $\mathbf{x}_i$ in the data set there is a corresponding list of values:

$$\vec{e}_{i,j} = \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|} \tag{5}$$

$$m_{i,j} = \frac{1}{2}|\mathbf{x}_j - \mathbf{x}_i| \tag{6}$$

$$coref = j \tag{7}$$

In equation 5, is the Delaunay edge from point $\mathbf{x}_i$ to one of its Delaunay neighbors $\mathbf{x}_j$ which has been normalized to a unit vector. Equation 6 gives $m_{i,j}$ as half the distance, or the midpoint, between $\mathbf{x}_i$ and $\mathbf{x}_j$. The final item in equation 7 is the reference index of where point $\mathbf{x}_j$ is stored in the data set.

3.6.3 Traversal

To find the nearest neighbor of a source point $\mathbf{p}$ within point set X utilizing the Delaunay graph, a series of traversals, or *walks* are taken from a point $\mathbf{x}_0$ to point $\mathbf{x}$, where $\mathbf{x}$ satisfies equation 4. The first step in the algorithm is calculating

$$\vec{u} = \mathbf{p} - \mathbf{x}_i \tag{8}$$

where $\vec{u}$ is the vector from $\mathbf{x}_i$ to $\mathbf{p}$. Next, the dot product

$$c = \vec{u} \cdot \vec{e}_{i,j} \tag{9}$$

between $\vec{u}$ and $\vec{e}_{i,j}$ is determined. Since $\vec{e}_{i,j}$ is a unit vector, this dot product is the scalar component of $\vec{u}$ in the direction of $\vec{e}_{i,j}$. The dot product c can then be compared to the midpoint $m_{i,j}$ from equation 6. If the dot product is greater than $m_{i,j}$, $\mathbf{p}$ is closer to $\mathbf{x}_j$ than $\mathbf{x}_i$. Since the algorithm is seeking the point closest to $\mathbf{p}$, the point $\mathbf{x}_j$ corresponding to the largest c is the next node in the graph to visit.

Thus, $\mathbf{x}_i$ is replaced by point $\mathbf{x}_j$, and the algorithm continues. However, if no c is greater than $m_{i,j}$, the algorithm stops and returns the current point $\mathbf{x}_i$ as the nearest neighbor. The pseudocode for the Delaunay traversal is shown in algorithm 8.

Algorithm 8 Delaunay Traversal Pseudocode

Require: $X :=$ Model Point Cloud
Require: $p :=$ Source Point
Require: $s_i :=$ Index to begin search
Require: $d_{edges} :=$ List of Delaunay edges
Require: $d_{mid} :=$ List of edge midpoints

```
 1: function DelaunayTraversal(X, p, s_i, d_edges, d_mid)
 2:     prev_max ← s_i
 3:     curr_max ← s_i
 4:     repeat
 5:         u⃗ ← p − x_i
 6:         foundNN ← TRUE
 7:         c_max ← 0.0
 8:         for each e⃗, m ∈ d_edges[prev_max], d_mid[prev_max] do
 9:             c ← DotProduct(u⃗, e⃗)
10:             if c > m then
11:                 foundNN ← FALSE
12:                 if c > c_max then
13:                     c_max ← c
14:                     curr_max ← e⃗.index()
15:                 end if
16:             end if
17:         end for
18:         prev_max ← curr_max
19:     until foundNN == TRUE||prev_max == curr_max
20:     return X[prev_max]
21: end function
```

3.6.4 Space and Time Complexity

For each vertex in a Delaunay graph, there are on average $\sqrt{n}$ edges [69]. Thus, the space complexity for the Delaunay triangulation of a dataset is on the order of $\mathcal{O}(n\sqrt{n})$. In terms of time complexity to find the nearest neighbor of point $\mathbf{p}$ utilizing the Delaunay traversal algorithm, since each vertex connects to on average $\sqrt{n}$ other

vertices, and all but one of those are rejected at each step of the traversal, by the time $\sqrt{n}$ vertices have been visited, $\sqrt{n}(\sqrt{n}-1) = n - \sqrt{n}$ vertices have been rejected, meaning the entire dataset has been visited. Thus, in the worst case, the algorithm will need to visit $\sqrt{n}$ vertices. However, because with each step of the traversal, the algorithm advances toward the neighbor and away from farther vertices, this case will rarely occur. In fact, there are heuristics, presented in section 3.7 that can greatly reduce the number of traversals required.

3.7 Delaunay Walk Variations

In this subsection, a plethora of *Delaunay Walk* variations and implementations are explored. The difference in these variations is the starting node or point to begin the Delaunay walk. In this thesis, the closest nearest neighbor is found, which intends $k = 1$ in *kNN*. However, the same *Delaunay Walk* variations presented can be used in applications with expanded k.

3.7.1 Zero Delaunay Walk

In the *Zero Delaunay Walk* variation, the walk is started from a the same node each iteration. This starts the walk immediately and removes a search for a best start node. The static starting node can be either the node closest to the center of mass, the most traveled node, or an arbitrary node.

3.7.2 KD Approximate Delaunay Walk

In the *KD Approximate (KD-ANN) Delaunay Walk* variation, the walk is started from the KD-ANN nearest neighbor node [70]. This node is found by conducting a depth first search of the KD Tree without back tracing. This approximate nearest neighbor returned is usually a very close neighbor, but in many cases it can return the

exact nearest neighbor. By starting the walk from a very close neighbor, the number of nodes in the walk are decreased substantially. Since this algorithm monotonically converges, a close starting point can significantly shorten the *Delaunay Walk*.

3.7.3 Previous Nearest Neighbor Delaunay Walk

In the *Previous Nearest Neighbor (PNN) Delaunay Walk* variation, the walk is started from the previous ICP iteration's nearest neighbor. The transformation of the fitted point cloud happens slowly over many iterations. In this manner, the previous nearest neighbor is extremely close to the current nearest neighbor. Because the previous nearest neighbor was already calculated, it does not add any computational complexity to the algorithm. Instead, it leverages the previous ICP iteration's work to make the current iteration more efficient.

3.7.4 PNN Optimized Delaunay Walk

In the *PNN Optimized Delaunay Walk* variation, the *KD-ANN Delaunay Walk* is used the first iteration and *PNN Delaunay Walk* is used in succeeding iterations. By combining aspects of these variations, this walk variation is the most efficient and has the highest performance.

3.8 ICP Filter

A GPU ICP filter was designed to aid with accuracy and filter out outlier sensed points. While finding the root mean square (RMS) error in ICP, it also finds the standard deviation of the error. It then is able to filter out outlier sensed points from the ICP solution based on the point's error and the number of standard deviations away from the average error. The user is able to set which iteration the filter starts and how many standard deviations to filter. Once this filter is turned on, it decides

which sensed points to use every iteration in ICP. It is possible for a point to be filtered out in an early iteration and rejoin the solution, if its error decreases bellow the filter threshold.

3.9 Experimental Design

In this first set of experiments, it shows this research against the current state of AAR. A CPU ICP implementation is tested against the accelerated ICP GPU implementation. Each implementation is tested for a plethora of number of different truth point and sensed point sets. However, the sets are identical for the tests between the CPU and GPU implementation respectively.

The object being sensed is the Aircraft A model. Figure 13 shows this object. The data sets used are different fidelities of this object ranging from 5053 to 40424 truth points. Figure 14 shows a 5053 point truth model example of the Aircraft A model. The specific fidelity used is closest to the number of sensed points. However, it also ensured the number of truth points are greater than or equal to the number of sensed points. The sensed points are generated by the OpenCV C++ library [71] from images of two calibrated virtual stereo cameras through stereo block matching. Figure 15 shows the left and right images of the stereo cameras. Figure 16 shows accelerated ICP running on the yellow sensed points generated from stereo block matching on these images.

Nvidia's profiling tool *nvprof* [72] is used to measure kernel timings on the GPU. Additionally, the C++ timing library of `std::chrono` [73] is used to time each implementation's overall elapsed real-time, also known as wall-clock time. This wall-clock time is reported as an average of all the runs for the particular point sets. Each ICP implementation runs continuously for 100 times until convergence with an error threshold of $1E-6$. It is noted, the implementations have identical output but dif-

Figure 13: Virtual Aircraft A sensed object

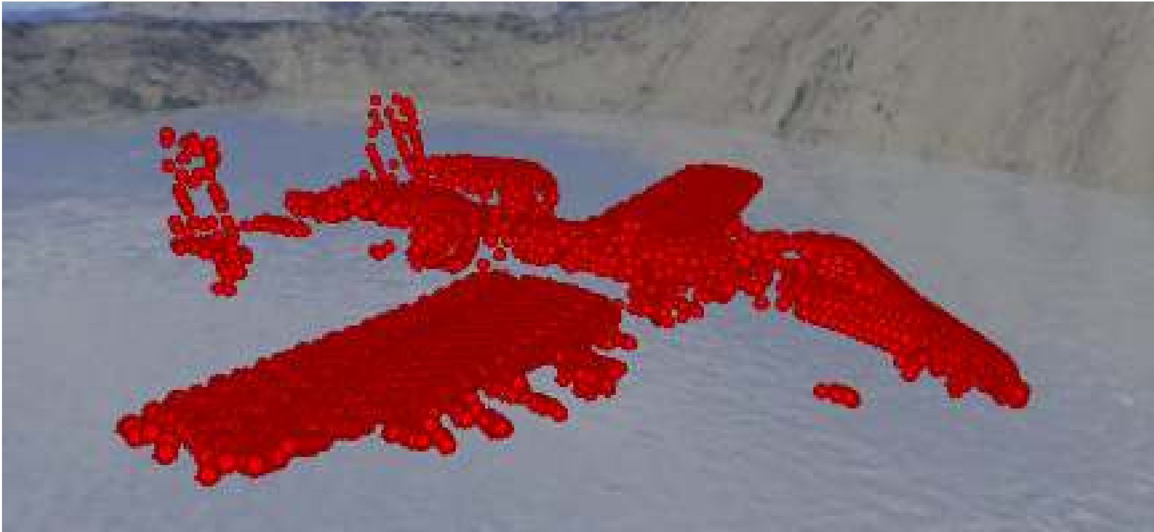

Figure 14: Virtual Aircraft A red 5053 point truth model based on sensed object

Figure 15: Left and right virtual stereo cameras images showing sensed object

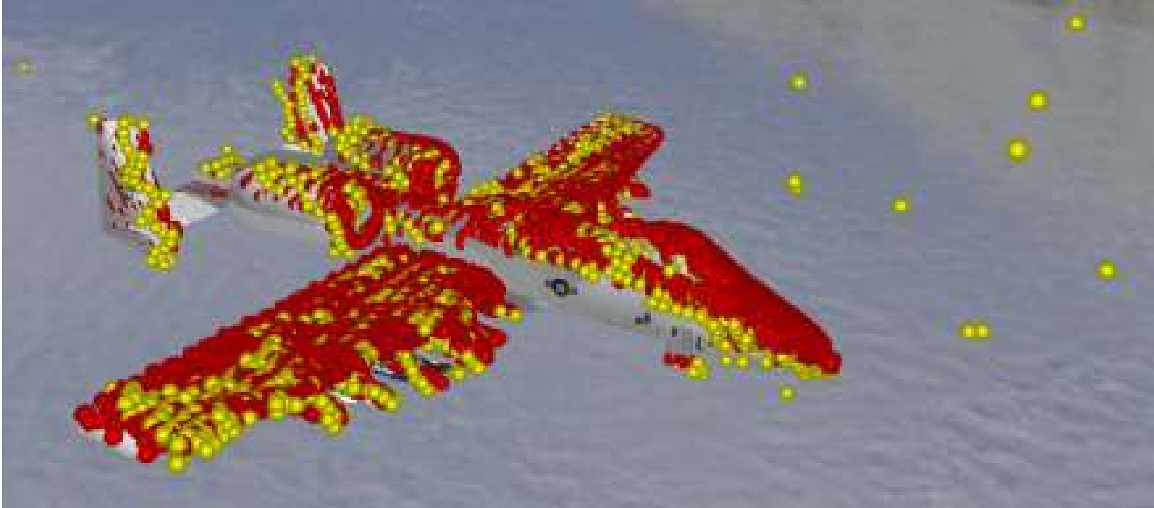

Figure 16: ICP running on yellow sensed points and red truth model points

fer in speed. The average ICP wall-clock time is reported. The profiler *nvprof* also reports the percent portion each kernel took in the overall accelerated ICP runtime. This output can be used to gain knowledge on how differing the point totals has on the GPU kernel's runtimes.

3.9.1 3D Models

In this experiment, every nearest neighbor matching algorithm was implemented in ICP with five different models with two different fidelities. The nearest neighbor algorithms compared are: *Brute Force*, *KD Tree*, *Zero Delaunay*, *KD-ANN Delaunay*, *PNN Delaunay*, and *PNN Optimized Delaunay*. The models are: a teapot as seen in Figure 17, an Aircraft A, an Aircraft B, a sports car as seen in Figure 18, and a dragon as seen in Figure 19. The experiments are conducted on both the CPU and GPU. The walk frequency of the Delaunay walk variations will be explored to explain efficiency.

ICP is executed with a maximum iteration count of 100 and an error of $1E-11$. The sensed points are placed on the models and the models go through a series of

rotations ranging from -20 to 20 degrees in roll, pitch, and yaw. This deterministically guarantees each algorithm receives the same exact experiments. Lastly, noise is added to the sensed points, as seen in Figure 20. The noise added is random from a normal distribution with a standard deviation ranging from 10^{-4} to $10^{-}2$.

3.9.2 Virtual and Real Stereo Block Matching

In this section, the most efficient nearest neighbor algorithms, CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay*, found in section 3.9.1 are used in ICP to register a model to points generated from *opencv* stereo block matching [74]. Two image sources are utilized: virtual and real. For the virtual cameras, the parameters such as resolution (4096×3000) and field of view (28.7 degrees) were set to match the real cameras as close as possible.

The Aircraft B model was used in an approach towards the cameras. The approach was captured with truth data recorded from a real-world Vicon 3D motion capture system [11]. Thus, the accuracy of ICP is able to be tested against the truth from the motion capture. Additionally, recordings of motion capture sessions are able to be replayed within a virtual 3D environment, and the accuracy of stereo block matching on real images can be verified against virtual images with the same sensed objects in a controlled environment. To mimic a real-time AAR scenario, ICP is executed with a maximum iteration count of 30 and an error of $1E-8$. Lastly, a GPU ICP filter is tested for increased accuracy starting on iteration 15 filtering out points further than 15 standard deviations. Iteration 15 was chosen because it's the halfway mark to iteration 30. By this iteration, ICP has converged substantially close to the final transformation and outlier sensed points are clear.

Figure 21 shows the virtual images in the lower left and right and the real images in the upper left and right, the sensed points in yellow, and the truth points in red

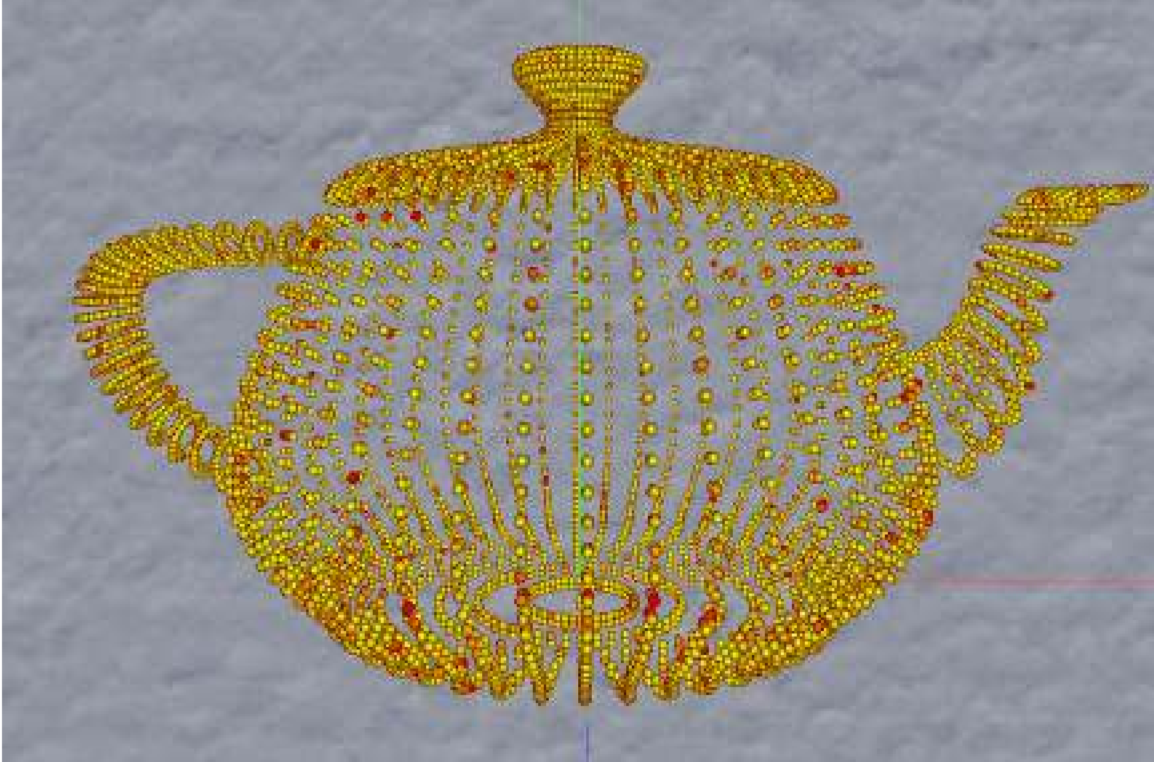

Figure 17: This shows ICP running on the teapot model point cloud with 8k points.

being fitted. The sensed points are displayed in yellow on top of the model of the aircraft, and the truth points appear in red. In the left and middle of Figure 21 depicts the location of the cameras in the real environment, with the red tubes depicting the view direction.

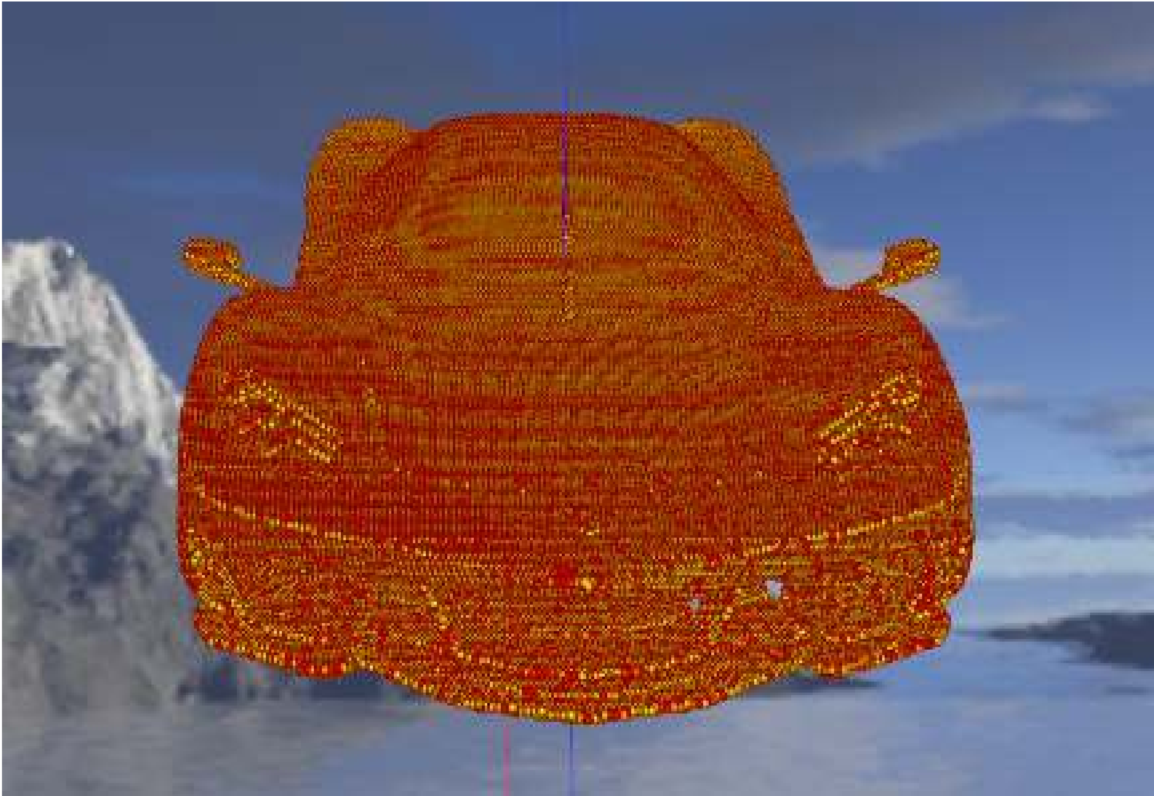

Figure 18: This shows ICP running on the sports car model with 27k points.

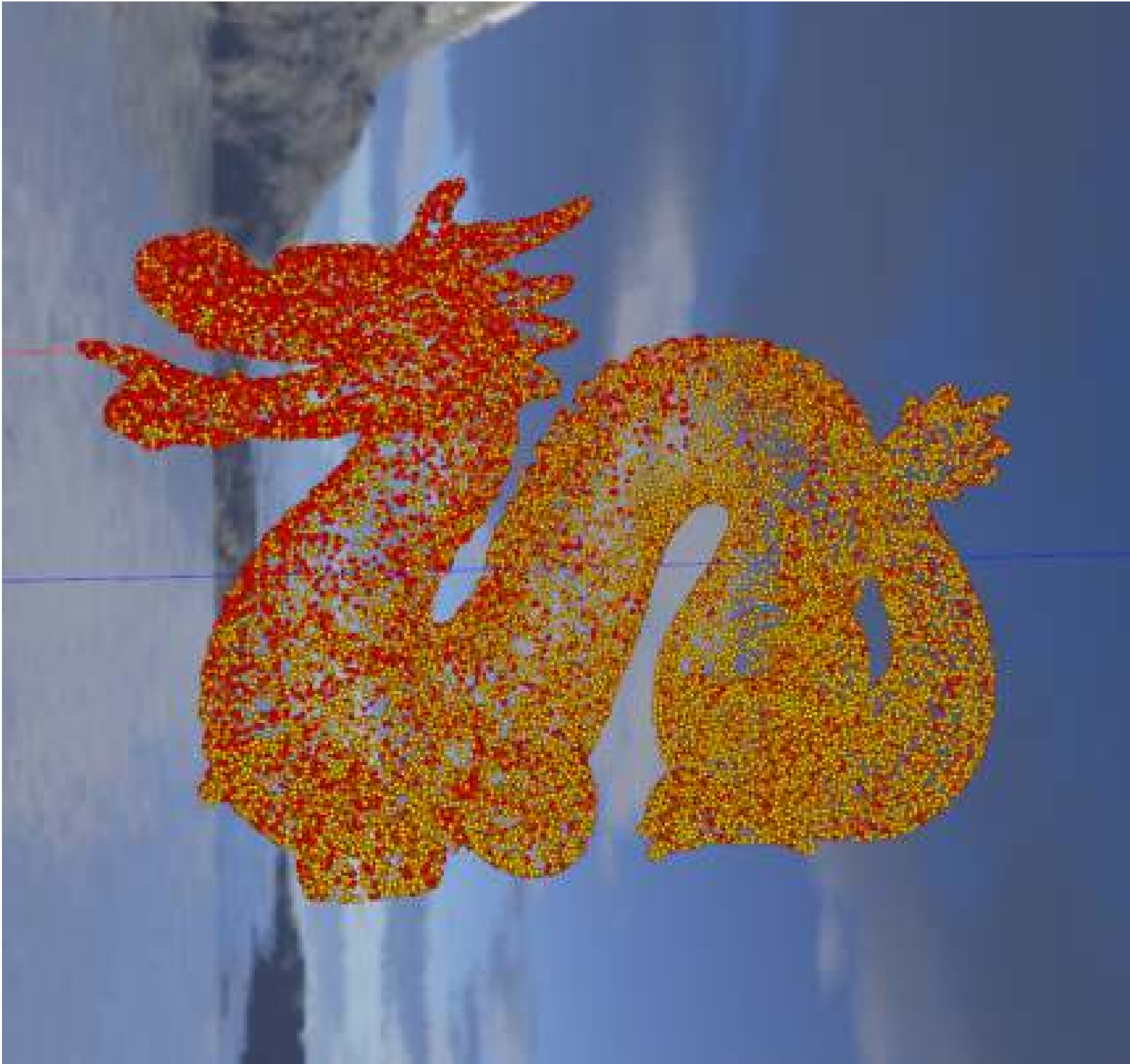

Figure 19: This shows ICP running on the dragon model with 31k points.

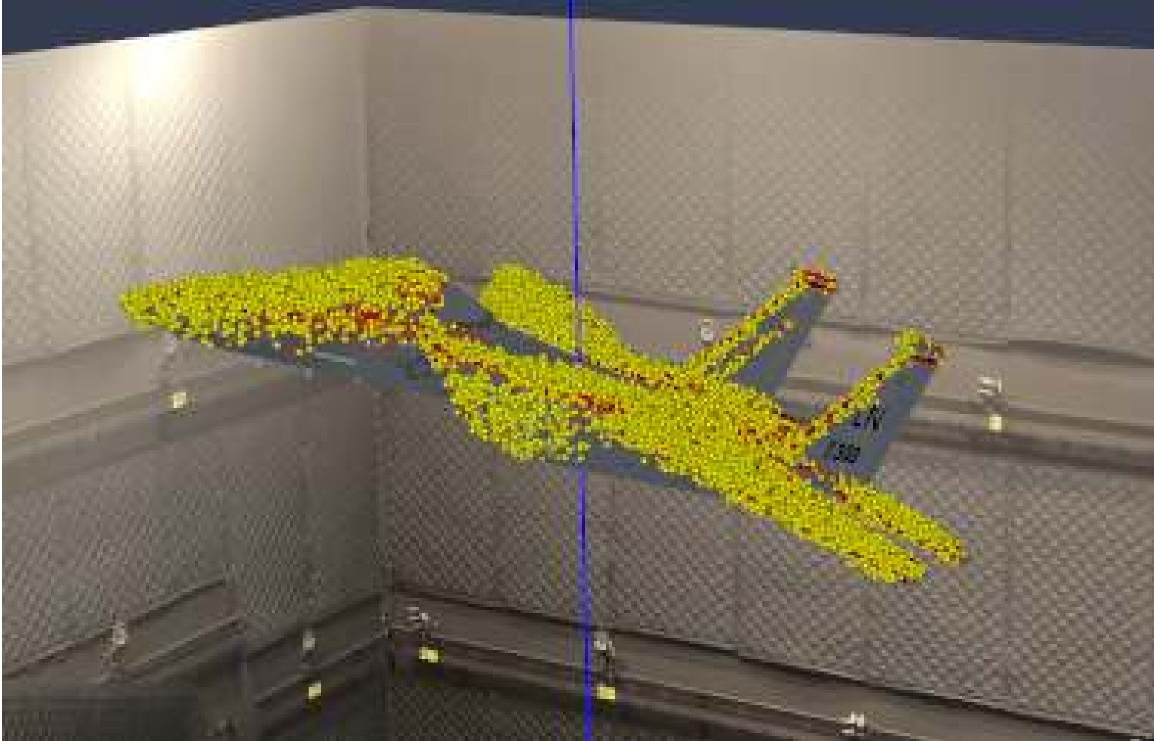

Figure 20: This shows the Aircraft B(21k) model with noise added to the yellow sensed points. The truth model in red is being fitted to the yellow points with ICP.

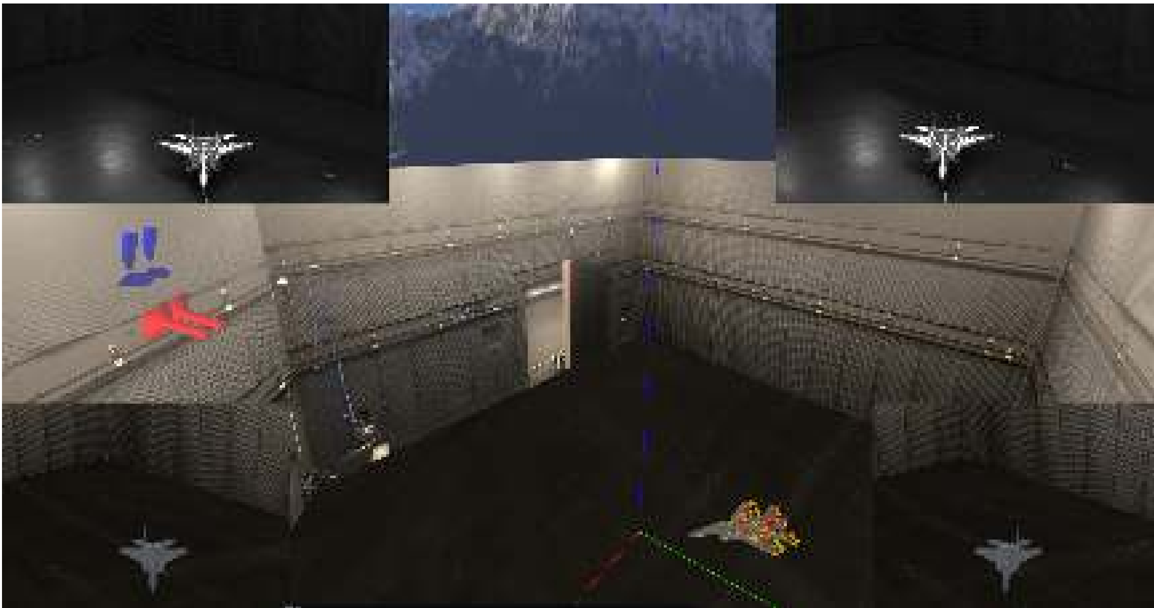

Figure 21: This shows ICP executed with real and virtual and real stereo block matching. ICP is fitting the red truth model to the sensed points. The real images are in the upper right and left. The virtual images are in the lower right and left. The model being fitted is Aircraft B(21k).

IV. Results and Analysis

4.1 Preamble

In Section 4.2 the first set of experiments are presented using the Nvidia Titan V graphics processing unit (GPU) and the Intel Xeon central processing unit (CPU). In Sections 4.3 and 4.4 the second set of experiments are presented using the Nvidia RTX 3080 GPU and the AMD Ryzen Threadripper 3970X 32-Core processor CPU.

4.2 Performance

As originally discussed in [12], Figure 22 shows an Automated Aerial Refueling (AAR) scenario in which the receiving aircraft is an Aircraft A and it is approaching the tanker. The two virtual images in the lower left and right are from the virtual cameras on the tanker. Stereo block matching is executed on these images and re-projected to produce the yellow sensed points. Then Iterative Closest Point (ICP) is executed to fit the red truth model to the yellow points. In return, the position and orientation (pose) of Aircraft A relative to the tanker is produced.

Table 2 shows the runtimes of the CPU and GPU ICP implementations while varying the number of truth and sensed points. The GPU implementation's runtime starts at an average of 32ms with 5053 truth and 1915 sensed points. The GPU's average runtime ends up at 42ms with 40424 truth and 30696 sensed points. Comparing these results with the CPU implementation, the CPU's average runtimes are 72ms and 4074ms for the same respective point clouds.

Figure 23 shows a graph of the GPU vs CPU implementation runtimes. As seen in Table 2 and Figure 23, as the number of points increases, the GPU vastly outperforms the CPU. The GPU's runtime is stable, while the CPU's runtime is increasing at a greater rate. Interestingly, with lower point totals, the CPU and GPU runtimes

are close. This similarity can be attributed to the processor the CPU implementation is using has a faster clock rate than the GPU processors. Additionally, the GPU has more overheads with memory transfers, kernel invocations, and thread synchronizations. However, these hardware differences and overheads become negligible with larger point totals.

Table 3 shows the GPU speedup over the CPU with respect to the number of truth and sensed points. The GPU only starts with a 2.25x speedup but ends with a 97.00x speedup. It should be noted that increasing the point totals past 40424 truth points becomes unreasonable for the CPU implementation runtime. A higher speedup may be achieved with higher point totals and more optimizations to the nearest neighbor and reduction kernels. Figure 24 shows a graph of the speedups with respect to the number of sensed points.

Table 1 shows the portion of runtime the Compute Unified Device Architecture (CUDA) kernels took as well as the runtime of those kernels with respect to the number of points. The nearest neighbor kernel starts with 42.47% and ends with 47.26% of the total time. The reduction kernel starts with 37.76% and ends with 34.61% of the total time. As shown here, the nearest neighbor kernel is taking the largest portion of the GPU runtime. This portion is closely followed by the reduction kernel. An interesting observation is as the truth and sensed point sizes become equivalent, the portion the nearest neighbor kernel takes is less than when the point differentials are greater. This decrease can be attributed to increased memory contention for truth points during the matching process. Lastly, it can be seen that future optimizations to the accelerated ICP implementation should be focused on the nearest neighbor and reduction kernels.

Table 4 shows the nearest neighbor kernel's real versus theoretical runtimes. Figure 25 shows the graph of this data. The theoretical runtime was calculated from Eq.

(1) and divided by the Titan V's clock rate of 1200 MHz. Table 5 shows the reduction kernel's real versus theoretical runtimes. Figure 26 shows the graph of this data. The theoretical runtime was calculated from Eq. (2) and divided by the Titan V's clock rate of 1200 MHz and by 2 to account for a single reduction. For both kernels, the overhead was calculated from real runtime minus theoretical runtime. Then, the minimum overhead was used to shift the real runtimes to account for overhead.

When accounting for overhead in the nearest neighbor kernel, the trend of real runtimes closely matches the trend of the theoretical runtimes. However, the first few points do not match this trend. The reason for this could be because the Titan V's 5120 cores were not saturated until 5120 or more points. The data point with 5053 sensed points is very close to core saturation. Additionally, non-deterministic instruction and warp scheduling could factor into this anomaly. Lastly, the theoretical runtime serves as a lower bound for real runtime.

In the reduction kernel, the trend of real runtimes closely matches the trend theoretical runtimes when accounting for overhead. The first few data points do not follow the theoretical trend as much as the data points with higher point totals. The first few data points from the nearest neighbor kernel show a similar anomaly. The same reasons for this anomaly hold true here. Higher point totals verify the theoretical trend lower bound of real runtime.

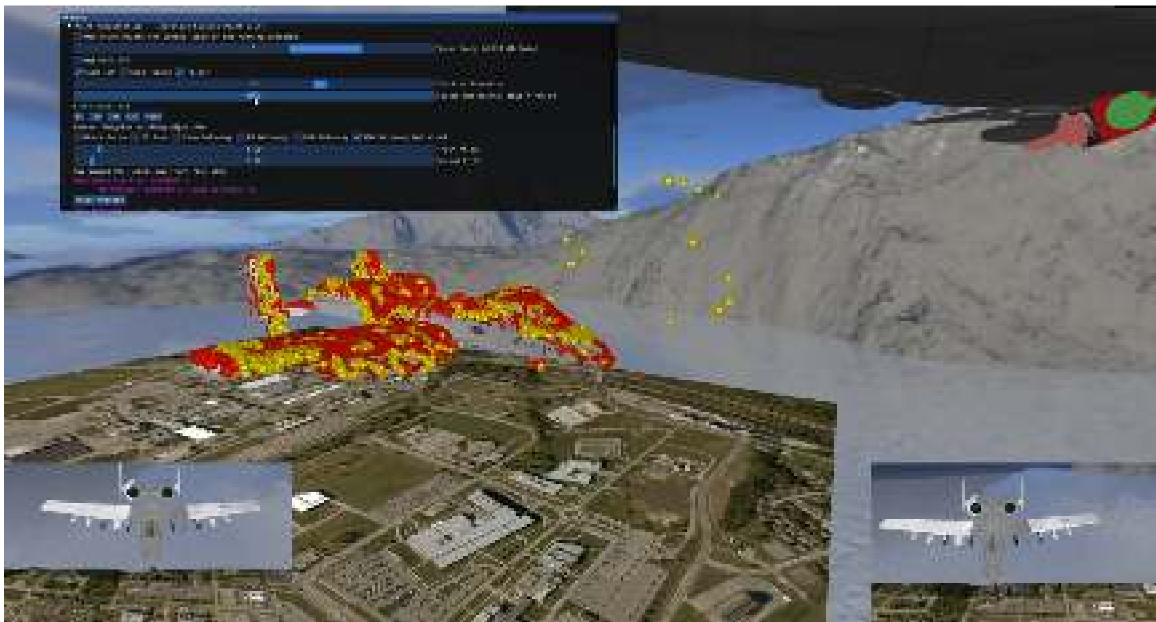

Figure 22: This shows an AAR scenario in which the receiving aircraft is Aircraft A and it is approaching the tanker. The two virtual images in the lower left and right are from the virtual cameras on the tanker. Stereo block matching is being run on these images to produce the yellow sensed points. Then ICP is executed to fit the red truth model to the yellow points. In return, the pose of Aircraft A relative to the tanker is produced.

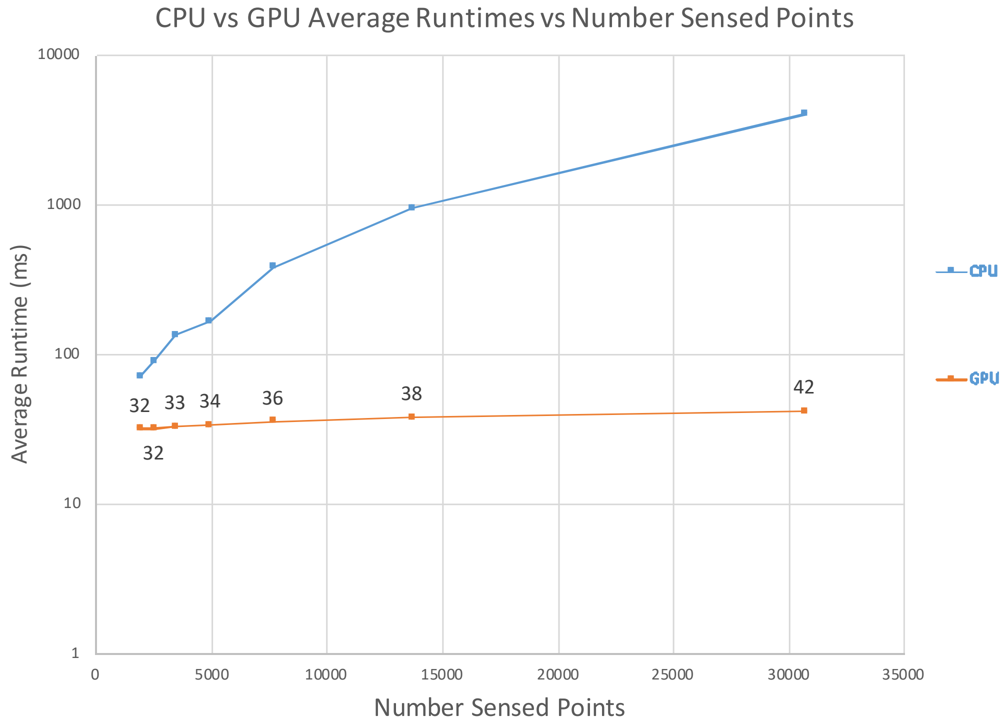

Figure 23: Graph of CPU versus GPU ICP overall runtimes versus number of sensed points.

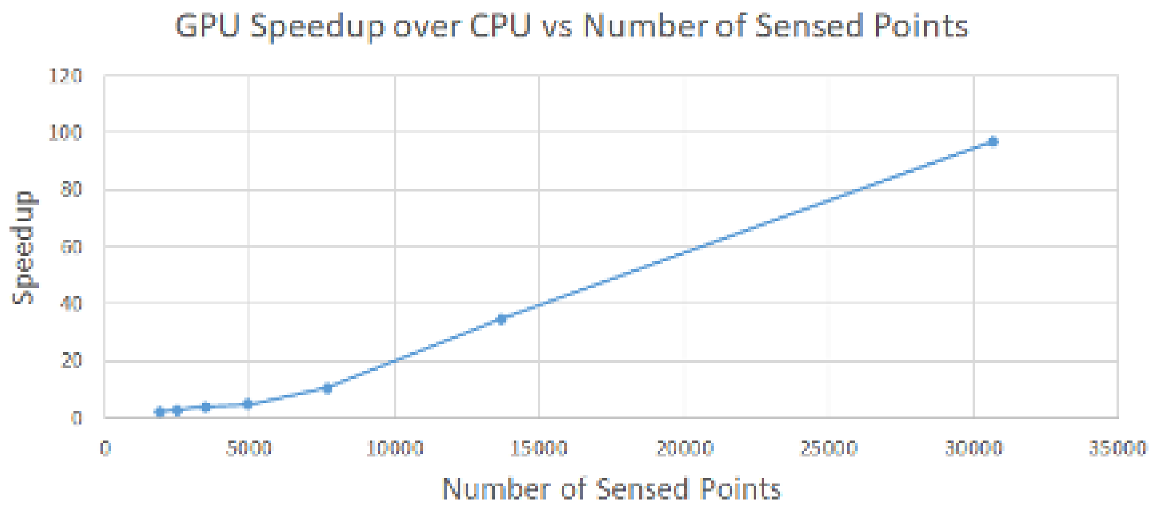

Figure 24: Graph of GPU speedup over CPU versus number of sensed points.

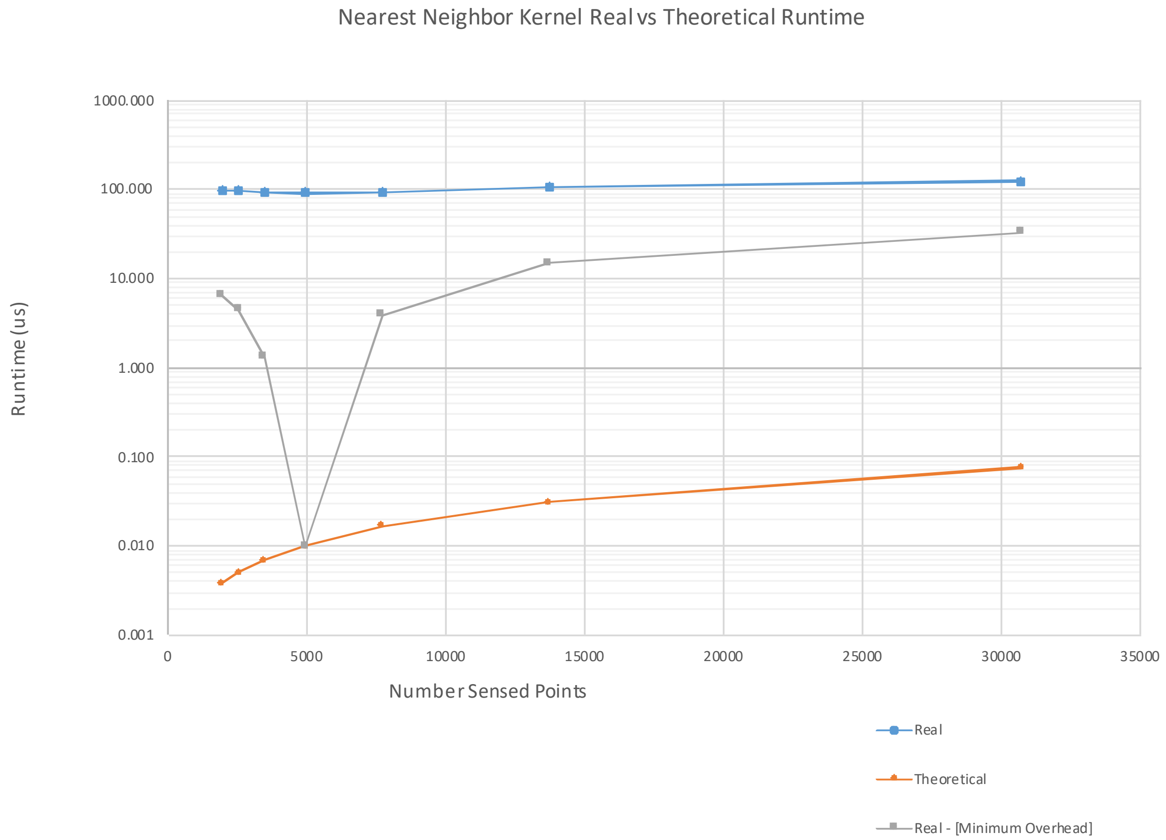

Figure 25: Graph of nearest neighbor kernel real and theoretical runtimes versus number of sensed points.

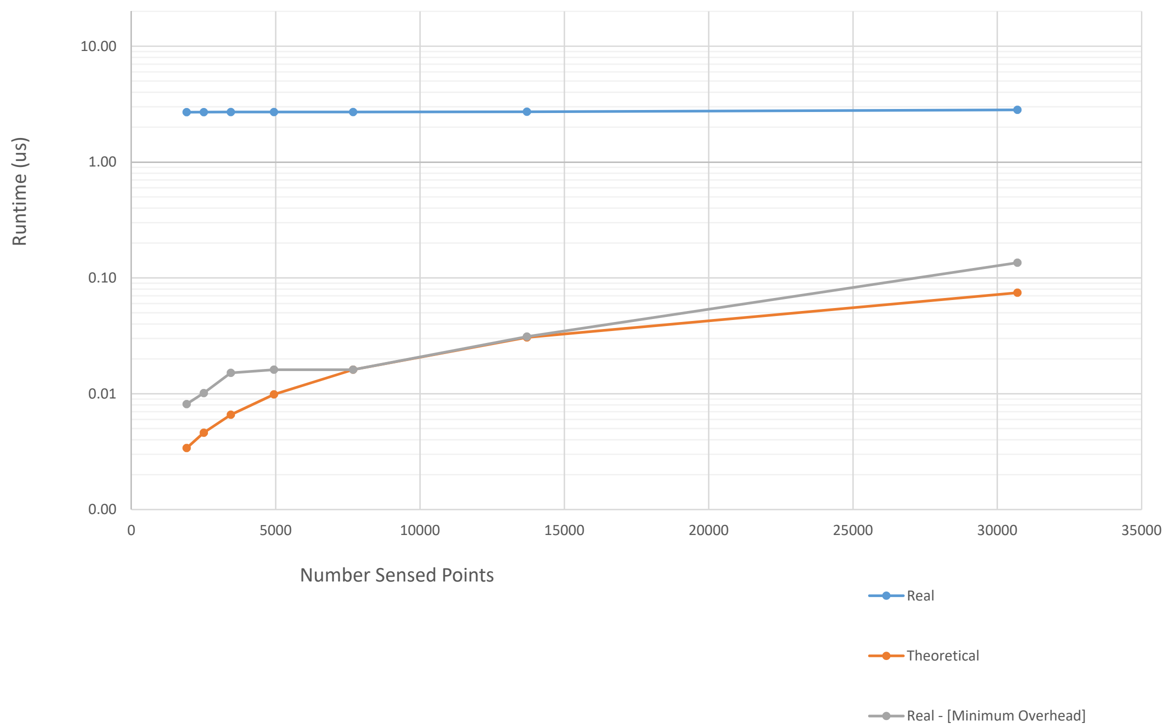

Figure 26: Graph of reduction kernel real and theoretical runtimes versus number of sensed points.

Table 1: Kernel portion of runtime and kernel runtimes reported by *nvprof*.

Number Truth Points (m)	Number Sensed Points (n)	Nearest Neighbor Portion	Nearest Neighbor Average Run Time (us)	Reduction Portion	Reduction Average Runtime (us)
5053	**1915**	42.47%	97.064	37.76%	2.696
5053	**2507**	42.08%	95.139	38.19%	2.698
5053	**3445**	41.28%	91.938	38.85%	2.703
5053	**4941**	40.74%	90.630	38.90%	2.704
10106	**7681**	41.95%	94.530	38.41%	2.704
15159	**13703**	44.34%	105.560	36.56%	2.719
40424	**30696**	47.26%	123.370	34.61%	2.823

Table 2: CPU and GPU runtimes and number of points.

Number Truth Points (m)	Number Sensed Points (n)	Average CPU Runtime (ms)	Average Titan V Runtime (ms)
5053	**1915**	72	32
5053	**2507**	91	32
5053	**3445**	136	33
5053	**4941**	168	34
10106	**7681**	386	36
15159	**13703**	947	38
40424	**30696**	4074	42

Table 3: GPU speedup over CPU and number of points.

Number Truth Points (m)	Number Sensed Points (n)	Speedup
5053	**1915**	2.25x
5053	**2507**	2.84x
5053	**3445**	4.12x
5053	**4941**	4.94x
10106	**7681**	10.72x
15159	**13703**	34.92x
40424	**30696**	97.00x

Table 4: Nearest neighbor kernel real and theoretical runtimes versus number of sensed points.

Number Truth Points (m)	Number Sensed Points (n)	Real (us)	Theoretical (us)	Overhead (us)	Real - [Minimum Overhead] (us)
5053	**1915**	97.06400	0.00383	97.06017	6.44389
5053	**2507**	95.13900	0.00502	95.13398	4.51889
5053	**3445**	91.93800	0.00690	91.93110	1.31789
5053	**4941**	90.63000	0.00989	90.62011	0.00989
10106	**7681**	94.53000	0.01663	94.51337	3.90989
15159	**13703**	105.56000	0.03097	105.52903	14.93989
40424	**30696**	123.37000	0.07645	123.29355	32.74989

Table 5: Reduction kernel real and theoretical runtimes versus number of sensed points.

Number Sensed Points (n)	Real (us)	Theoretical (us)	Overhead (us)	Real - [Minimum Overhead] (us)
1915	2.69600	0.00340	2.69260	0.00814
2507	2.69800	0.00461	2.69339	0.01014
3445	2.70300	0.00659	2.69641	0.01514
4941	2.70400	0.00987	2.69413	0.01614
7681	2.70400	0.01614	2.68786	0.01614
13703	2.71900	0.03065	2.68835	0.03114
30696	2.82300	0.07447	2.74853	0.13514

4.3 3D Models

As originally discussed in [13], Figure 27 shows the time taken per nearest neighbor algorithm and model executed on the CPU. This shows the *Previous Nearest Neighbor (PNN) Delaunay* nearest neighbor variation executed the fastest on all the models on the CPU. Figure 28 shows the time taken per nearest neighbor algorithm and model executed on the GPU. This shows the *PNN Optimized Delaunay* nearest neighbor variation executed the fastest on average on all the models on the GPU. The experiments show the most efficient algorithms are the *PNN Delaunay* on the CPU and *PNN Optimized Delaunay* on the GPU. For this reason, these algorithms will be compared against each other to show the hardware acceleration from using a GPU over a CPU. Because *PNN Delaunay* nearest neighbor leverages the prior iteration's nearest neighbor it makes these nearest neighbor variations performed best. Hardware differences like cache size could explain the difference between *PNN Delaunay* and *PNN Optimized Delaunay* on the CPU and GPU respectively.

Figure 29 shows a runtime comparison of the CPU *PNN Delaunay* algorithm and the GPU *PNN Optimized Delaunay* nearest neighbor algorithms. This shows the GPU runtime is about an order of magnitude faster than the CPU. Figure 30 shows the GPU speedup over the CPU for these algorithms. It is seen the GPU achieves just under a 140x speedup on the *Dragon(62k)* model. Figure 33 shows the overall total ICP runtime and Figure 34 shows the speedup. The GPU achieves about a whole order of magnitude in runtime and 25x speedup when comparing the most efficient implementations.

Figure 31 shows the teapot(8k) model can by the CPU *PNN Delaunay* algorithm. Figure 32 shows the teapot(8k) model can by the GPU *PNN Optimized Delaunay* algorithm. Both of these examples show how the *PNN Delaunay* variants leverage the prior iteration's nearest neighbor. These algorithms increase in efficiency as iterations

deepen. This can be seen by the negative slope.

With respect to the number of traversals required to determine the nearest neighbor, Figs. 39, 40, 41, and 42 depict the mean frequency for the four traversal variations on two sizes of two different models. As seen in these graphs, the *Zero Delaunay* method has the largest variance and highest mean. When utilizing an approximate neighbor as the starting point, the mean and variance decrease significantly. Finally, when using the previous result of the neighbor search, the mean is close to two walks with little variance meaning there are few searches that take more than two traversals of the graph. Additionally, it can be seen that regardless of the number of points in the model, the number of traversals for each algorithm changes very little. Although the number of points are doubled from the Teapot 4k to 8k, the mean number of traversals for the *Zero Delaunay* only increases from 8.89 to 11.99.

Again, Figs. 39, 40, 41, and 42 show *PNN Delaunay* and *PNN Optimized Delaunay* have the lowest average walks. PNN Optimized Delaunay has a lower average walk distance and variance than *PNN Delaunay* because the first iteration executes *KD Approximate (KD-ANN) Delaunay* to find a preferable starting location. Walk variance is important when executing on the GPU. If a single thread has a long walk distance, the whole thread block and possibly kernel will wait for the slowest thread to finish executing. This could also explain why the PNN Optimized Delaunay performs best on the GPU.

Figure 35 shows the time breakdown of the ICP algorithm when using CPU *PNN Delaunay* nearest neighbor on the Aircraft A(5k) model. Figure 36 shows the time breakdown of the ICP algorithm when using GPU *PNN Optimized Delaunay* nearest neighbor on the Aircraft A(5k) model. This shows the nearest neighbor portion of the algorithm on the CPU takes 92.12% of the total ICP time. In contrast, the nearest neighbor portion only takes 11.75% of the total time. It can be seen the hardware

acceleration from the GPU is in the nearest neighbor portion of ICP.

Figure 37 shows the time breakdown of the ICP algorithm when using CPU *PNN Delaunay* nearest neighbor on the Aircraft A(63k) model. Figure 38 shows the time breakdown of the ICP algorithm when using GPU *PNN Optimized Delaunay* nearest neighbor on the Aircraft A(63k) model. These figures show as the number of points increase, every portion of ICP is benefitting from the hardware acceleration of the GPU.

In these experiments, each nearest neighbor algorithm variant returned identical results. The nearest algorithms are all logically equivalent but vary in speed. For this reason the resulting ICP rotations and translations returned are identical. These experiments executed with little to no error.

Figure 44 shows the ICP total runtime with adding noise, using virtual, and using real images. This shows a dirty sensed point cloud does not negatively affect the runtime. This proves the robustness of the ICP and nearest neighbor algorithms.

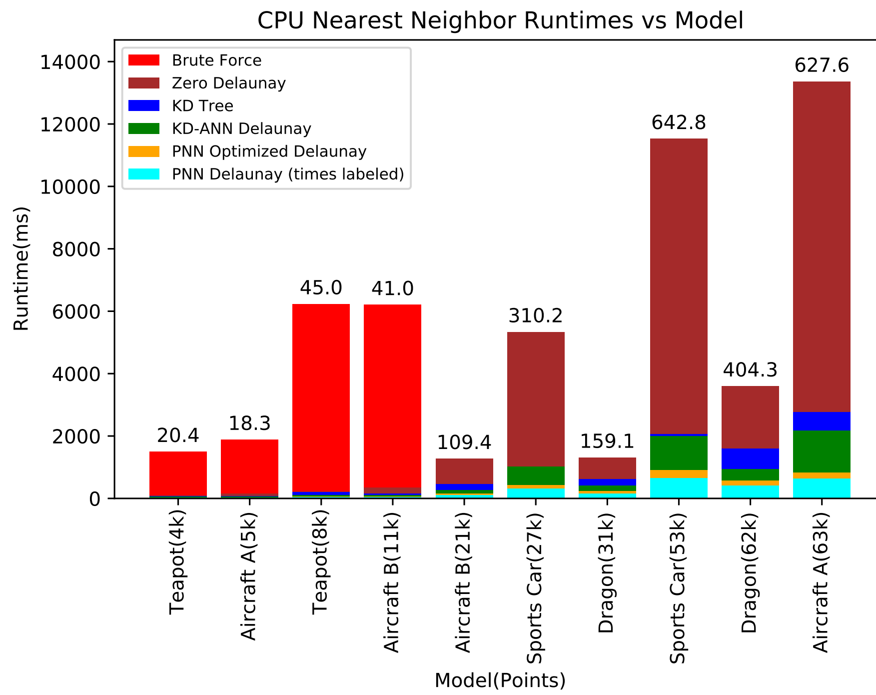

Figure 27: CPU Nearest Neighbor Algorithm Runtime vs Model

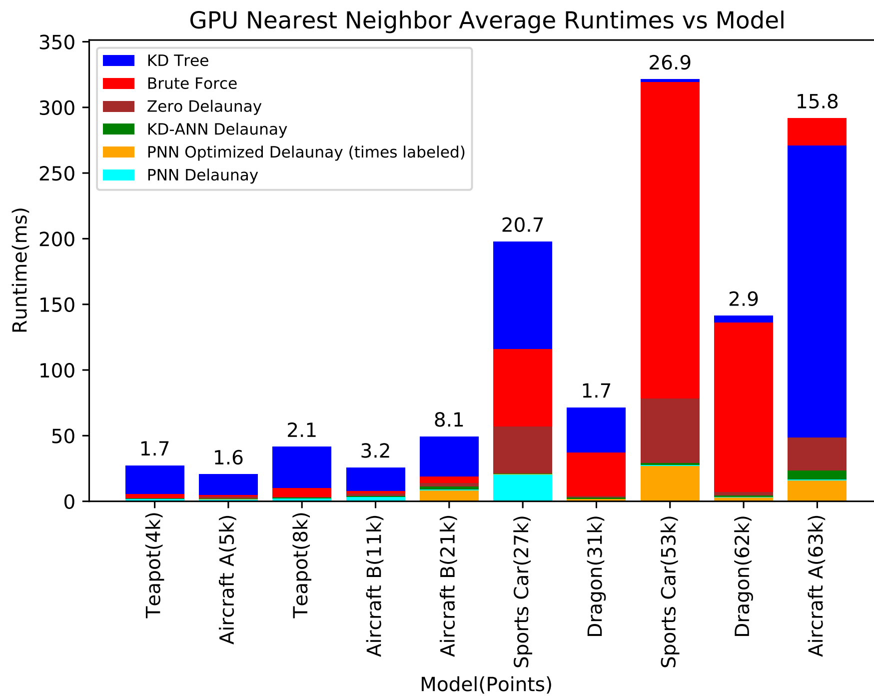

Figure 28: GPU Nearest Neighbor Algorithm Runtime vs Model

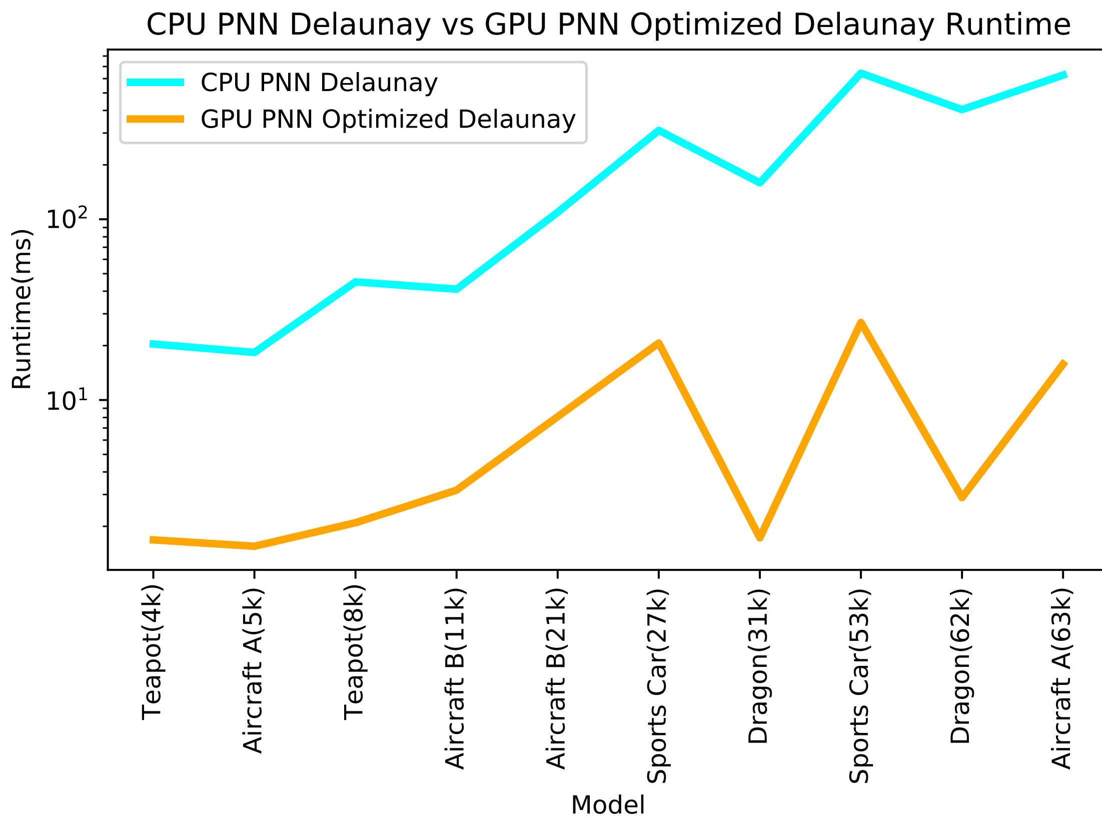

Figure 29: ICP Runtime CPU *PNN Delaunay* vs GPU *PNN Optimized Delaunay*

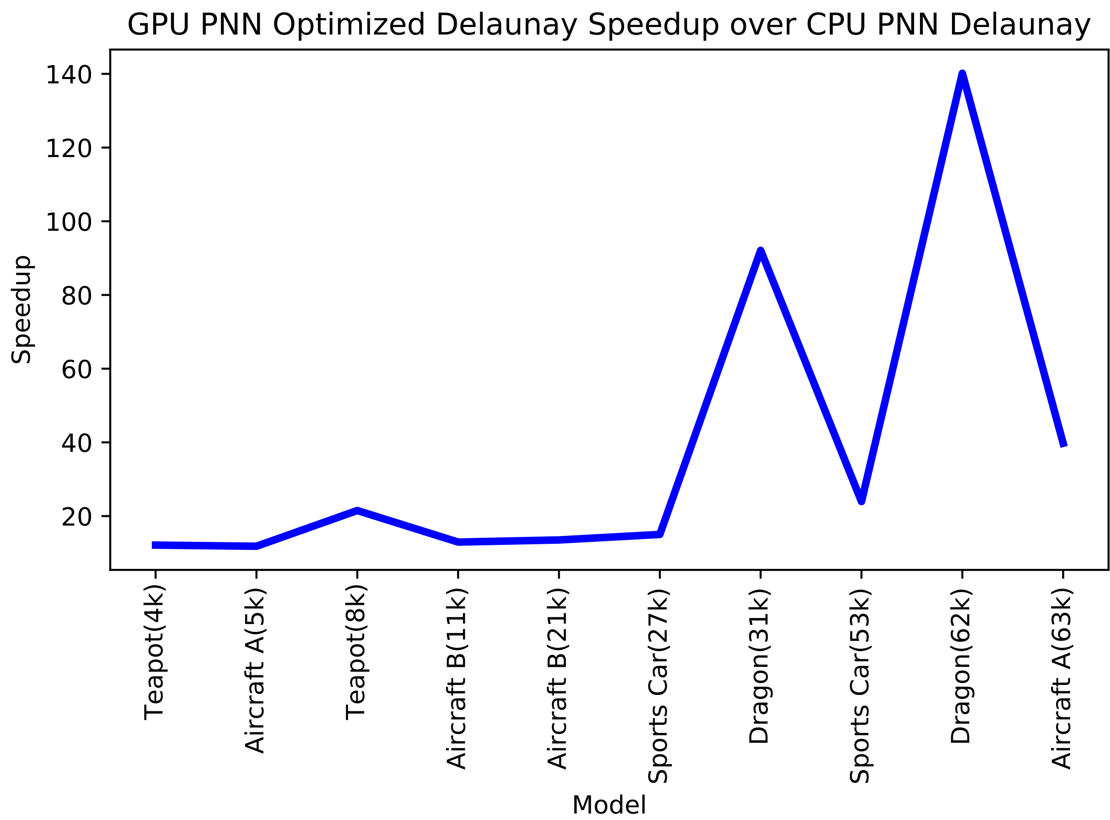

Figure 30: ICP GPU *PNN Optimized Delaunay* Speedup over CPU *PNN Delaunay*

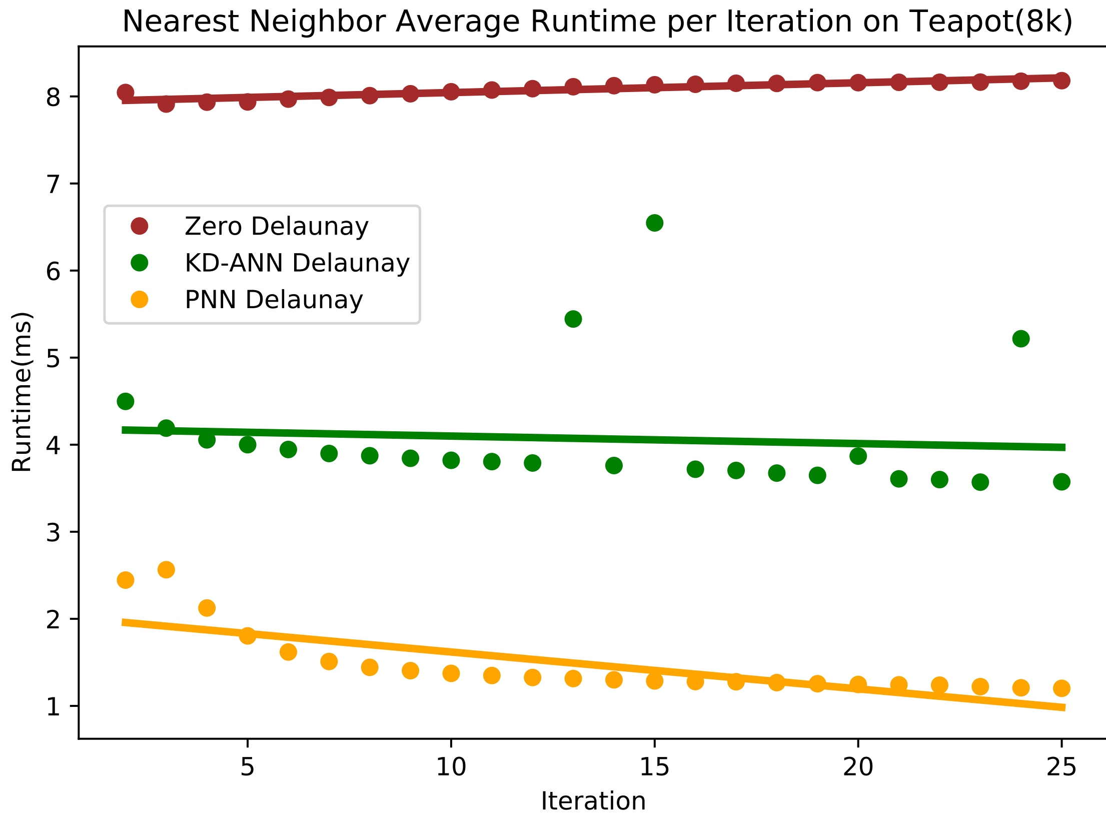

Figure 31: This graph shows an nearest neighbor algorithmic comparison of the runtime per iteration on the CPU running ICP on the Teapot(8k) model.

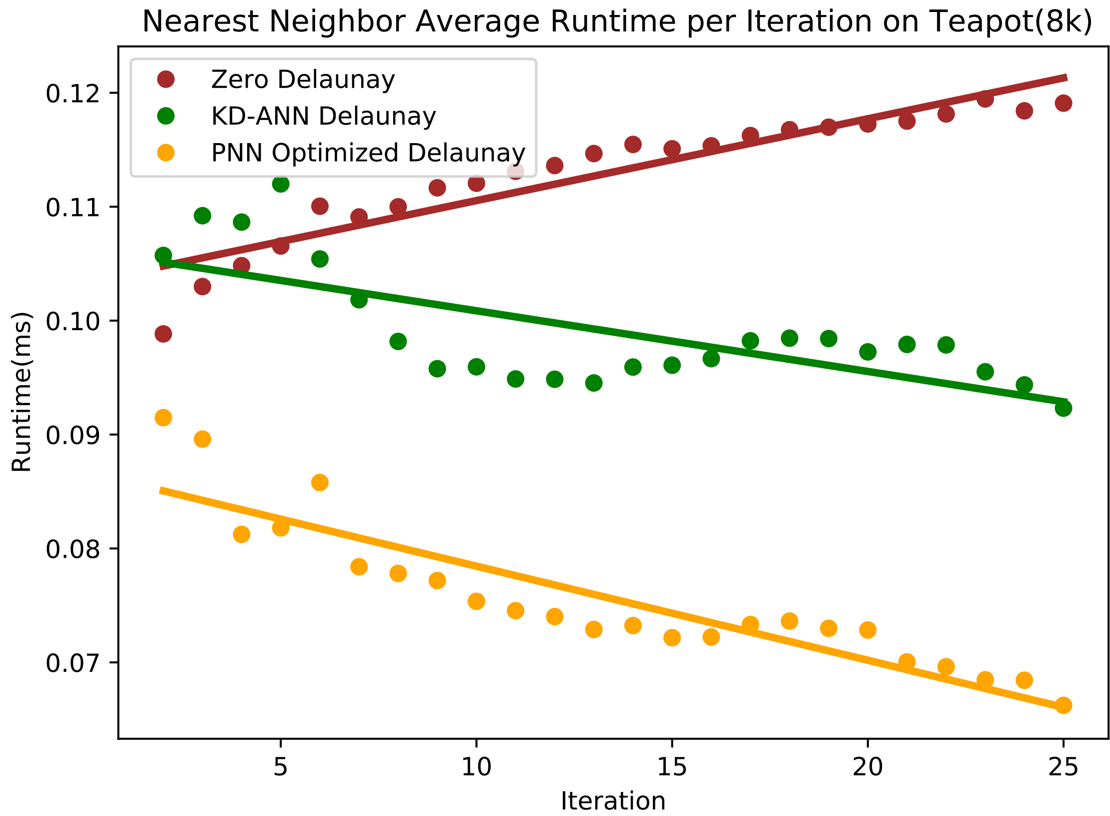

Figure 32: This graph shows an nearest neighbor algorithmic comparison of the runtime per iteration on the GPU running ICP on the Teapot(8k) model.

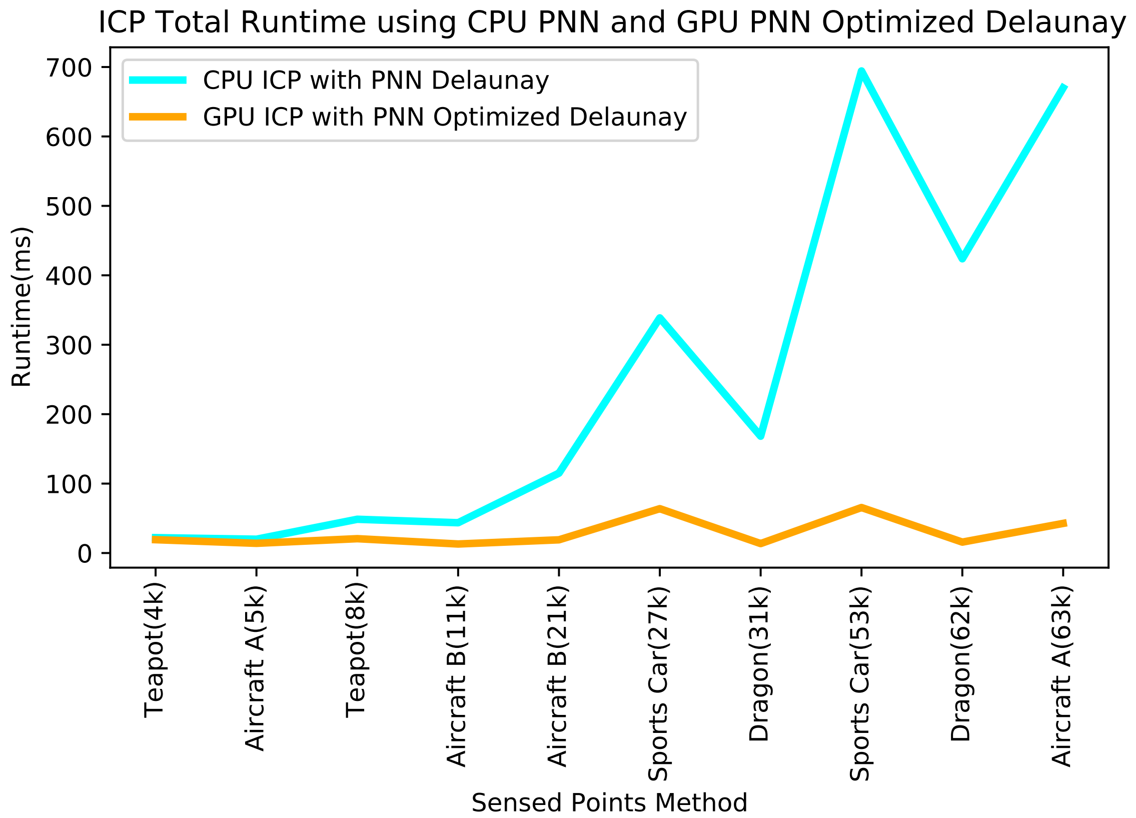

Figure 33: This shows the total ICP runtime when using the CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* nearest neighbor algorithms.

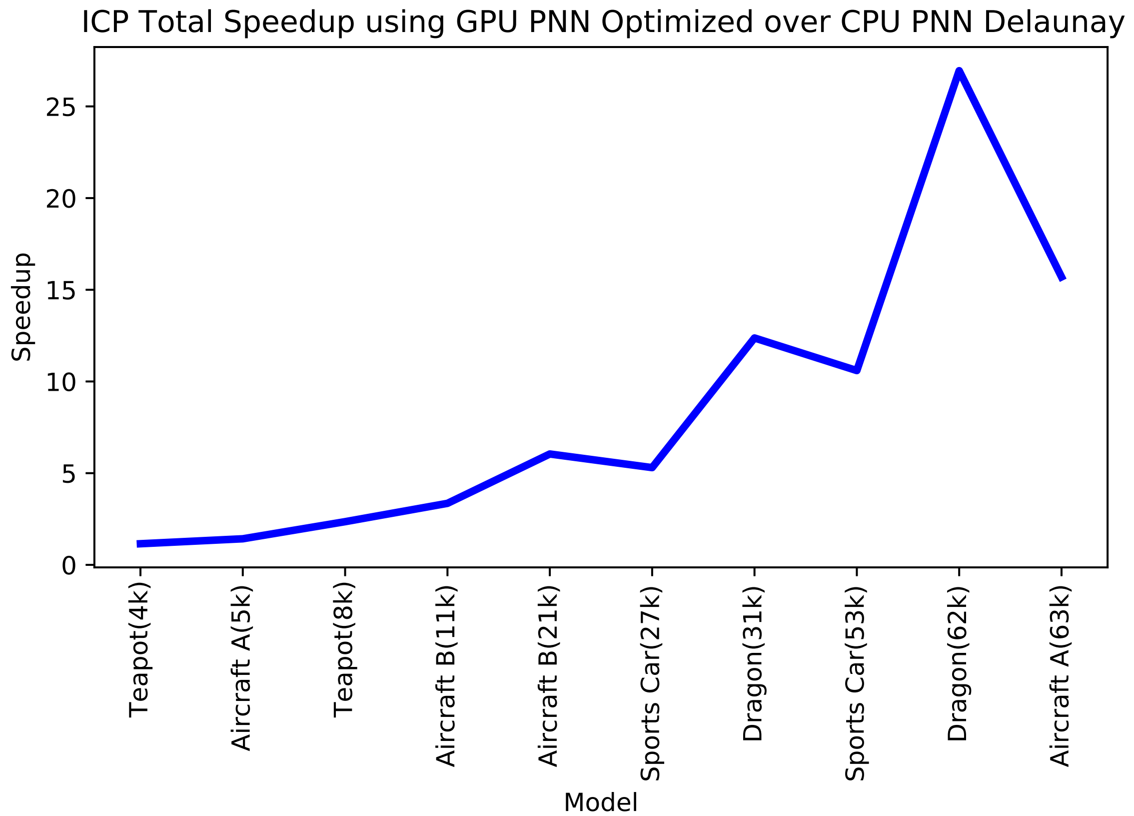

Figure 34: This shows the ICP speedup when using GPU *PNN Optimized Delaunay* nearest neighbor algorithms over CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* nearest neighbor algorithms.

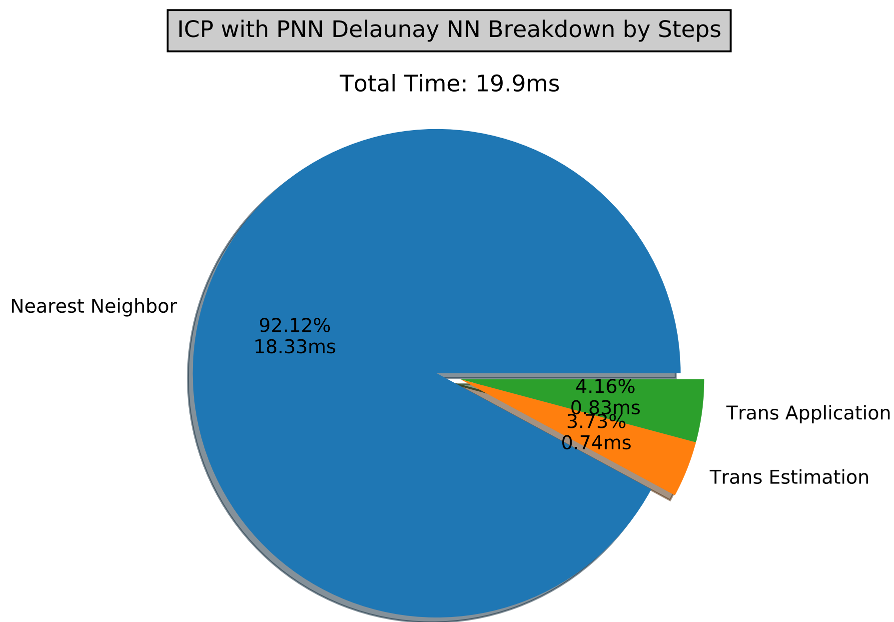

Figure 35: CPU ICP with *PNN Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(5k) model.

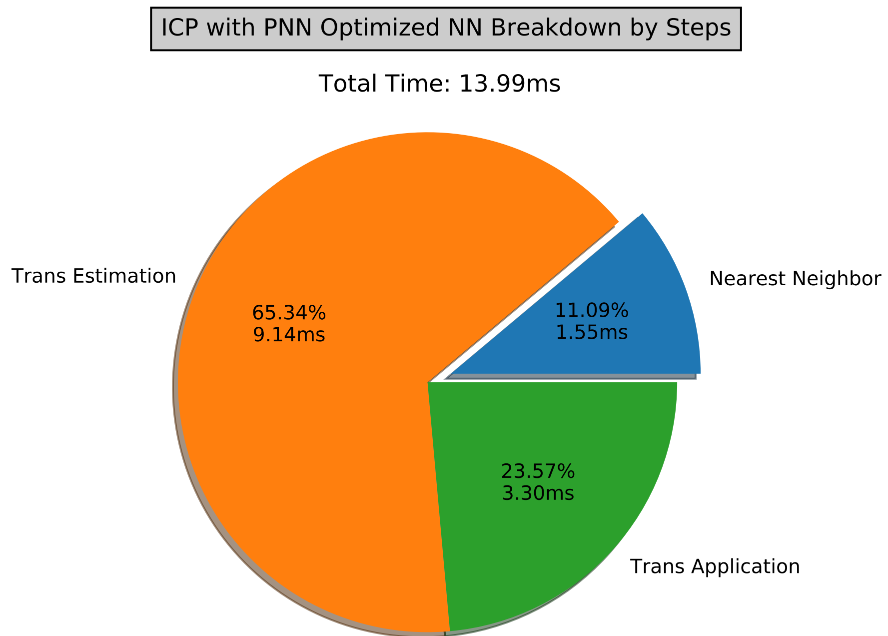

Figure 36: GPU ICP with *PNN Optimized Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(5k) model.

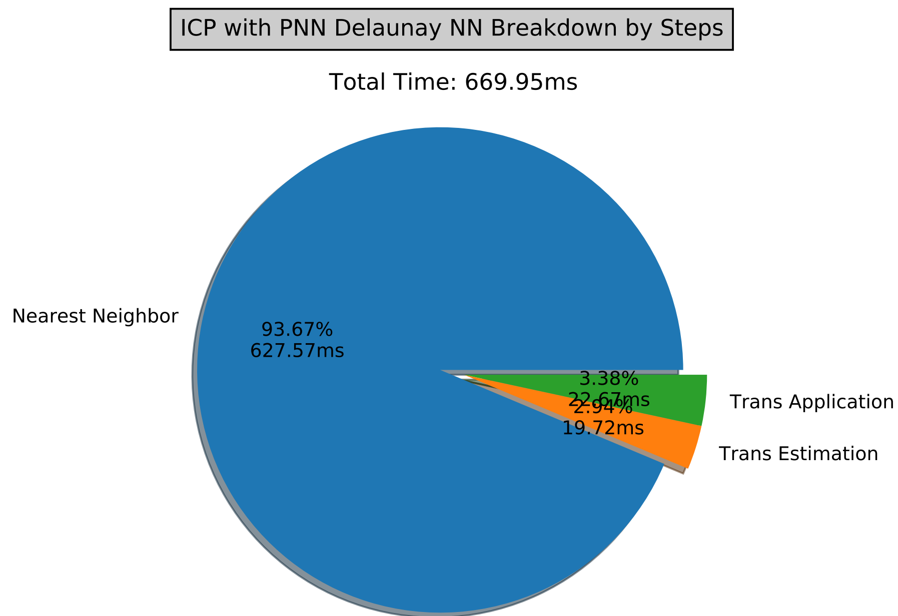

Figure 37: CPU ICP with *PNN Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(63k) model.

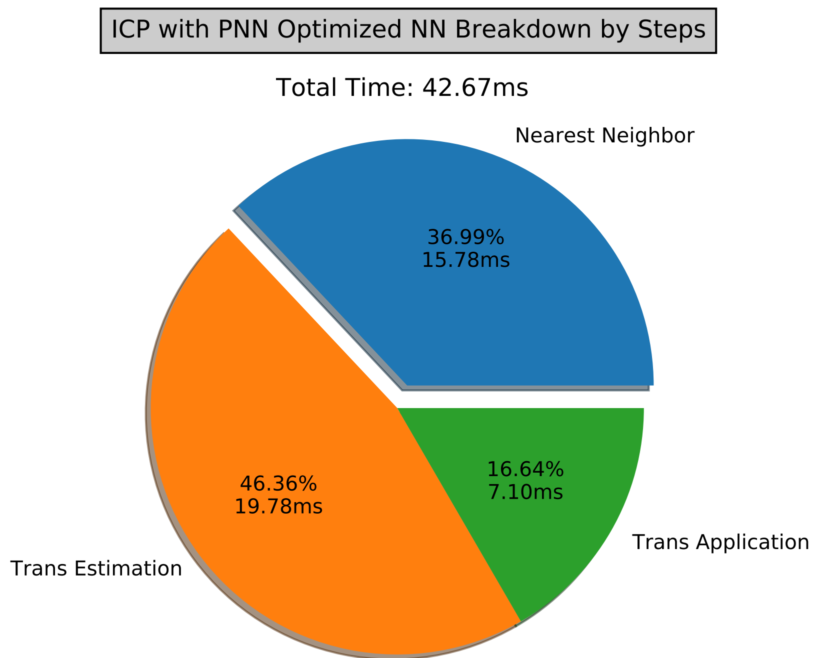

Figure 38: GPU ICP with *PNN Optimized Delaunay* nearest neighbor algorithm breakdown by steps on the Aircraft A(63k) model.

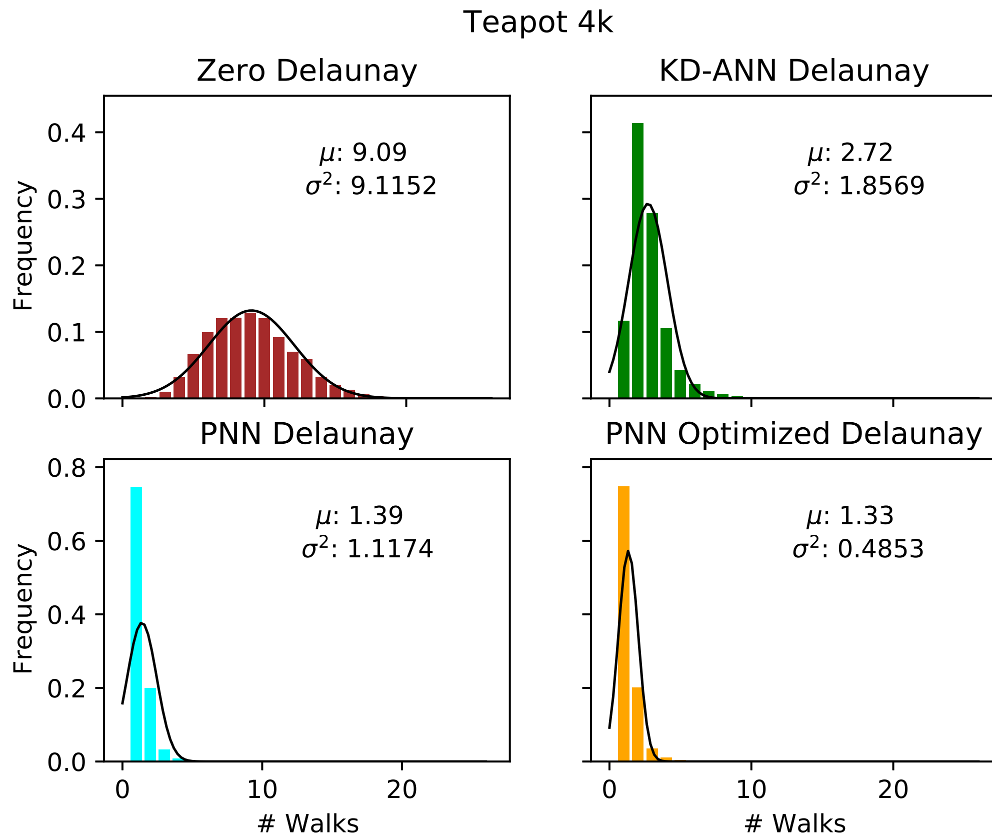

Figure 39: This shows the frequency of how many walks are taken by the algorithms for the 4k Teapot model.

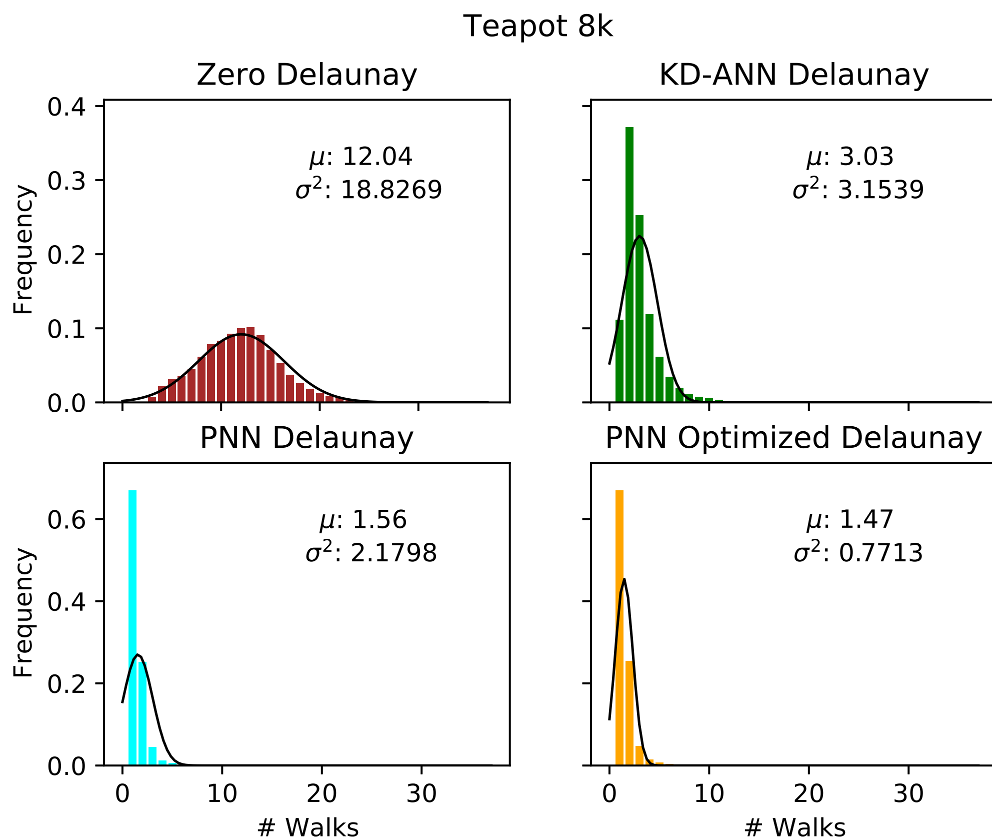

Figure 40: This shows the frequency of how many walks are taken by the algorithms for the 8k Teapot model.

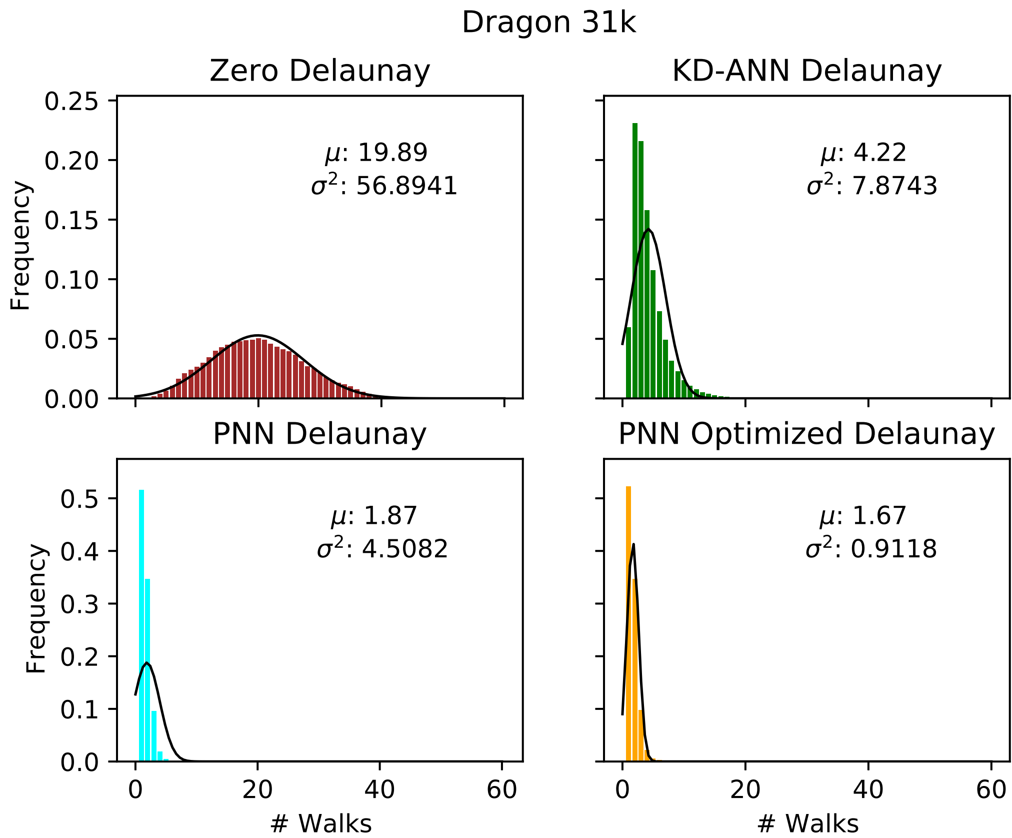

Figure 41: This shows the frequency of how many walks are taken by the algorithms for the 31k Dragon model.

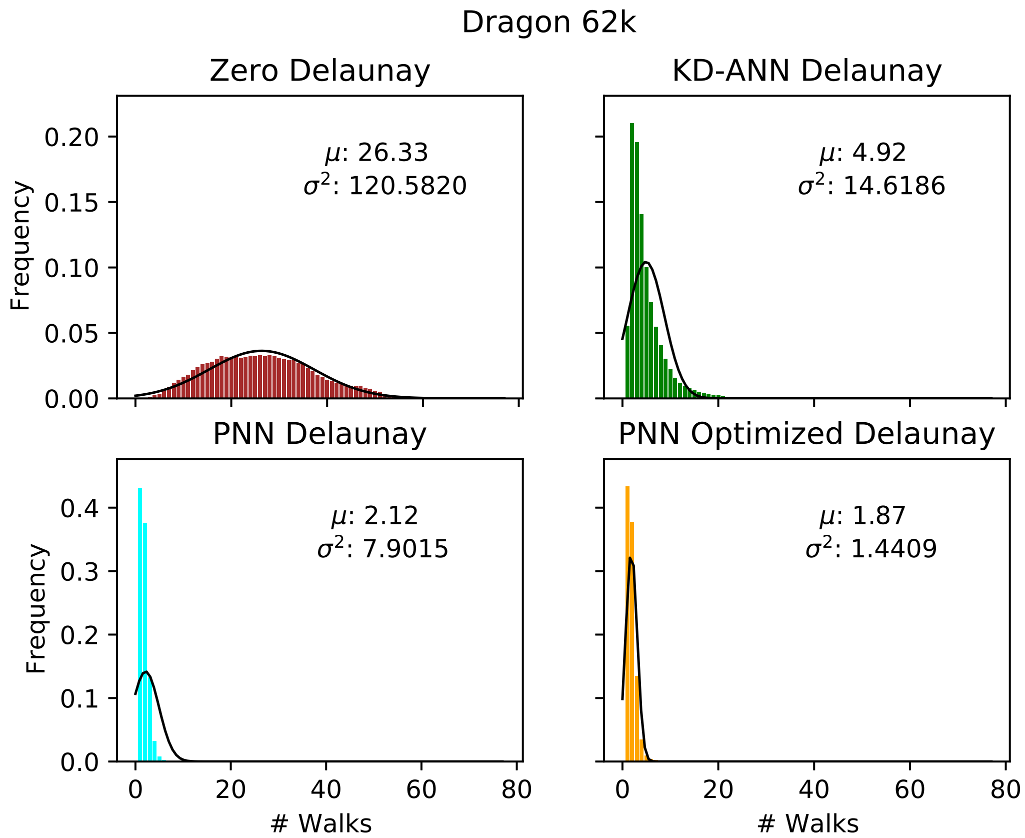

Figure 42: This shows the frequency of how many walks are taken by the algorithms for the 62k Dragon model.

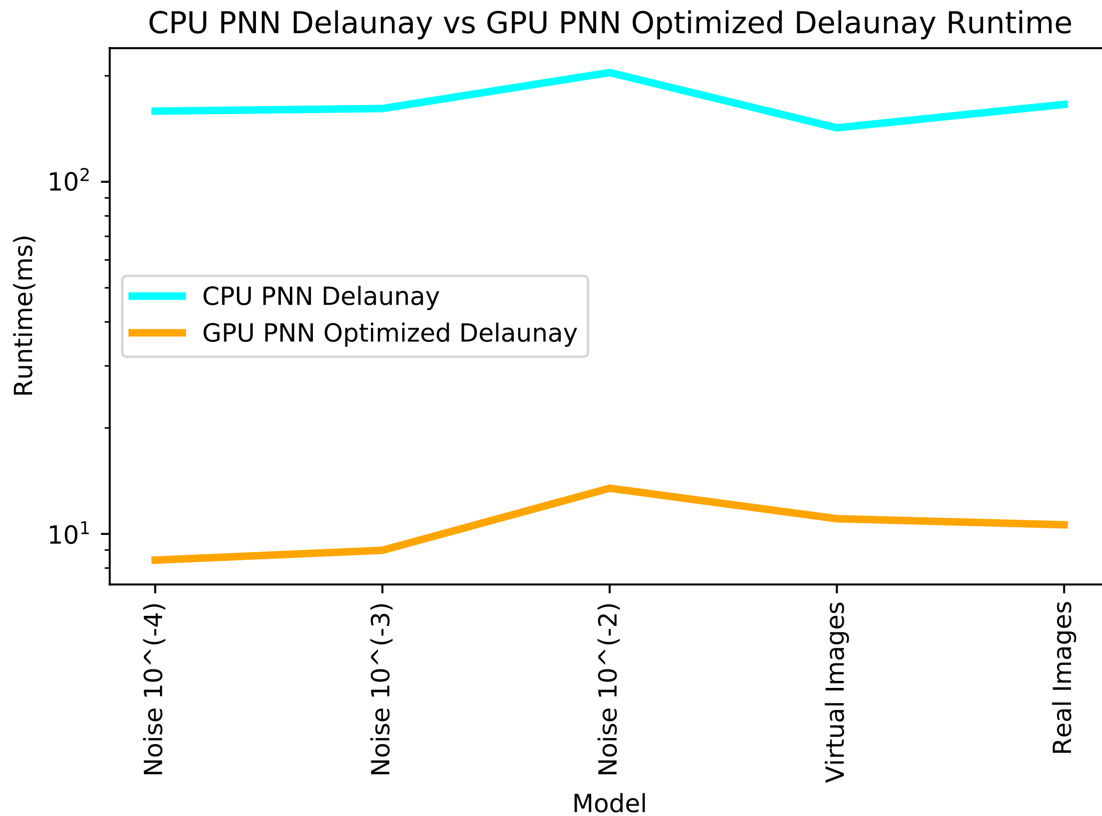

Figure 43: This shows the nearest neighbor runtime between CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* with noise, virtual images, and real images all on the Aircraft B(21k) model.

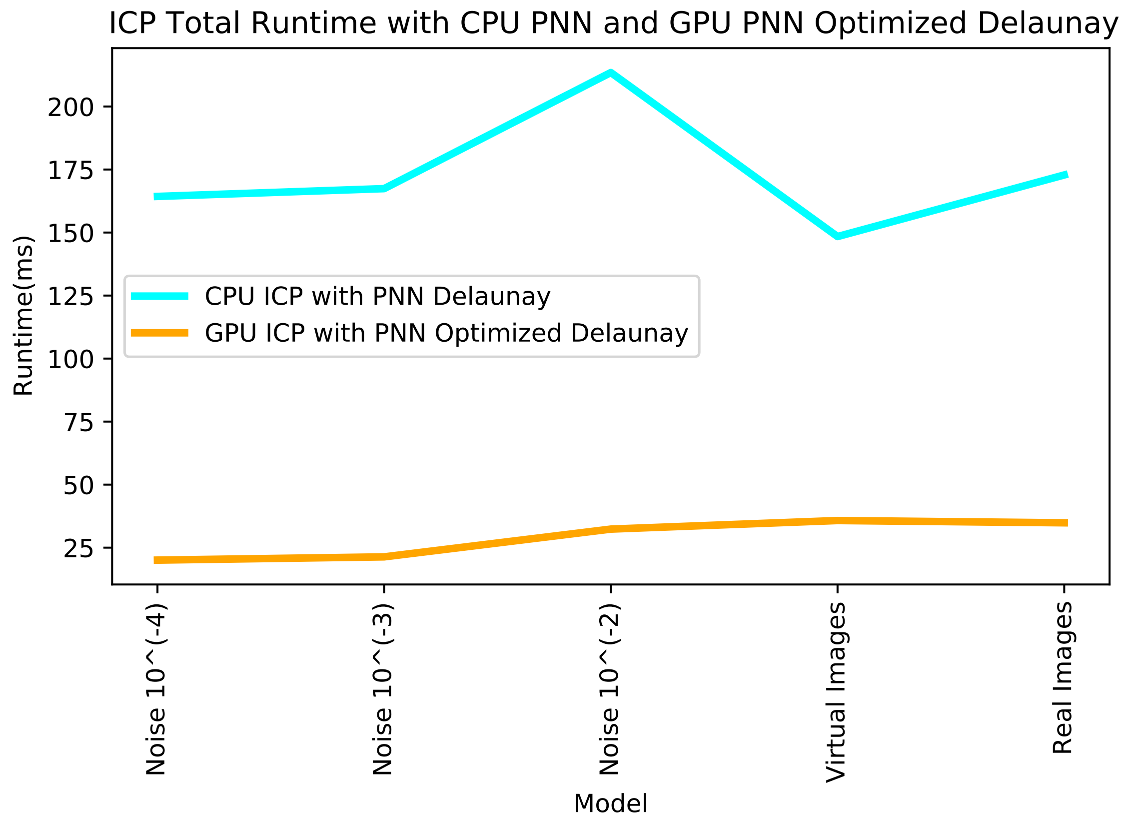

Figure 44: This shows the total ICP runtime when using the CPU *PNN Delaunay* and GPU *PNN Optimized Delaunay* nearest neighbor algorithms with noise, virtual images, and real images all on the Aircraft B(21k) model.

4.4 Virtual and Real Stereo Block Matching

From processing the reprojected points from real and virtual stereo block matching, it can be determined that the CPU and GPU implementation's of the Delaunay traversal and ICP perform with the same accuracy. As seen in Figs. 45 and 46, when registering points generated from virtual images, the CPU and GPU implementations perform with the same accuracy. Furthermore, when the virtual images are replaced with real images, and the points generated from stereo block matching contain more error, both CPU and GPU implementation still perform with the exact same level of accuracy, as is demonstrated in Figs. 47 and 48. Figure 49 shows error vs distance on real imagery with the GPU ICP filter applied. This shows a slightly tighter bounded error range than without the filter. This means the error is more consistent than without the filter.

Tables 6 and 7 show the positional and rotational mean magnitude error of ICP when using real images experiments. This shows the CPU and GPU have a very close error and the minor difference is due to hardware differences that may cause rounding differences. The CPU reports a slightly more accurate position and the GPU reports a slightly more accurate rotation. Overall, there's less than an average magnitude of 7.3 centimeters in positional error and a magnitude of 1.3 degrees of rotational error.

The GPU ICP filter increases the rotational accuracy of the GPU. However, it resulted in a slightly less accurate position. The filter reduced the size of the confidence interval for both position and rotation. The filter increased the consistency of the error. Overall, the filter did increase the accuracy and consistency of ICP on the GPU.

Figure 50 shows the AAR vision pipeline runtime breakdown when using Region of Interest (ROI) on real 4k images. ROI allows stereo block matching to search only the portion of the images where the sensed model is located, to reduce runtime. This

shows ICP is no longer the computational bottleneck of the AAR vision pipeline. Table 8 shows the stereo block matching and reprojection runtimes. This shows ROI does reduce the runtime.

Table 6: This shows the positional mean magnitude error in meters as well as the 99.5% confidence interval.

Method	**Positional Mean Magnitude Error (Meters)**	**99.5% Confidence Interval (Meters) (+/-)**
CPU	0.016	0.0002
GPU	0.016	0.0009
GPU Filter	0.016	0.0004

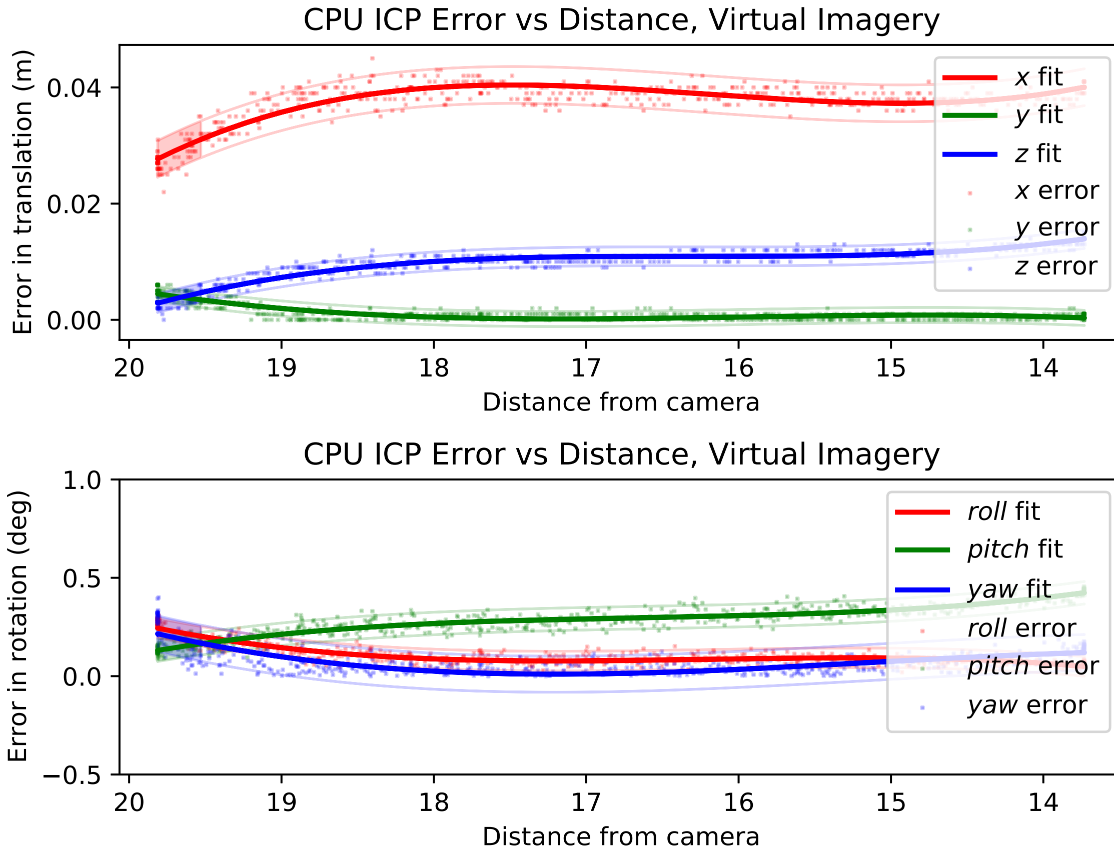

Figure 45: Errors in translation and rotation from ICP on virtual imagery on the CPU

Table 7: This shows the rotational mean magnitude error in degrees as well as the 99.5% confidence interval.

Method	**Rotational Mean Magnitude Error (Degrees)**	**99.5% Confidence Interval (Degrees) (+/-)**
CPU	1.29	0.018
GPU	1.27	0.070
GPU Filter	1.26	0.041

Table 8: This shows the SBM/Reprojection mean magnitude runtime in seconds as well as the 99.5% confidence interval.

Method	**Runtime Mean Magnitude (ms)**	**99.5% Confidence Interval (ms) (+/-)**
4k Images	302.10	0.411
4k Images ROI	266.89	0.417
2k Images	53.42	0.023

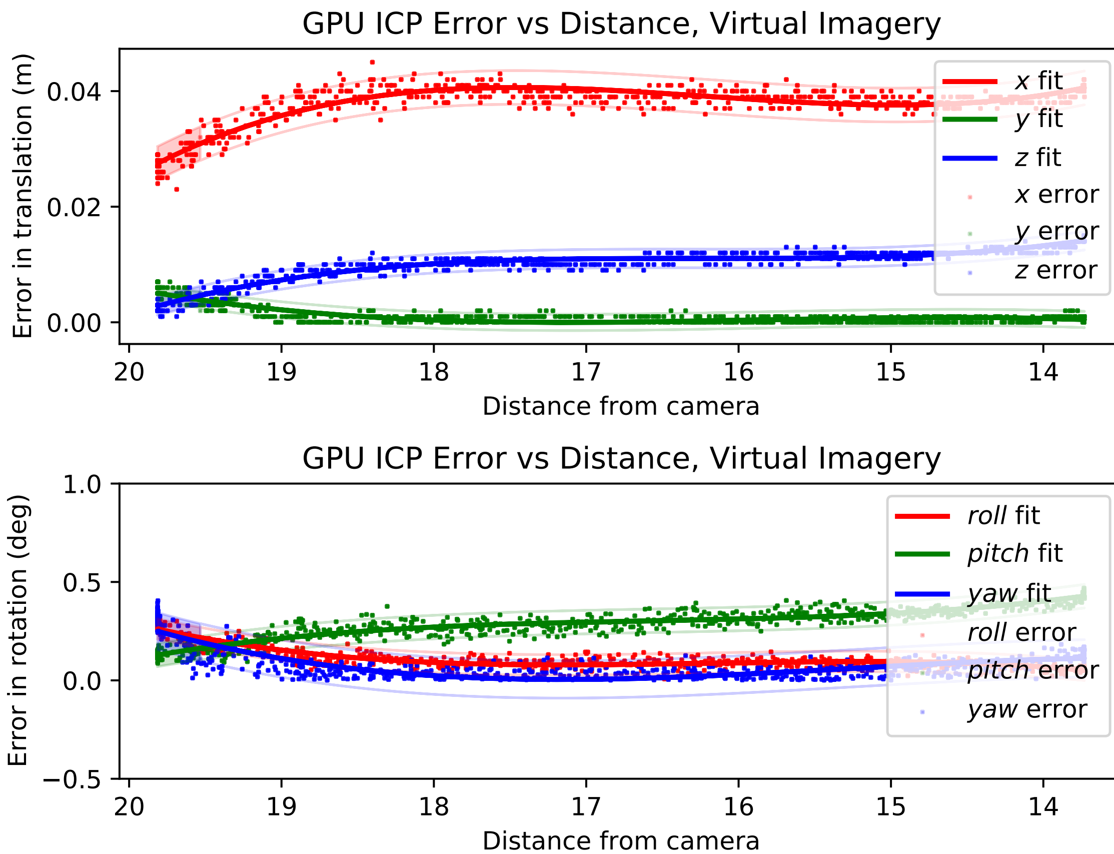

Figure 46: Errors in translation and rotation from ICP on virtual imagery on the GPU

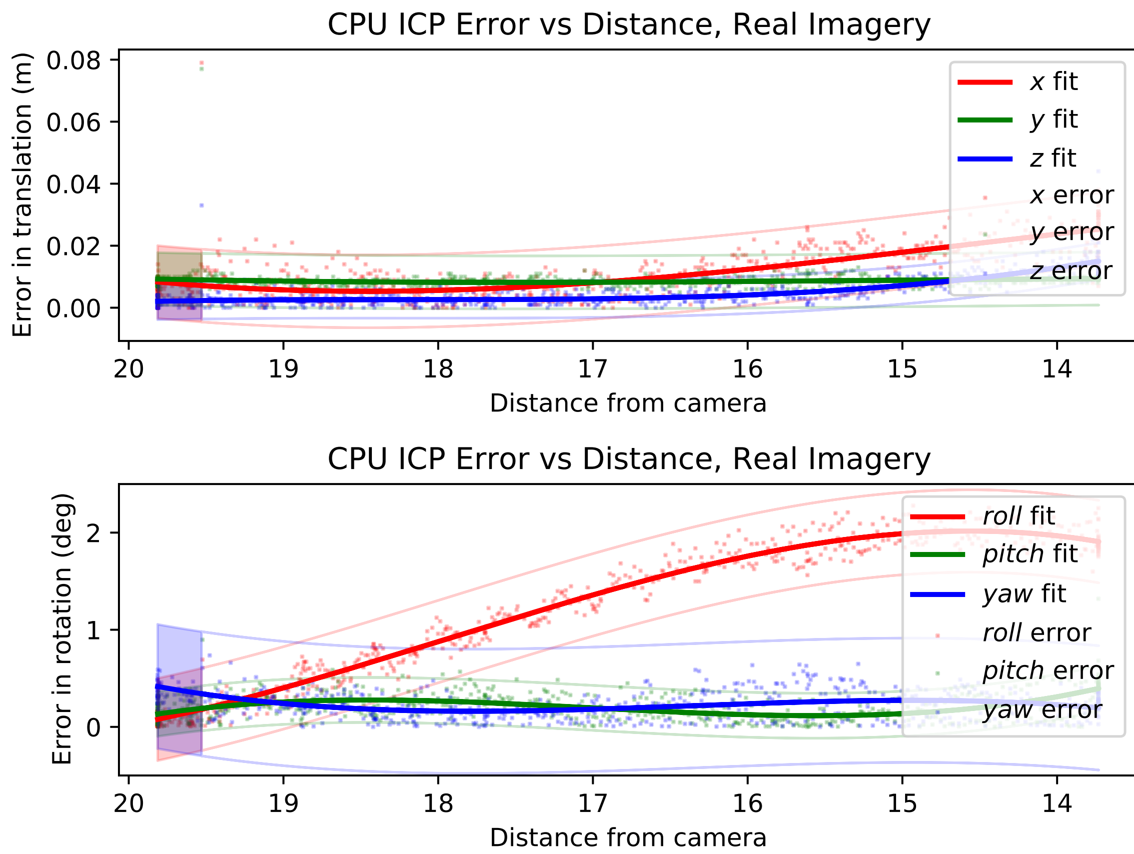

Figure 47: Errors in translation and rotation from ICP on real imagery on the CPU

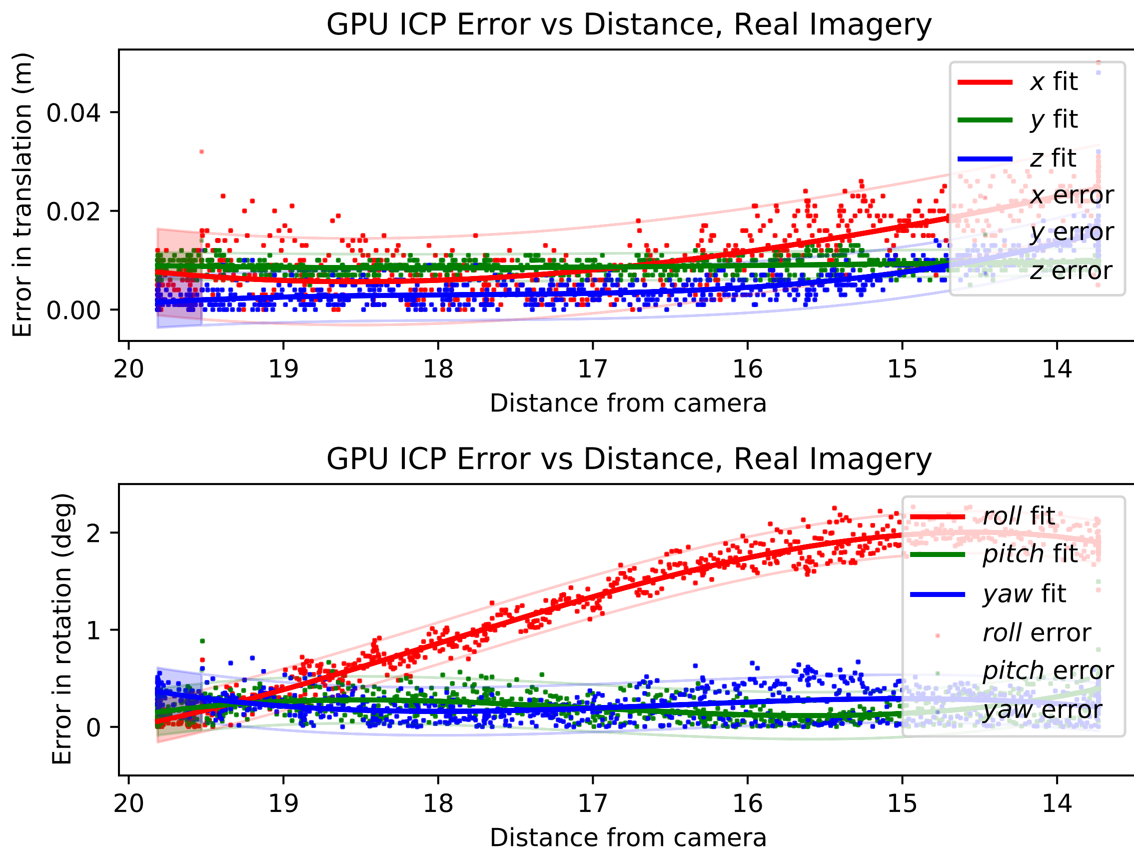

Figure 48: Errors in translation and rotation from ICP on real imagery on the GPU

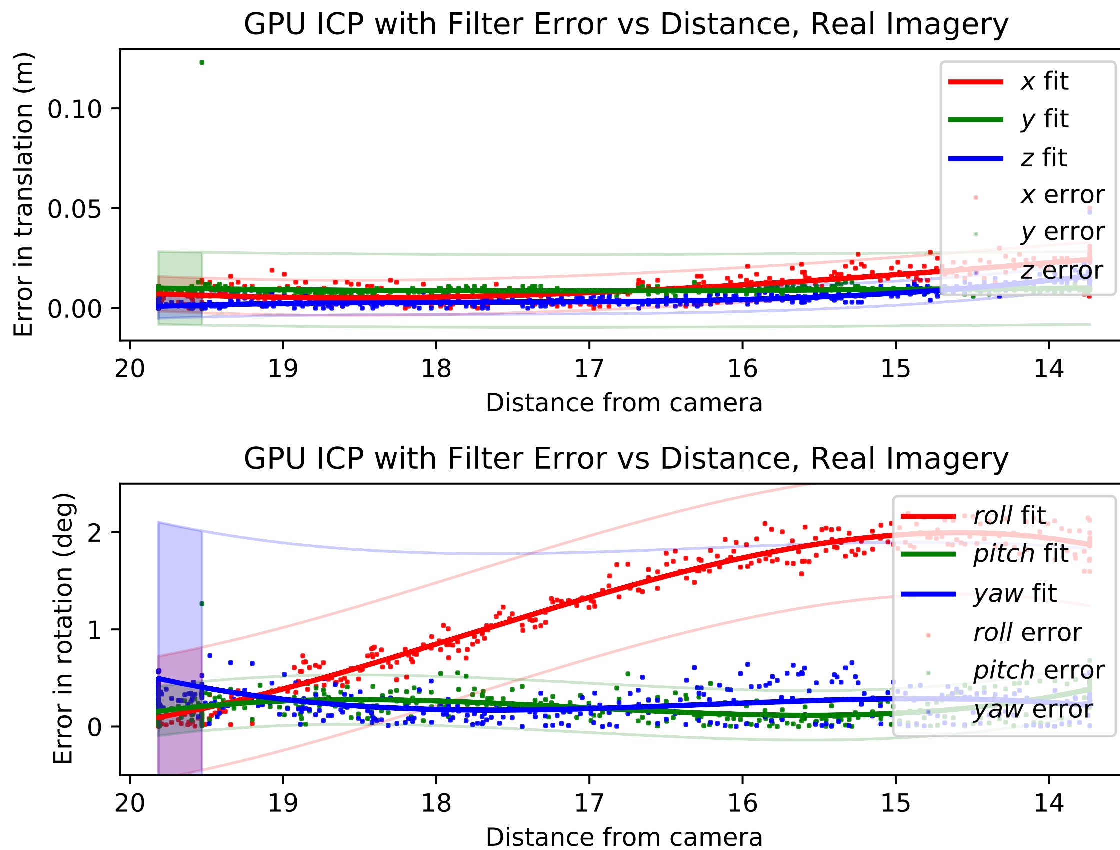

Figure 49: Errors in translation and rotation from ICP on real imagery on the GPU with filter

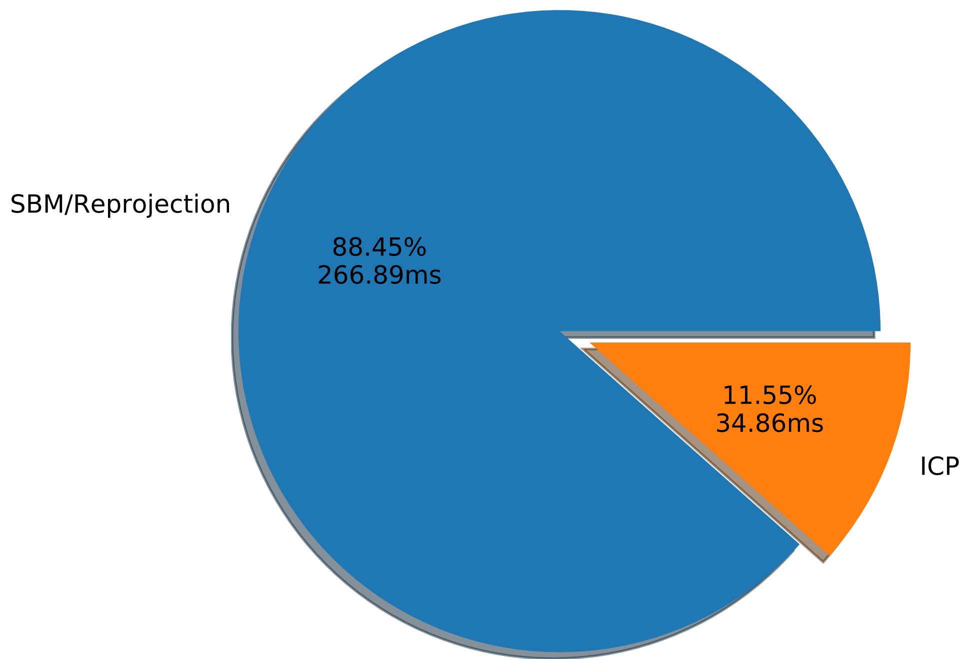

Figure 50: This shows the AAR vision processing pipeline runtime from image acquisition with real 4k images using ROI and ICP with with GPU *PNN Optimized Delaunay* on the Aircraft B(21k) model.

V. Conclusions

In this study, the Iterative Closest Point (ICP) algorithm was expressed as a massively parallel algorithm and implemented and mapped to an Nvidia Titan V graphics processing unit (GPU) and an Nvidida RTX 3080 GPU. Highly efficient and novel nearest neighbor matching algorithms were introduced and implemented based on the Delaunay triangulation. The algorithmic analysis and runtime experiments showed *Previous Nearest Neighbor (PNN) Delaunay* is the most efficient on the central processing unit (CPU) and *PNN Delaunay Optimized* on the GPU. The algorithm, implementation, and parallelization approaches for each step of the algorithm were discussed. The runtime and speedup with respect to the CPU implementation were compared to realize a speedup of approximately 2 orders of magnitude. Additionally, the specific portion and runtime of the Compute Unified Device Architecture (CUDA) kernels were profiled. Also, an algorithmic analysis and comparison was conducted of the parallel algorithm between theoretical and real runtimes. Lastly, Registration of simulated and real sensed points was presented with a low error tolerance to show robustness. Automated Aerial Refueling (AAR) will be able to use this research to accelerate the point registration block of the vision processing pipeline and rapidly calculate the receiving aircraft's position and orientation (pose).

5.1 Future Work

Future work will execute experiments with higher point totals and also more real-time experiments with different models. It is planned to execute the experiments with different CPUs and GPUs. This includes an application using the NVLink interconnect with multiple GPUs. Additionally, it is planned to integrate the nearest neighbor algorithms into other applications like point-to-plane ICP. To enable real-

time AAR, future work should include accelerating stereo block matching and point reprojection portions of the vision processing pipeline. Lastly, an ideal comparison between the serial and parallel algorithms would compare identical processors for the serial and parallel implementations respectively.

Appendix A. Additional Results

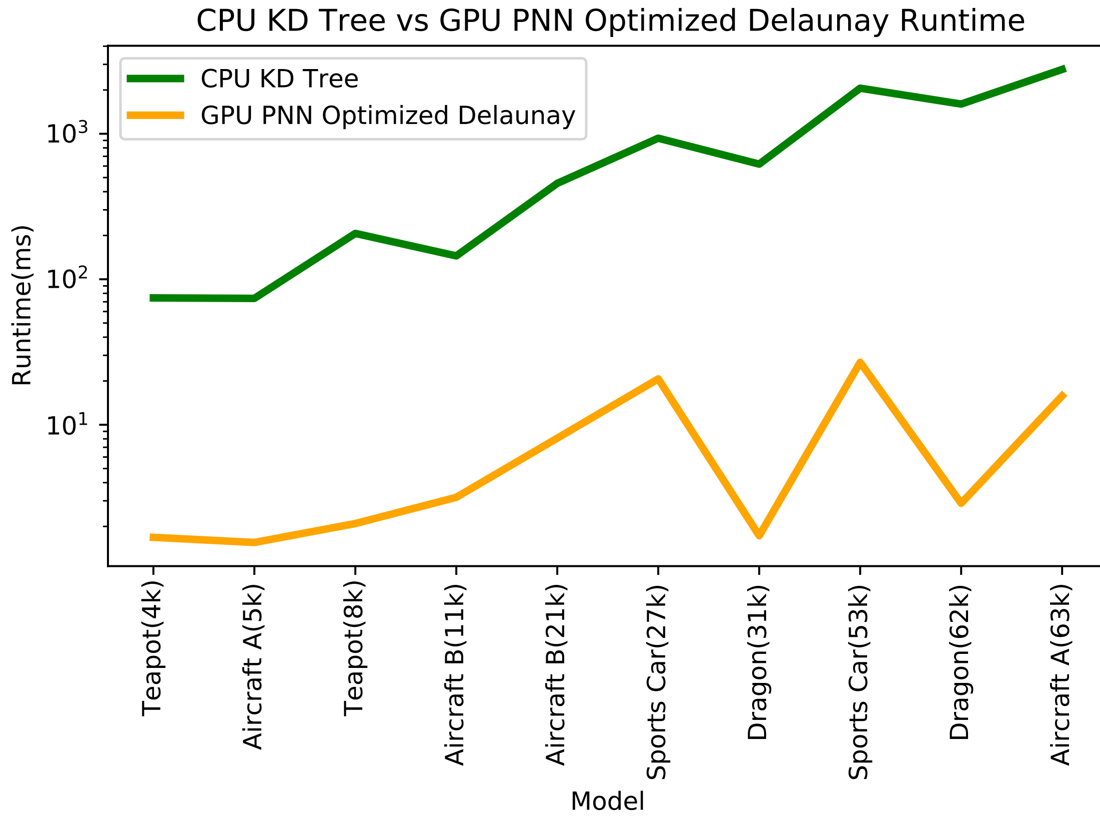

Figure 51: This shows the CPU *KD Tree* and GPU *PNN Optimized Delaunay* nearest neighbor runtimes.

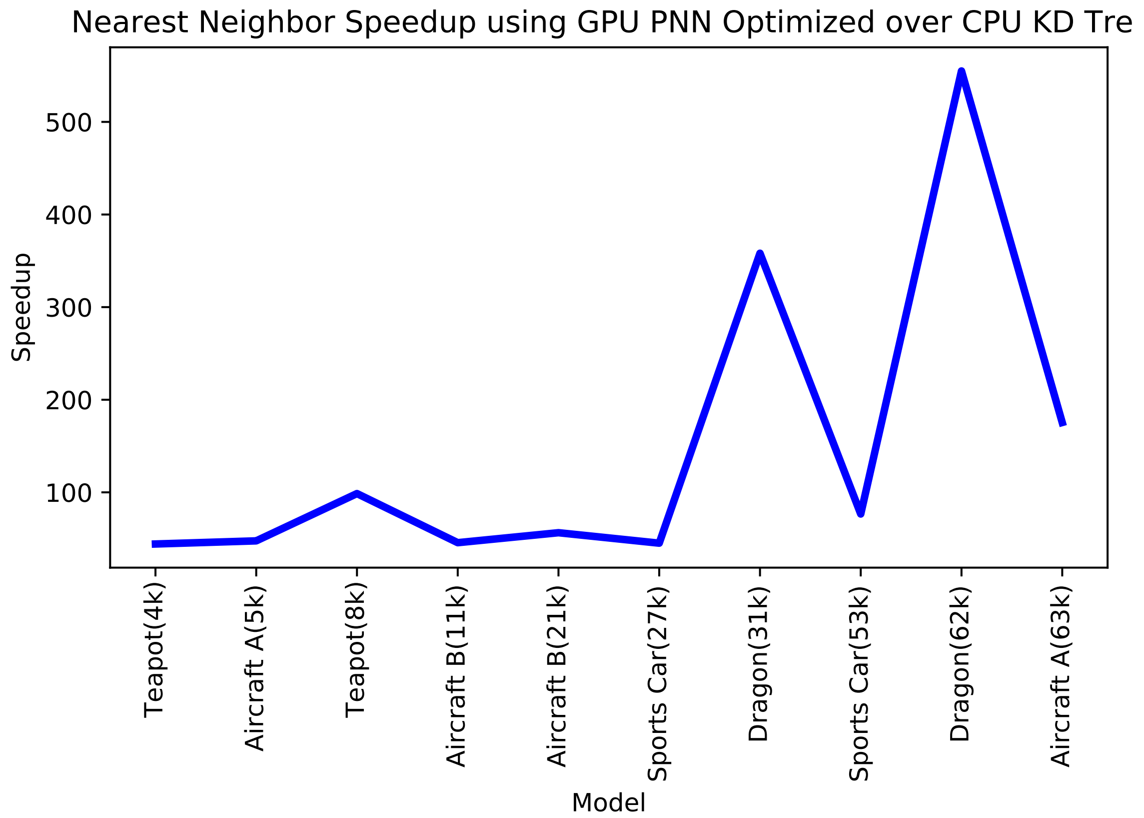

Figure 52: This shows GPU *PNN Optimized Delaunay* over CPU *KD Tree* nearest neighbor speedup.

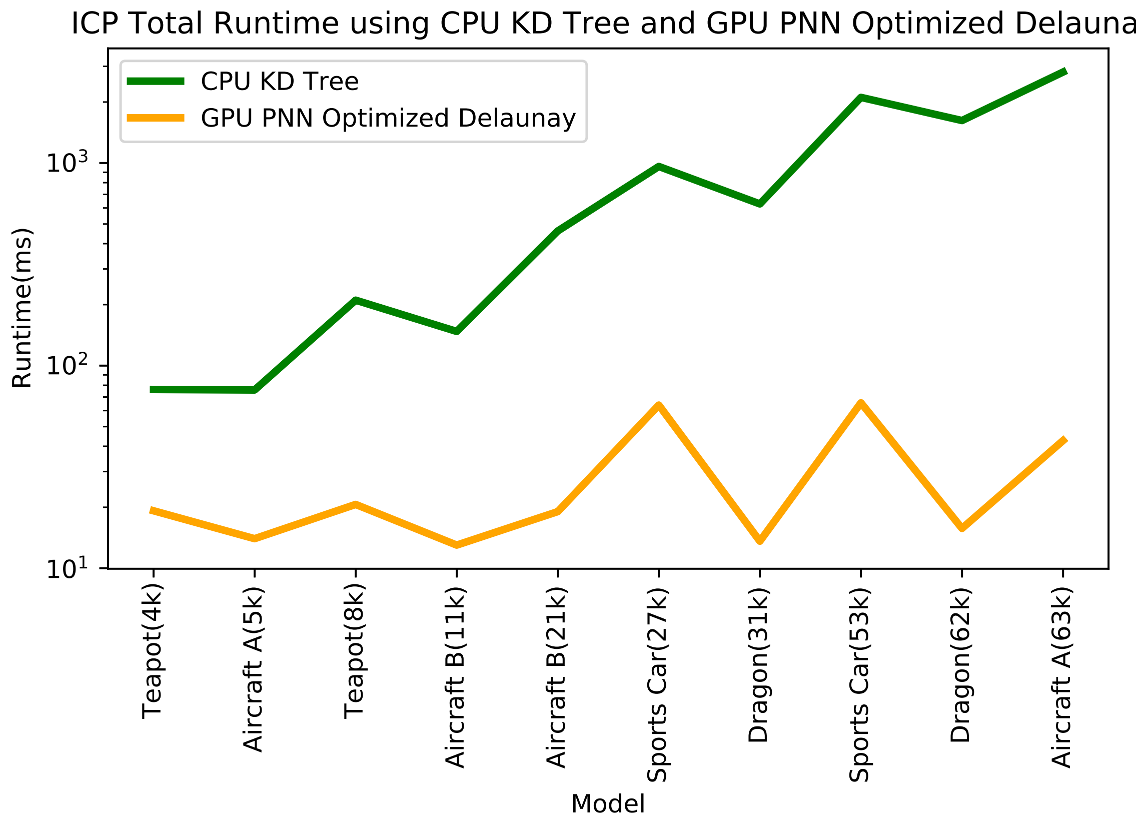

Figure 53: This shows the CPU *KD Tree* and GPU *PNN Optimized Delaunay* total ICP runtimes.

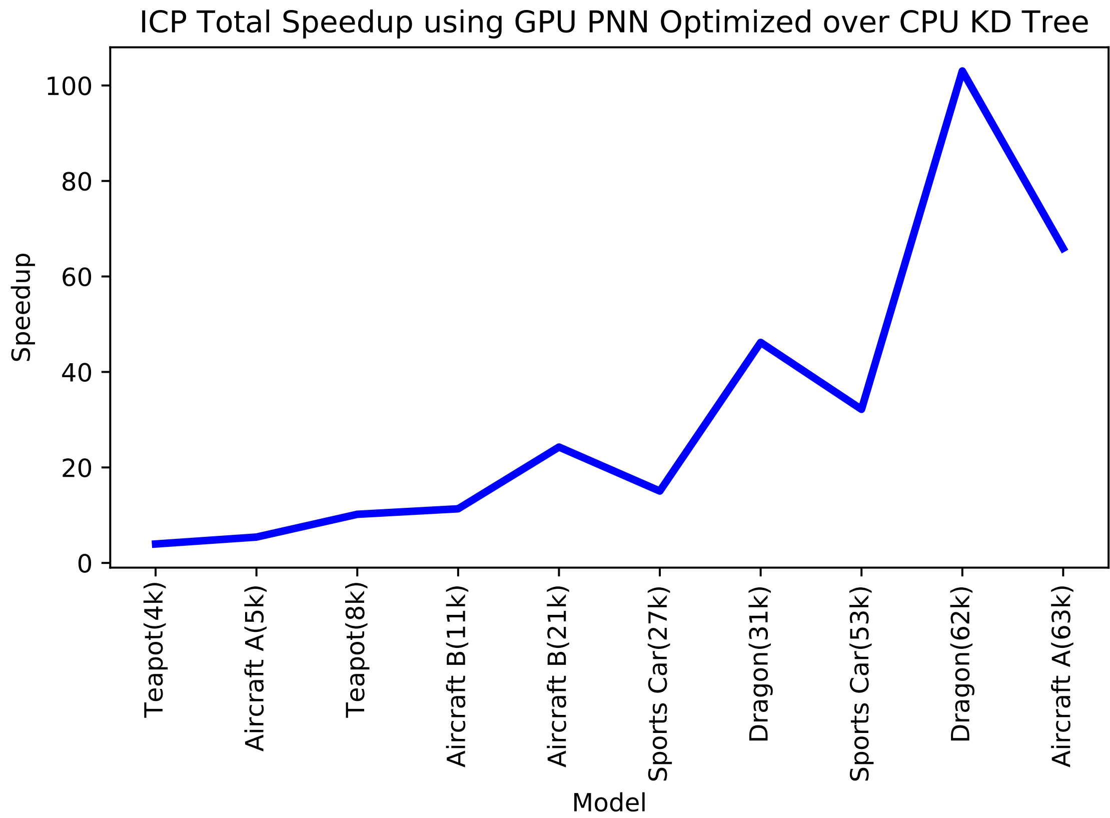

Figure 54: This shows GPU *PNN Optimized Delaunay* over CPU *KD Tree* total ICP speedup.

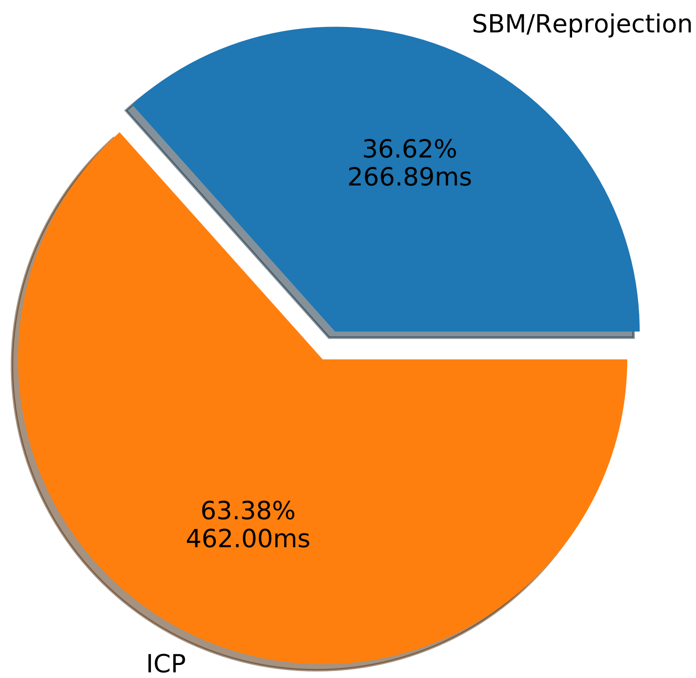

Figure 55: This shows the AAR vision processing pipeline runtime from image acquisition with real 4k images using ROI and ICP with with CPU *KD Tree* on the Aircraft B(21k) model.

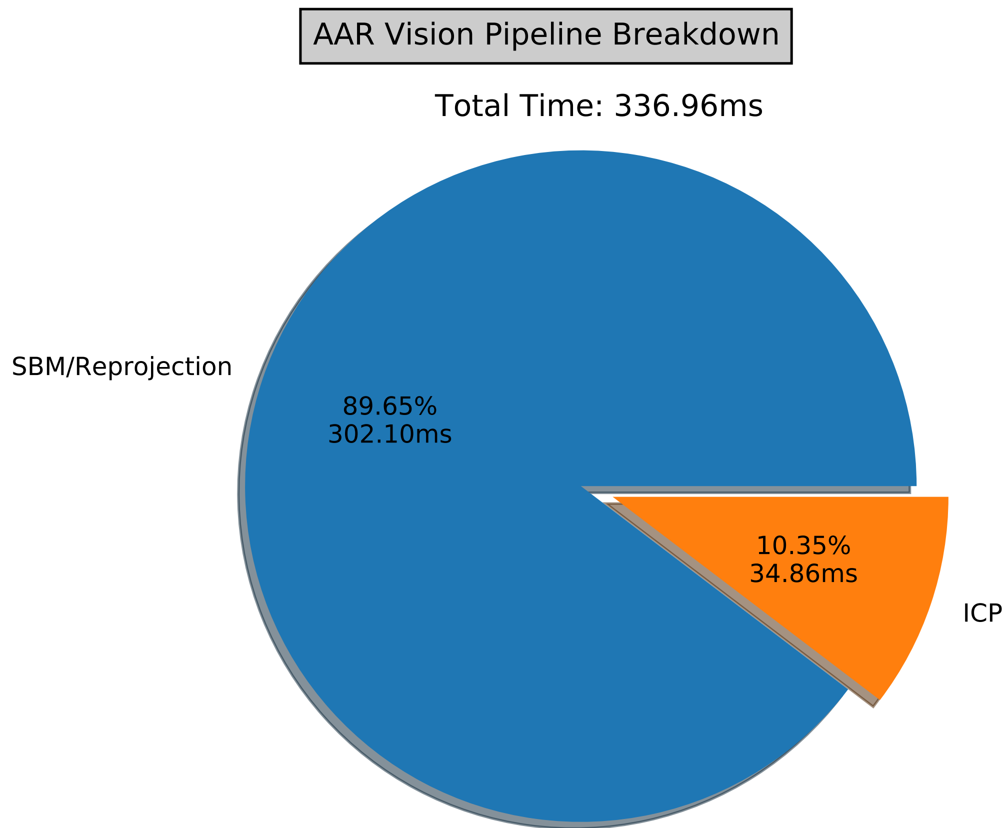

Figure 56: This shows the AAR vision processing pipeline runtime from image acquisition with real 4k images without ROI and ICP with with GPU *PNN Optimized Delaunay* on the Aircraft B(21k) model.

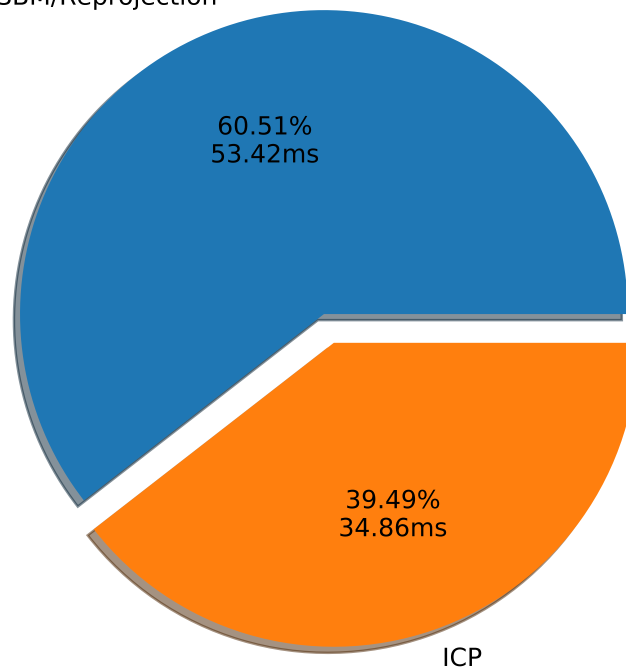

Figure 57: This shows the AAR vision processing pipeline runtime from image acquisition with real 2k images without ROI and ICP with with GPU *PNN Optimized Delaunay* on the Aircraft B(21k) model.

Appendix B. Code Implementation

For code to implement this project and research please go to the git repository: `https://git.nykl.net/aar/aarviz_picp.git` and checkout branch *master*.

Bibliography

1. Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606. International Society for Optics and Photonics, 1992.

2. Kurt Konolige. Small vision systems: Hardware and implementation. In *Robotics research*, pages 203–212. Springer, 1998.

3. Andriy Myronenko and Xubo Song. Point set registration: Coherent point drift. *IEEE transactions on pattern analysis and machine intelligence*, 32(12):2262–2275, 2010.

4. Hao Zhu, Bin Guo, Ke Zou, Yongfu Li, Ka-Veng Yuen, Lyudmila Mihaylova, and Henry Leung. A review of point set registration: From pairwise registration to groupwise registration. *Sensors*, 19(5):1191, 2019.

5. Andriy Myronenko and Xubo Song. Point set registration: Coherent point drift. *IEEE transactions on pattern analysis and machine intelligence*, 32(12):2262–2275, 2010.

6. Robert L Stamper, Marc F Lieberman, and Michael V Drake. Chapter 8 - visual field theory and methods. In Robert L Stamper, Marc F Lieberman, and Michael V Drake, editors, *Becker-Shaffer's Diagnosis and Therapy of the Glaucomas (Eighth Edition)*, pages 91 – 97. Mosby, Edinburgh, eighth edition edition, 2009.

7. M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

8. Kilian Q. Weinberger and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. *J. Mach. Learn. Res.*, 2009.

9. François Pomerleau, Francis Colas, and Roland Siegwart. A Review of Point Cloud Registration Algorithms for Mobile Robotics. *Found. Trends Robot.*, 2015.

10. T. M. Cover and P. E. Hart. Nearest Neighbor Pattern Classification. *IEEE Trans. Inf. Theory*, 1967.

11. Tracker documentation - tracker 3.9 documentation - vicon documentation. `https://docs.vicon.com/display/Tracker39`. (Accessed on 01/13/2021).

12. Ryan M Raettig, James D Anderson, Laurence D Merkle, and Scott L Nykl. Accelerated point set registration method, Oct 2020.

13. James D Anderson, Ryan M Raettig, and Scott L Nykl. Fast nearest neighbor matching delaunay walk algorithm. 2020.

14. Yang Chen and Gerard Medioni. Object modeling by registration of multiple range images. In *Proc. - IEEE Int. Conf. Robot. Autom.*, 1991.

15. Ben Bellekens, Vincent Spruyt, Raf Berkvens, Rudi Penne, and Maarten Weyn. A benchmark survey of rigid 3d point cloud registration algorithms. *International Journal On Advances in Intelligent Systems*, 1, 06 2015.

16. Kl Low. Linear Least-squares Optimization for Point-to-plane ICP Surface Registration. *Chapel Hill, Univ. North Carolina*, 2004.

17. Aleksandr Segal, Dirk Hähnel, and Sebastian Thrun. Generalized-icp. 06 2009.

18. Andreas Nüchter, Kai Lingemann, and Joachim Hertzberg. Cached k-d tree search for ICP algorithms. In *3DIM 2007 - Proc. 6th Int. Conf. 3-D Digit. Imaging Model.*, 2007.

19. M. Greenspan and M. Yurick. Approximate k-d tree search for efficient ICP. In *Proc. Int. Conf. 3-D Digit. Imaging Model. 3DIM*, 2003.

20. Soon-Yong Park and Murali Subbarao. An accurate and fast point-to-plane registration technique. *Pattern Recognition Letters*, 24(16):2967–2976, 2003.

21. Y. Chen and G. Medioni. Object modeling by registration of multiple range images. *Proceedings. 1991 IEEE International Conference on Robotics and Automation, Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, page 2724, 1991.

22. V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, 2008.

23. Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

24. Robert F Sproull. Refinements to nearest-neighbor searching ink-dimensional trees. *Algorithmica*, 6(1-6):579–589, 1991.

25. Tomoyuki Shibata, Takekazu Kato, and Toshikazu Wada. Kd decision tree: An accelerated and memory efficient nearest neighbor classifier. In *Third IEEE International Conference on Data Mining*, pages 641–644. IEEE, 2003.

26. Jace Robinson, Matt Piekenbrock, Lee Burchett, Scott Nykl, Brian Woolley, and Andrew Terzuoli. Parallelized iterative closest point for autonomous aerial refueling. In *International Symposium on Visual Computing*, pages 593–602. Springer, 2016.

27. Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

28. Christian Langis, Michael Greenspan, and Guy Godin. The parallel iterative closest point algorithm. In *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, pages 195–202. IEEE, 2001.

29. Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

30. Md Mushfiqur Rahman, Panagiota Galanakou, and Georgios Kalantzis. A fast gpu point-cloud registration algorithm. *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2018 19th IEEE/ACIS International Conference on*, pages 111 – 116, 2018.

31. Jar-Ferr Yang and Chiou-Liang Lu. Combined techniques of singular value decomposition and vector quantization for image coding. *IEEE Transactions on Image Processing*, 4(8):1141–1146, 1995.

32. T. Tamaki, M. Abe, B. Raytchev, and K. Kaneda. Softassign and em-icp on gpu. *2010 First International Conference on Networking and Computing, Networking and Computing (ICNC), 2010 First International Conference on*, pages 179 – 183, 2010.

33. Steven Gold, Anand Rangarajan, Chien-Ping Lu, Suguna Pappu, and Eric Mjolsness. New algorithms for 2d and 3d point matching: pose estimation and correspondence. *Pattern recognition*, 31(8):1019–1031, 1998.

34. Sébastien Granger and Xavier Pennec. Multi-scale em-icp: A fast and robust approach for surface registration. In *European Conference on Computer Vision*, pages 418–432. Springer, 2002.

35. NVIDIA. Nvidia cuda compute unified device architecture programming guide version 1.0. `http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`. (Accessed on 06/25/2020).

36. NVIDIA. CUDA Toolkit Documentation v11.2.76.

37. Lena Oden. Lessons learned from comparing c-cuda and python-numba for gpu-computing. *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Parallel, Distributed and Network-Based Processing (PDP), 2020 28th Euromicro International Conference on*, pages 216 – 223, 2020.

38. Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

39. Vincent Garcia, Eŕic Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *Proc. - Int. Conf. Image Process. ICIP*, 2010.

40. Benjamin Bustos, Oliver Deussen, Stefan Hiller, and Daniel Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, pages 196–199, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

41. Lawrence Cayton. Accelerating nearest neighbor search on manycore systems. In *Proc. 2012 IEEE 26th Int. Parallel Distrib. Process. Symp. IPDPS 2012*, 2012.

42. Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2227–2240, 2014.

43. Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 1975.

44. Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.*, 1977.

45. B. Delaunay. Sur la sphère du vide. *Bull. l'Académie des Sci. l'URSS*, 1934.

46. Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

47. Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

48. C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

49. Sterling J. Anderson, Sisir B. Karumanchi, and Karl Iagnemma. Constraint-based planning and control for safe, semi-autonomous operation of vehicles. In *IEEE Intell. Veh. Symp. Proc.*, 2012.

50. Xiang Yang Li, Gruia Calinescu, and Peng Jun Wan. Distributed construction of a planar spanner and routing for ad hoc wireless networks. In *Proc. - IEEE INFOCOM*, 2002.

51. Alexander Stepanov and James Mac Gregor Smith. Modeling wildfire propagation with Delaunay triangulation and shortest path algorithms. *Eur. J. Oper. Res.*, 2012.

52. F Cazals and J Giesen. Delaunay Triangulation Based Surface Reconstruction : Ideas and Algorithms. *INRIA Rapp. Rech.*, 2004.

53. N. Amenta and M. Bern. Surface reconstruction by Voronoi filtering. *Discret. Comput. Geom.*, 1999.

54. P. Labatut, J. P. Pons, and R. Keriven. Robust and efficient surface reconstruction from range data. *Comput. Graph. Forum*, 2009.

55. Kieran F. Mulchrone. Application of Delaunay triangulation to the nearest neighbour method of strain analysis. *J. Struct. Geol.*, 2003.

56. Teng Qiu and Yongjie Li. Nonparametric Nearest Neighbor Descent Clustering based on Delaunay Triangulation. *arXiv Prepr. arXiv1502.04837*, 2015.

57. Marcel Birn, Manuel Holtgrewe, Peter Sanders, and Johannes Singler. Simple and fast nearest neighbor search. In *2010 Proc. 12th Work. Algorithm Eng. Exp. ALENEX 2010*, 2010.

58. Bertram H. Drost and Slobodan Ilic. Almost constant-time 3D nearest-neighbor lookup using implicit octrees. *Mach. Vis. Appl.*, 2018.

59. Marius Muja and David G. Lowe. Fast approximate nearest neighbors with auto-

matic algorithm configuration. In *VISAPP 2009 - Proc. 4th Int. Conf. Comput. Vis. Theory Appl.*, 2009.

60. Jia Pan and Dinesh Manocha. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *GIS Proc. ACM Int. Symp. Adv. Geogr. Inf. Syst.*, 2011.

61. John A Stratton, Nasser Anssari, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Daniel Liu, and Wen-mei Hwu. Optimization and architecture effects on gpu computing workload performance. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. IEEE, 2012.

62. NVIDIA. The World's Most Powerful Graphics Card — NVIDIA TITAN V.

63. NVIDIA. volta-architecture-whitepaper.pdf. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. (Accessed on 06/27/2020).

64. 3rd gen ryzen™ threadripper™ 3970x — desktop processor — amd. `https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3970x`. (Accessed on 01/11/2021).

65. Geforce rtx 3080 graphics card — nvidia. `https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080/`. (Accessed on 01/11/2021).

66. M. Awatramani, J. Zambreno, and D. Rover. Increasing gpu throughput using kernel interleaved thread block scheduling. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 503–506, 2013.

67. W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *40th Annual IEEE/ACM*

International Symposium on Microarchitecture (MICRO 2007), pages 407–420, 2007.

68. Edward Richter, Ryan Raettig, Joshua Mack, Spencer Valancius, Burak Unal, and Ali Akoglu. Accelerated shadow detection and removal method. In *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–8. IEEE, 2019.

69. Raimund Seidel. The upper bound theorem for polytopes: an easy proof of its asymptotic version. *Comput. Geom. Theory Appl.*, 1995.

70. M. Greenspan and M. Yurick. Approximate k-d tree search for efficient icp. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 442–448, 2003.

71. Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: computer vision in C++ with the OpenCV library.* " O'Reilly Media, Inc.", 2016.

72. NVIDIA. Profiler :: Cuda toolkit documentation. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`. (Accessed on 06/27/2020).

73. Nicolai M Josuttis. The c++ standard library: a tutorial and reference. 2012.

74. Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: computer vision in C++ with the OpenCV library.* " O'Reilly Media, Inc.", 2016.

Acronyms

AAR Automated Aerial Refueling. iv, ix, xi, xii, 1, 3, 5, 7, 16, 47, 50, 55, 58, 84, 85, 91, 92, 93, 98, 99, 100, 1

AFIT Air Force Institute of Technology. 3

API application programming interface. 10

CPU central processing unit. viii, ix, x, xi, xii, xiii, 8, 16, 17, 20, 35, 47, 49, 50, 55, 56, 59, 62, 63, 64, 65, 66, 68, 69, 70, 72, 73, 74, 76, 82, 83, 84, 92, 94, 95, 96, 97, 98

CUDA Compute Unified Device Architecture. 5, 10, 11, 17, 20, 21, 30, 35, 37, 38, 56, 92

EM Expectation-Maximization. 10

FDH Full Delaunay Hierarchies. 13

GPU graphics processing unit. ix, x, xi, xii, xiii, 3, 5, 9, 10, 11, 16, 17, 20, 30, 35, 37, 38, 46, 47, 49, 50, 55, 56, 59, 62, 63, 64, 65, 67, 68, 69, 71, 72, 73, 75, 77, 82, 83, 84, 91, 92, 94, 95, 96, 97, 99, 100

ICP Iterative Closest Point. iv, viii, ix, x, xi, xii, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 17, 18, 19, 20, 21, 22, 29, 30, 32, 33, 34, 35, 36, 37, 40, 41, 42, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 58, 59, 63, 64, 65, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 83, 84, 85, 92, 96, 97, 98, 99, 100, 1

KD-ANN KD Approximate. 45, 46, 49, 64

PNN Previous Nearest Neighbor. x, xi, xii, 46, 49, 50, 63, 64, 65, 72, 73, 82, 83, 91, 92, 94, 95, 96, 97, 99, 100

pose position and orientation. iv, ix, 1, 2, 3, 55, 58, 92, 1

RMS root mean square. 46

ROI Region of Interest. xi, xii, 84, 85, 86, 91, 98, 99, 100

SPMD single program multiple data. 10, 20, 31

SVD Singular Value Decomposition. 10

USAF United States Air Force. 1

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
25–03–2021	Master's Thesis	June 2019 — Mar 2021

4. TITLE AND SUBTITLE

Accelerating Point Set Registration for Automated Aerial Refueling

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Ryan M. Raettig

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-21-M-075

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RQQC
Dan Schreiter
WPAFB OH 45433-7765
COMM 937-938-7765
Email: dan.schreiter@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RQQC

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of AAR is to control the tanker boom to safely refuel a receiving aircraft with no input or aid from the boom operator. To achieve this, the pose of the receiver realative to the tanker must be known. Point set registration is a fundamental issue used to estimate the relative pose of an object in an environment. However, it's likely a computational bottleneck of a vision processing pipeline. In addition, the matching of each sensed point with a closest truth point, nearest neighbor matching, is the most costly portion of the point set registration process. For this reason, this research focuses on speeding up the ICP algorithm and nearest neighbor algorithms. This research lays out novel nearest neighbor matching algorithms based on the Delaunay triangulation with a reduced cost compared to conventional nearest neighbor matching algorithms. The ICP algorithm is transformed into a massively parallel algorithm and mapped onto a vector processor to realize a speedup of approximately 2 orders of magnitude. Lastly, this thesis presents algorithmic and runtime analysis with augmented, virtual, and real experiements.

15. SUBJECT TERMS

Aerial Refueling, Computer Vision, Point Set Registration, Nearest Neighbor, Parallel Computing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Scott L. Nykl, AFIT/ENG
					19b. TELEPHONE NUMBER *(include area code)*
U	U	U	UU	127	(937) 255-3636, ext 4395; scott.nykl@afit.edu