PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

FAST LIDAR DATA REGISTRATION USING GPUs

Tesis para obtener el título profesional de Ingeniero Electrónico

AUTOR:

Carlos Enrique Huapaya Avalos

ASESOR:

César Alberto Carranza de la Cruz

Lima, Agosto, 2023

Informe de Similitud

Yo, **Cesar Alberto Carranza de la Cruz**, docente de la Facultad de **Ciencias e Ingeniería** de la Pontificia Universidad Católica del Perú, asesor de la tesis titulada **FAST LIDAR DATA REGISTRATION USING GPUs**, del autor **Carlos Enrique Huapaya Avalos**,

dejo constancia de lo siguiente:

- El mencionado documento tiene un índice de puntuación de similitud de 10 %. Así lo consigna el reporte de similitud emitido por el software *Turnitin* el 01/08/2023.
- He revisado con detalle dicho reporte y la Tesis o Trabajo de Suficiencia Profesional, y no se advierte indicios de plagio.
- Las citas a otros autores y sus respectivas referencias cumplen con las pautas académicas.

Lugar y fecha: Lima, 25 de agosto de 2023

Apellidos y nombres del asesor: Carranza de la Cruz, Cesar Alberto	
DNI: 09641576	Firma
ORCID: 0000-0003-1222-0118	

ET LUX IN TENEBRIS LUCET
MCMXVII

Resumen

En los últimos años, la llegada de las cámaras de profundidad de bajo costo y sensores LiDAR ha incentivado a las industrias a invertir en estas tecnologías, lo cual incluye también mayor interés en investigaciones sobre procesamiento digital de señales. En esta ocasión, la reconstrucción tridimensional de túneles mineros utilizando LiDARs y un robot de auto-navegación ha sido propuesta como proyecto de investigación, y el presente trabajo forma parte encargándose del alineamiento de nubes de puntos tridimensionales en tiempo real, un proceso que es más conocido como Registro de Nubes de Puntos. Existen muchos algoritmos que pueden resolver este problema, pero para el proyecto, el algoritmo solo necesita calcular la alineación fina y rígida. Al comparar los algoritmos de registro más avanzados, se encontró que el popular algoritmo ICP es el más adecuado para este caso debido a su alta robustez y eficiencia. Dentro de este algoritmo, se encuentran 3 pasos simples: relación, minimización y transformación, junto con una colección de variaciones de estos pasos que han sido desarrolladas a lo largo de las últimas décadas. Basándose en esto, en este trabajo se diseñó e implementó un algoritmo ICP paralelo en CPU y GPU. Además, las optimizaciones en recursos de memoria, ocupación de núcleos y el uso de la técnica de desenrollado de bucles para la implementación en GPU permiten que la implementación propuesta de ICP alcance un rendimiento 95 veces más rápido que implementaciones de CPU altamente optimizadas.

Abstract

In the last years, the advent of low-cost depth cameras and LiDAR sensors has encouraged industries to invest in these technologies, which includes research about digital signal processing. This time, 3D surfaces reconstructions of mining tunnels using LiDARs and a self-navigation robot have been proposed as a research project and this work is in charge of performing the alignment of data that come from the LiDARs in real-time, a process that is most known as 3D Point Cloud Registration. Many algorithms can solve this problem, but for the project, the algorithm only needs to compute for fine-rigid alignment. Comparing state-of-the-art registration algorithms, it is found that the popular ICP algorithm is the best suited for this case since its high robustness and efficiency. Inside this algorithm, there are 3 simple steps: matching, minimization, and transformation, and a diverse collection of variations of these steps that have been developed through the last decades. Based on this, a parallel ICP algorithm is designed and implemented in CPU and GPU. Moreover, optimizations in memory resources, cores occupancy, and the usage of the loop unrolling technique for the GPU implementation lead the proposed ICP implementation to reach a performance of 95 times faster than highly-optimized CPU implementations.

ET LUX IN TENEBRIS LUCET
MCMXVII

Dedicado con mucho cariño a mis padres, a mi hermano,
a toda mi familia y amigos que me han brindado su apoyo a lo largo de todos estos años.

General Index

Introducción 1

1 3D Model Reconstructions and the Point Cloud Registration Problem 2

1.1 Motivation . . . 2

1.2 Overview . . . 3

1.3 State of the Art . . . 4

1.3.1 3D Point Cloud Registration Algorithms . . . 4

1.3.2 Standard ICP Algorithm . . . 5

1.3.3 ICP Variants . . . 6

1.3.3.1 Selection . . . 7

1.3.3.2 Matching . . . 7

1.3.3.3 Error Metric . . . 8

1.3.4 ICP Biggest Drawback . . . 9

1.4 Graphic Processing Units (GPUs) . . . 9

1.5 Justification . . . 10

1.6 Objectives . . . 11

1.6.1 General Objective . . . 11

1.6.2 Specific Objectives . . . 11

2 Theoretical Framework of the ICP Algorithm 12

2.1 Complexity of Algorithms . . . 12

2.2 Least-Square Minimization . . . 14

2.3 Singular Value Decomposition (SVD) . . . 15

2.4 Principal Component Analysis (PCA) . . . 16

2.5 ICP Theory . . . 16

2.5.1 ICP Least-Square Model . . . 16

2.5.2 Matching . . . 17

2.5.2.1 Brute Force 17
2.5.2.2 K-D Tree 18
2.5.3 Error Metric and Minimization 19
2.5.3.1 Point-to-point Error Metric 20
2.5.3.2 Point-to-plane Error Metric 21
2.5.4 RMS Error Value 22
2.6 Parallel Programming in CUDA 22

3 Design and Implementation of the Parallel ICP Algorithm 25
3.1 Previous Considerations 25
3.1.1 Input Data 25
3.1.2 Data Grouping 25
3.2 CPU ICP Implementation 26
3.2.1 MKL Library 27
3.2.2 Sequential ICP Algorithm 27
3.3 GPU ICP: Design and Implementation 29
3.3.1 Receiving Data from the LiDAR OS1-16 29
3.3.2 Transforming Raw Data into Cartesian Coordinates 32
3.3.3 Getting Normals of Model Point Cloud Q using PCA 33
3.3.4 Finding the Correspondence using Brute Force 34
3.3.5 Minimizing the Error Metrics 35
3.3.6 Transforming the Source Point Cloud P 36
3.3.7 Finding the Current RMS Error Value 36
3.3.8 Validation 36

4 Results 37
4.1 Test Conditions 37
4.1.1 Implementation Description 37
4.1.2 Devices 38
4.1.3 Input Data 38
4.2 Computational Results 40
4.2.1 Time Complexity 40
4.2.2 Speedup 40
4.2.3 Running Time 41
4.2.4 Error 42

4.3 Optimizations 43
4.4 Testing the Optimized Parallel ICP Algorithm 44
4.5 Analysis of Results 45

Conclusions **47**

Recommendations and Future Work **48**

Bibliography **49**

Figure Index

1.1 Steps involved and the sequential flow in the standard ICP algorithm. 6

1.2 Comparison between the three most common matching ICP variants: Brute Force (standard ICP), K-D Tree, and Delaunay Triangulation using the surface $z = x^2 - y^2$ with 6400 points. 8

1.3 CPU vs GPU comparison in number of core processors [22]. 10

2.1 K-D tree for a set of 3D points. Based on [20]. 19

2.2 Graphic representation of a K-D Tree in 3 dimensions. This graphic was made using the tree in Figure 3. X, Y and Z planes are in light blue, red and green respectively. 19

2.3 CUDA programming model [28] 24

3.1 Two types of grouping point cloud data. 26

3.2 Sequential implementation flow diagram 27

3.3 Sequential ICP algorithm 28

3.4 Proposed solution model 1. 29

3.5 Proposed solution model 2. 30

3.6 LiDAR data storage 31

3.7 Data from LiDAR OS1-16 flow diagram. 32

3.8 Parallel algorithm for coordinate conversion. 33

3.9 Normal estimation flow diagram. 34

3.10 Parallel algorithm for correspondence. 35

3.11 Parallel algorithm for transformation. 36

4.1 Synthetic data. 38

4.2 Stanford Bunny dataset. 39

4.3 Real dataset from LiDAR OS1-16. 39

4.4 Time complexity. 40

4.5 Speedup 41
4.6 RMS error. 42
4.7 Matching optimization for 1 iteration. 44

ET LUX IN TENEBRIS LUCET
MCMXVII

Table Index

1.1 Comparison among three accurate registration algorithms 5

2.1 Examples of Big-O notation 13

4.1 Synthetic data running time for 1 iteration. 42
4.2 Synthetic data running time for all iterations. 42
4.3 Optimized ICP results. 45

Acronyms

1. **API**: Application programming Interface
2. **ASCII**: American Standard Code for Information Interchange
3. **BLAS**: Basic Linear Algebra Subprograms
4. **CPU**: Central Processing Unit
5. **CUBLAS**: CUDA Basic Linear Algebra Subprograms
6. **CUDA**: Compute Unified Device Architecture
7. **CPD**: Coherent Point Drift
8. **DO**: Discriminative Optimization
9. **DOF**: Degrees of freedom
10. **FFT**: Fast Fourier Transform
11. **FLANN**: Fast Library for Approximate Nearest Neighbors
12. **GPU**: Graphics Processing Unit
13. **ICP**: Iterative Closest Point
14. **KNN**: K-Nearest Neighbors
15. **LAPACK**: Linear Algebra Package
16. **LIDAR**: Light Detection And Ranging Sensor
17. **MKL**: Math Kernel Library
18. **NNS**: Nearest Neighbor Searching
19. **PCA**: Principal Component Analysis
20. **RAM**: Random-Access Memory
21. **RMS**: Root Mean Squared
22. **RTX**: Ray Tracing Texel eXtreme
23. **SVD**: Singular Value Decomposition

Introduction

Light Detection and Ranging (LiDAR) sensors' popularity has raised through the years. Nowadays, it is easy to find lots of applications of this technology in different fields, such as medicine, topography, and mining, among others, because of the fact that these sensors can represent objects or places in 3D computational models with great accuracy. One of the reasons behind its well-acceptance is that LiDARs have a very simple principle: they use light waves from lasers to measure distances to the target and then these measurements are stored forming point clouds. However, 3D applications require LiDAR sensors to be in movement or to take several shots from different angles producing many point clouds that are sorted in various positions. So, a process is required to map the clouds where they belong. This problem is known as Point Cloud Registration.

In the scientific community, a great branch of research focuses on finding the best way to solve this registration problem using methods and algorithms, where one of the most famous is the Iterative Closest Point (ICP) algorithm. Its robustness, efficiency, simplicity, and powerful variants made this algorithm to be the best choice when aligning point clouds.

Nevertheless, most of the research aims at improving the ICP in serial implementations leaving aside the possibility to parallelize it. This work is dedicated to exploring parallelization methods for the ICP algorithm, due to the necessity of real-time applications.

Chapter 1

3D Model Reconstructions and the Point Cloud Registration Problem

This chapter describes some of the safety problems of the mining industry in Peru and one of the solutions for these issues using 3D model reconstruction techniques with LiDARs. Later on, the Point Cloud Registration problem, which is a fundamental part of 3D reconstructions, is defined, and state-of-the-art solutions for it are listed. Among them, the ICP algorithm comes out to be the most powerful and so state-of-the-art research of this algorithm is placed. Finally, the justification and objectives are presented at the end of this chapter.

1.1 Motivation

The mining industry is a very important sector in Peru. According to [26], mining represents 10% of the Gross National Product (GNP), and 60% of the exportations value of this country.

There are two ways to extract mineral material in Peru, which are opencast and underground or tunnel mining. Due to their ways of operation, tunnel mining is more dangerous for workers than opencast, because when operating underground, workers have to inspect or extract material, whereas opencast mining only uses explosives to triturate minerals and trucks to transport them. Inside tunnels, physical and intoxication risks are the most fatal and thus the ones that cause more accidents in this industry. In fact, according to [1], accidents inside mining tunnels, such as rock falls, intoxications, and contact with electrical energy represents 10 of the 23 mortal accidents that occurred in the sector of medium- and large-scale mining industry in 2018. Besides this security aspect, it is also important for mining companies to measure the volume of the mineral material that has been extracted when building a tunnel. This activity can be accomplished with trucks carrying the minerals and then weighted them using a scale, but this process is imprecise and inefficient [2].

To improve the inspection and volume of material extracted processes, the application of a self-navigation robot equipped with a LiDAR sensor and an efficient point cloud registration to obtain 3D models of mining tunnels is vital. With this approach, it would not be necessary to send workers to inspect tunnels and the volume calculations can be accomplished highly efficiently and accurately.

1.2 Overview

A 3D model representation is a computational representation of an object or scene in its three dimensions. To construct this model, 3D scanning techniques commonly require 2D data or images of the object from different angles, which can be taken from sensors or cameras using certain techniques.

On the one hand, optical 3D measurement techniques use laser sensors and are divided into 2 groups: time-of-flight and triangulation [34]. The time-to-flight technique consists in projecting a beam of light onto an object and calculating the distance to it by measuring the round-trip time of the beam with a sensor. Triangulation technique also projects a laser onto an object but in this case, depth is measured by the deformation of the laser, using angles and the cosine law. As a matter of fact, triangulation is mostly used when short distances are going to be measured whereas time-to-flight is more useful for mid and long distances. On the other hand, cameras offer some other techniques, such as photogrammetry, stereo vision, and structured light, where the working principle is getting 2D shots of an object with a camera or arrangement of cameras and then aligning them to construct the 3D model.

In the last decade, lots of 3D reconstruction systems combining described techniques and including different models and types appeared. Each of them is characterized by a certain accuracy, efficiency, and application. For instance, Structured Light Scanners (SLSs), which are usually equipped with a projector or a laser, one or more cameras, and a computer, are greatly accurate when the object to scan is near the system [13]; KINECT, a low-cost Microsoft technology that uses the time-to-flight technique and a camera for detecting color and tracking human movements, is suitable for representing indoor and outdoor scenes [29], [11]; and Light Detecting and Ranging (LiDAR), a technology that purely works with the time-to-flight technique, are systems that can almost match every 3D application, due to the fact that it offers a large range of system products.

Whether detecting from the air or the ground, LiDAR systems are able to reach great performance and accuracy. Applications are diverse, from creating 3D reconstructions of mining trucks loads with an error of 4.41% [2] to generating reconstructions of large highways, giving road features with a 5% error on average approximately [6], and huge topographic models, resulting in an average error of a bit more than 10 cm [18]. These results indicate that LiDARs are effective systems with high application potential.

At the same time that LiDARs take measurements, registers of it are saving all the measured points. The groups of these points are called point clouds. After collecting different point clouds of an object, the processing and alignment of the points are required. A well-designed aligning step is vital for reliable results and a fast computation of it is a must if the application is real-time. This last aspect will be the study of this work.

1.3 State of the Art

1.3.1 3D Point Cloud Registration Algorithms

Various algorithms have tried to make the aligning process simple, effective, and robust so that they can greatly perform different applications. They can be classified into different branches [7]. To mention some of them, according to the registration process, they can be separated into coarse and fine registration; according to the registration feature, they can be split into global and local registration; according to the deformation, they can be partitioned into rigid and non-rigid registration.

For the sake of clarity, this subsection is only focused on studying fine-alignment and non-rigid algorithms, which are registrations that only converge when the point clouds are close to each other and have only passed through rigid transformations (rotations and translations). Here, we can find the Coherent Point Drift (CPD), the Discriminative Optimization (DO), and the Iterative Closest Point (ICP), to mention the most known. Table 1.1 shows a comparison between these three algorithms. The results are obtained from experiments evaluated in [7].

On the one hand, these results show that the CPD algorithm performs the registration process quite worse than the other two algorithms, in the three aspects. On the other hand, the ICP and DO algorithms are both very robust and fast, but the ICP seems to be more sensitive to noise than the DO algorithm. However, if using a large number of points, it can be demonstrated that ICP has a much slower computation time than DO. So, for requirements of rapidness in the application and since the registration algorithm is needed for aligning two clouds that are very close to each other (in a way that noise effect does not influence the alignment), the ICP is the selected registration algorithm for this work.

Table 1.1: Comparison among three accurate registration algorithms

Registration Method	**CPD**	**DO**	**ICP**
Noise Effect	low	low	high
Operating Speed	slow	fast	fast
Robustness	bad	good	good

1.3.2 Standard ICP Algorithm

In its basic form, the ICP algorithm focuses on solving a Least-Square problem by iteratively minimizing the error metric function for two given point cloud datasets: source and target. The goal is to map the source to the target in such a way that in every iteration, they are brought closer. If conditions are met, after a number of iterations the ICP algorithm converges and so the output of this algorithm is the transformed source point cloud and the parameters associated with this transformation, i.e., rotation and translation matrices.

The standard ICP algorithm can be disintegrated in 3 steps as shown in figure 1, based on [13]. In the beginning, the matching step finds the specific correspondence of every point from the data to the model point cloud. After this, the minimization or pose estimation step calculates the rotation and translation that best align the point clouds. If the algorithm has not converged until this last step, then it has to come back to the first step and start again.

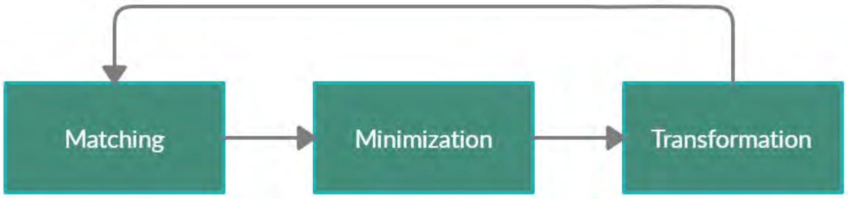


Figure 1.1: Steps involved and the sequential flow in the standard ICP algorithm.

1.3.3 ICP Variants

In [30], a taxonomy of the ICP algorithm is presented. They can be in or added to the processes shown in figure 1 and are the following ones:

- Selection: Data point cloud can be preprocessed to optimize the process of matching. This can be done by grouping the data cloud according to their normals (using PCA), curvature, or color.
- Matching: As it was mentioned, data points need to find a correspondence point in the model data set. This can be achieved using Nearest Neighbors Searching (NNS) techniques, such as brute force (Euclidean distance), KD-Trees, Delaunay Triangulation, and Normal Shooting, just to mention a few.
- Weighting: From the matching process, each matched data point has a different percentage of compatibility. This means that each point can have a weight, that can multiply its own error metric. Normal, curvature and distance compatibility can be calculated with certain mathematic expressions.
- Rejecting: In some cases, a number of points can be irrelevant for processing. This variant can work with two kinds of rejections: statistical, in which a certain percentage of the total points are rejected, and edge rejection, where points in the edge are rejected.
- Error metric: As it was mentioned, an error metric function model is defined due to the Least- Square Minimization problem. The most common error metrics are the point-to-point, point- to-plane and plane-to-plane minimization models.
- Minimizing: Computation of the rotation and translation variables of the data cloud using the minimized parameters.

The presented variants itemize almost every process used in the state-of-the-art improved ICP algorithms, even if they are recent. Also, because of the great number of options given in this taxonomy, lots of combinations can be made and, as this algorithm has been obtaining popularity, various studies are trying to find the most robust and efficient modified ICP algorithm. In the following subsections, some of the most relevant variants are presented. The variants that are not presented here may have little incidence in most of the cases, so they will not be covered. Moreover, in most cases modified ICP algorithms reduced these variants to their most simplified form.

1.3.3.1 Selection

Preprocessing is a good practice for real-time applications and so is a good initial alignment when target and source point clouds are separated by large rotations matrices or large translation vectors, or if a high ratio of convergence is needed when data sets are considerably near to each other. In [4], an objective is to find an improvement of the ICP algorithm and the selected one uses a subsampling based on the SVD method, a variant that makes this alternative a good option for real-time scenarios given shown results. In [3], it's proposed that a good initial guess for the alignment based on the PCA method yields good results. In [27], it is shown that random subsampling, which means randomly reducing the source point cloud, also improves the algorithm's performance.

1.3.3.2 Matching

In [8], an improved ICP algorithm is presented and it uses a voxel grid method for selection and rejection, a FLANN (Fast Library for Approximate Nearest Neighbors) algorithm, which is based on K-D Trees for matching, and point-to-plane for the error metric. Using the bunny and happy stand data sets, this work approaches better alignments than standard ICP and, in terms of time, it's four times faster than the standard algorithm. In [16], K-D Trees play an important role in the improved algorithm, demonstrating that this matching variant has good potential. Experiments are executed for creating a 3D representation of a tree with LiDARs. A high number of points ($\approx 200k$) is used and results show that this improvement of the ICP algorithm reaches a 10 times faster computation than the standard algorithm including a better final error metric result. In [35], a comparison of the three most common NNS techniques (matching) is made and results show that K-D Trees are way computationally faster than the other 2, which are the ones used in the standard ICP, well known as brute force, and Delaunay Triangulation. Figure 1.2 reasserts this last result, by also comparing these three techniques using a MATLAB code, taken from [14]. It can clearly see that every matching algorithm follows the same error metric curve and thus has the same number of iterations, but the big difference between them is how much time it takes every method to converge. K-D Trees surpass the other two methods with a convergence time of 0.43

seconds.

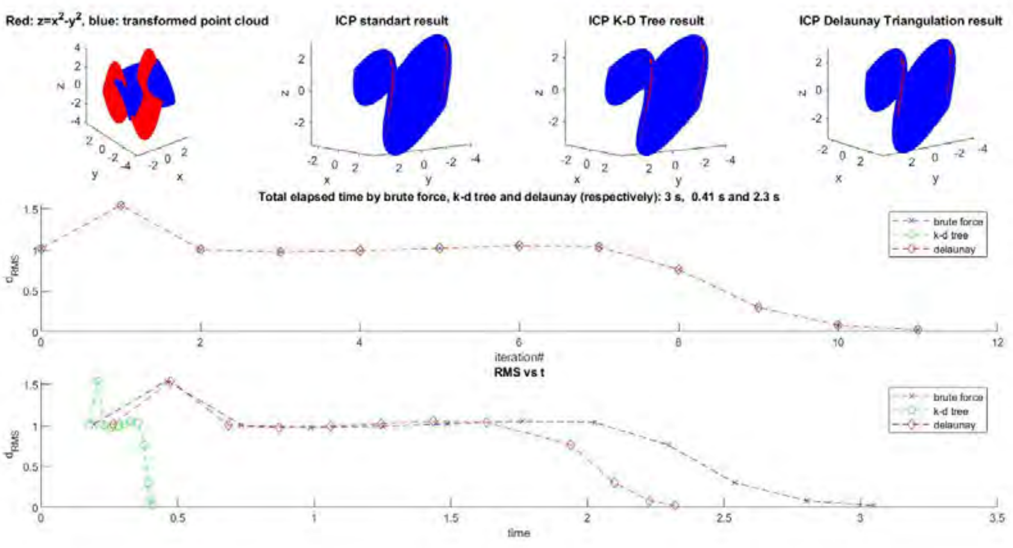


Figure 1.2: Comparison between the three most common matching ICP variants: Brute Force (standard ICP), K-D Tree, and Delaunay Triangulation using the surface $z = x^2 - y^2$ with 6400 points.

All of these experiments reveal that K-D Trees are powerful structures and can improve the ICP algorithm. However, the huge disadvantage of K-D Trees is that they are very computational expensive, which means that a big percentage of the total ICP algorithm execution time goes on K-D Trees. The work done in [19], holds this last statement showing that 92% of the total ICP algorithm execution time is for creating the K-D Tree and processing the NNS working with a CPU.

1.3.3.3 Error Metric

Most of the presented mentioned research concludes that a variant in the error metric is an important consideration for creating a well-design ICP algorithm. A generalized ICP is described in [31] This paper demonstrates, mathematically, that point-to-point and point-to-plane error metric expressions come from a general plane-to-plane error metric model. Generalized ICP seems to have a better performance than point-to-plane and standard ICP error metric models. However, due to the generalized ICP complicated model, most of the modified ICP algorithms use point-to-plane error metric models.

1.3.4 ICP Biggest Drawback

By this point, it is clear that the ICP algorithm can yield high-accuracy performances in choosing the right variants of the algorithm. Nonetheless, the biggest ICP defect is found when a locally optimal solution problem is presented. According to [15], the ICP algorithm can easily drop on a local optimal minimum, which is a wrong solution, for some applications. To solve this problem, before going through the ICP algorithm, a global registration of the data sets can be made. This registration focuses on two aspects: feature descriptors (description of a point context) and search strategies for correspondences (strategy for point-to-point correspondence).

However, for the application of this work, this verification is not necessarily due to the fact that the points clouds are always quite close to each other and so the ICP only computes fine alignments.

1.4 Graphic Processing Units (GPUs)

A GPU (Graphic Processing Unit) is a piece of hardware designed principally for image processing applications and float operations computing. As a matter of fact, due to the powerfulness of GPUs, the number of their applications has been increasing really fast, and now it is easy to find lots of industries using this equipment, such as game development, healthcare, supercomputing, telecommunications, retail and so on [21].

GPUs are quite different from CPUs. Actually, there are a lot of facts to analyze when comparing them, but maybe the most important ones are latency and throughput. Due to the fact that a CPU is designed with few cores and the GPU with hundreds of them as shown in Figure 3, CPUs are good at optimizing for latency because they make their cores work harder and faster than the ones from a GPU whereas GPUs are good at optimizing for throughput since they are equipped with lots of cores. This is why the cores from a CPU are very powerful and complex although the ones from a GPU are less powerful and simpler. Furthermore, the main reason why GPUs are selected for solving high-computational tasks is that they save considerable amounts of energy that CPUs would need to complete them.

In order to optimize resources CPUs and GPUs can work together. On one side CPUs can offer to be the head of programs and the ones preferred to run serial tasks, and on the other side, GPUs can be the devices to execute programs that adapt better for parallelization.

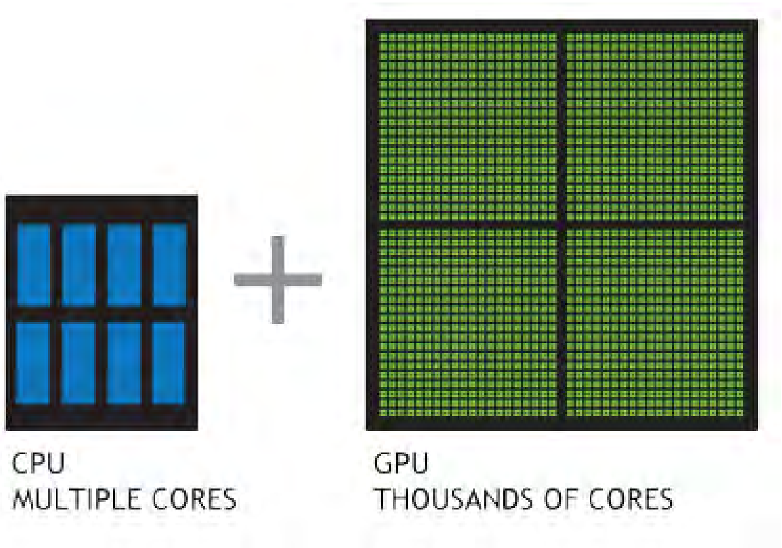


Figure 1.3: CPU vs GPU comparison in number of core processors [22].

1.5 Justification

State-of-the-art research suggests that there are various ICP algorithms based on the presented ICP variants. Depending on the application, combinations of the variants offer a certain efficiency and running time. Therefore, the election of the ICP algorithm is then a good choice to perform the 3D registration process of the mentioned application.

Nevertheless, the design and choice of the right ICP algorithm contribute to the first half of this work. The other half comes from the implementation. Here, sequential implementations of the ICP algorithm can highly delay the alignment when working with big quantities of data, whereas parallel designs and implementations present the option to greatly accelerate the algorithm's computation. Thus, implementing a parallelized ICP algorithm can outperform any other point cloud registration algorithm with the right implementation.

GPUs fit perfectly with the last-mentioned requirement about implementation. They have been evolving through the years to offer technology that can enormously reduce computation times when using big amounts of data. Then, running the parallel ICP algorithm with some of the latest GPUs will surpass, in terms of running time, any other ICP algorithm, whether the implementation is sequential or parallel.

1.6 Objectives

1.6.1 General Objective

Create a new parallel ICP algorithm and implement it on a GPU for fast computation of LiDAR data registration.

1.6.2 Specific Objectives

- Analyze different ICP algorithms, evaluate their theoretical performance and select the best suited for parallelization.
- Design and implement a new parallel ICP algorithm.
- Optimize the implementation to improve the running time.
- Evaluate the new algorithm performance in terms of running time and compare it with other solutions.

Chapter 2

Theoretical Framework of the ICP Algorithm

In this chapter, theoretical concepts about the ICP algorithm are covered. Furthermore, criteria to select algorithms are approached in order to have tools to choose the best possible algorithms for each function of the implemented algorithm.

2.1 Complexity of Algorithms

When solving any problem, different alternatives can come to mind. Expressing this from a computational view, those alternatives turn out to be algorithms or structures. If it is supposed that all of them can solve the problem with exactly the same result, the question would be which one can resolve the problem more efficiently.

To address this matter, the complexity of algorithms defines multiple criteria to give quantities of an algorithm's performance. These criteria basically focused on finding correspondence between a variable to analyze in relation to the number of entries of the algorithm. From the variety of metrics, time and memory are the most important since they offer quantities about running time and memory usage. In fact, when relating them with the number of inputs, the resulting functions are called time and space complexity.

Comparing algorithms can be challenging using the exact complexity functions. This is why it is necessary to use a sort of standard representation. The Big-O family offers a group of standard notations than can be very useful when comparing algorithms. Among the best-known of this family are Big-O, Big-Ω, and Big-Θ and they represent the worst, best, and average scenario in an algorithm, respectively. As a matter of fact, according to their definition, a complexity function is approximated by an upper bound function when using Big-O, by a lower bound function when using Big-Ω, and by an upper and lower function when using Big-Θ [36]. Some of the examples of the most known approximations for time complexity functions are presented in Table 2.1. The complexity of the functions in the table goes from the lowest to the highest complexity as coming down in the table.

Table 2.1: Examples of Big-O notation

Big-O function	**Function Name**
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n * \log n)$	Linearithmic time
$O(n^2)$	Quadratic time
$O(n^c)$	Polynomial time
$O(2^n)$	Exponential time

Although these notations offer a good reference to compare algorithms, it is a good practice to also consider real complexity functions or at least consider some values of running time or usage of memory for certain input sizes. This is because, when comparing algorithms, it might happen that a low-complexity Big-O function has a greater running time or more usage of memory than a high-complexity function and the explanation for this is mainly the error range that Big-O notation leaves while doing approximations.

Because the presented work's principal objective is to implement a fast computation of the ICP algorithm, time complexity is the selected criterion to compare algorithms. Also, since the Big-O notation expresses the worst scenario in the algorithm, time complexity using the Big-O notation will represent, approximately, the slowest running times in relation to the input sizes.

2.2 Least-Square Minimization

Least-Square minimization method is quite famous when fitting a line to a group of scattered points with the minimum possible error. Equation 2.1 rules this method for best fitting lines. The function f is usually called the cost function and m and b are called the arguments or the degrees of freedom (DOF) of f [4]. The key is to find values for m and b in such a way that sum of the squared errors is as low as possible.

$$f(m,b) = \min_{m,b} \sum_{i=1}^{n} (y_i - \hat{y})^2 \tag{2.1}$$

Where:

- y_i represents the scattered points.
- $\hat{y} = m \cdot x_i + b$ represents the points in the best fitting line.
- n is the number of points.

The best fitting line is just one of the innumerable applications the Least-Square method has. In general, this technique is used when an ill-posed problem is proposed, i.e. when the number of parameters or DOF is less than the number of equations. Thus, the general equation of the Least-Square Method is described in equation 2.2. Furthermore, depending on what it is being minimized, the Least-Square can be linear or non-linear. The error e_i gives a way to identify whether the method is of one type or the other. If this metric changes linearly with respect to the parameters, then the Least-Square is linear, in other cases, the method is called non-linear.

$$f(x_1, x_2, \ldots, x_k) = \min_{x_1, x_2, \ldots, x_k} \sum_{i=1}^{n} \|e_i\|^2 \tag{2.2}$$

Where:

- e_i error at the i_{th} point.
- $x_1, x_2, \ldots, x_k$ are the parameters or the least squares solution.
- n is the number of points.

Since f is an $\mathbb{R}^n \rightarrow \mathbb{R}$ function, there are diverse ways to optimize the Least-Square problem. Gradient-based methods [36], Newton and Quasi-Newton methods [32], Singular Value Decomposition (SVD) [38], and Global Optimizers [36] are some the most known.

2.3 Singular Value Decomposition (SVD)

As it was mentioned, SVD is used to optimize Least-Squares. Actually, the ICP Least-Square model, which will be presented in subsection 2.5.1, has a demonstrated optimization using this method [4].

Basically SVD decomposes a matrix into three other matrices, and its definition is supported by the theory of diagonalization of a matrix [12]. Defining M as a real $n \times m$ matrix, equation 2.3 shows the SVD for M. It is important to mention that U and V are orthogonal matrices, i.e. $U^T = U^{-1}$ and $V^T = V^{-1}$, and Σ is a diagonal one. Also, the diagonal values of Σ are ordered by importance, having the greatest value to the left and the smallest to the right.

$$M = U\Sigma V^T \tag{2.3}$$

Where:

- U and V are orthogonal matrices of sizes $m \times m$ and $n \times n$ respectively.
- Σ is a diagonal $m \times n$ matrix.

Now, in order to understand what the decomposition matrices mean, the SVD of a correlation matrix, i.e. a matrix multiplied by its transpose, is presented down below. Matrix M from equation 2.3 is used.

$$\begin{aligned} MM^T &= (U\Sigma V^T)(V\Sigma^T U^T) \\ MM^T &= U\Sigma^2 U^T \\ &\text{or} \\ MM^T &= V\Sigma^2 V^T \end{aligned} \tag{2.4}$$

Both represent an eigenvector decomposition. Using the last expression, V and Σ^2 symbolize the M correlation eigenvectors and eigenvalues, respectively. An equivalent conclusion can be extracted for the other correlation matrix MM^T. Finally, it is worth mentioning that the matrix M uses to have more much more rows than columns or vice versa. Depending on how it is defined, MM^T will be much bigger or much smaller than $M^T M$. This is important to consider in order to have the dimensions of U,Σ and V as small as possible.

2.4 Principal Component Analysis (PCA)

The principal component analysis offers a way to represent variance in higher dimensional data. This method will be useful when needing the normals of a group of points, more specifically, for a certain point and a close neighborhood to it.

PCA holds its definition with the covariance matrix of a group of points. For a given set of points Q, the covariance matrix is detailed in equation 2.5 [35]. This covariance matrix shows how the data deviate from the centroid. Additionally, and this is where the PCA contribution is, the covariance matrix keeps data about in which direction points deviate from the centroid. The method mentions that the eigenvalues of the covariance matrix quantize deviations from the centroid in the directions of the eigenvectors associate with them. Moreover, these eigenvectors form an orthogonal basis and are classified as the first, second, and third principal components of the covariance matrix. It comes to happen that when points are coplanar or they are close to having this characteristic, the third principal component gives an estimation for the normal of the formed plane [9]. Actually, what it will be used when needing normal estimations of a point cloud is basically form a group of points Q with a close neighborhood to each point.

$$\text{Cov}(Q) = \frac{1}{n}\sum_{i=1}^{n}(q_i - \bar{q}) \cdot (q_i - \bar{q})^T \tag{2.5}$$

Where:

- $Q = \{q_1, q_2, \ldots, q_n\}$, $q_i \in \mathbb{R}^3$.
- $\bar{q}$ is the centroid of Q.
- n is the number of points.

2.5 ICP Theory

In this subsection, concepts that directly involves the ICP algorithm are covered.

2.5.1 ICP Least-Square Model

In 1991, Y. Chen and G. Medioni proposed a method to model a 3D object by using the registration of its multiple range images [37] and one year later P. Besl and N. Mckay presented an approach to align curves and set of points [5] .This method is called the Iterative Closest Point (ICP). As described in chapter one, the objective is to align two point clouds, data and model, using transformations. In other words, and to give them reference, data point cloud is transformed in order to be, as close as possible, equal to the model point cloud. Although, rigid and non-rigid transformations can be used to reach the mentioned objective, rigid ones, i.e. rotations and translations, will be used due to of the application of this work. Equation 2.5 shows the ICP Least- Square model to align a data point cloud, P, with a model point cloud, Q. It essentially expresses the minimization of the sum of the Euclidean distances of P and Q point clouds by finding the best possible $\hat{R}$ matrix and $\hat{t}$ vector. Applying these transformations to the data point cloud, P, will bring it closer to the model point cloud, Q. However, the ICP process does not end there: it iterates the minimization shown in equation 2.6 until the two clouds are close enough. Criterions about error will be discussed in section 2.6. Finally, to make referring easier, notation of the variables declared in this equation will be used in the rest of this chapter.

$$\hat{R}, \hat{t} = \min_{R,t} \sum_{i=1}^{n} \|(Rp_i + t) - q_i\| \tag{2.6}$$

Where:

- $P = \{p_1, p_2, \ldots, p_n\}$, $p_i \in \mathbb{R}^3$.
- $Q = \{q_1, q_2, \ldots, q_m\}$, $q_i \in \mathbb{R}^3$.
- $R \in \mathbb{R}^{3\times 3}$ and $t \in \mathbb{R}^3$

2.5.2 Matching

Due to the necessity of correspondence between the point clouds in equation 2.5, matching techniques are required. What the ICP algorithm propose is to find the closest points of P in Q. Since this process affects quite critically to the running time of the entire ICP algorithm [19] an efficient matching algorithm must be used. Nearest Neighbor Searching (NNS) meets this requirement with a variety of algorithms associated to it. In the rest of this subsection, notions of some of the most relevant NNS algorithms are presented.

2.5.2.1 Brute Force

A simple way to find the correspondence between the two set of points is comparing each point of P with all of the points from the model cloud using the Euclidean distance. Equation 2.7 sums up mathematically the previous statement and the correspondence problem is solved by detecting the points q_j where the distance is minimum.

$$d_i = \min \sum_{j=1}^{m} \|p_i - q_j\|^2 \tag{2.7}$$

Where:

- $i = 1, 2, 3, \ldots, n$.
- d_i is the minimum distance from p_i to the point cloud Q.

As it is expected, because of its naive approach, brute force could be more efficient in terms of running time. In fact, for each iteration in the ICP algorithm, the time complexity is $O(n * m)$ [22], which corresponds to a linear correspondence of the time with respect to the size of the two point clouds. However, one advantage that brute force has is that it does not need preprocessing. This means that the only thing brute force needs to calculate the correspondence is the coordinates of the two point clouds.

2.5.2.2 K-D Tree

This algorithm creates a binary tree that represents boxes graphically and it is called K-D Tree because its definition is valid for k dimensions, where $k \in \mathbb{Z}^+$. The essence of K-D Trees for this application is to construct the boxes separating the space, defined by a point cloud, in halves [20].

Figure 2.1 shows a K-D Tree for a set of eight random points $X = \{A = (1, 0, 2); B = (10, 3, 1); C = (4, 7, 7); D = (9, 11, 3); E = (3, 5, 8); F = (12, 1, 10); G = (-4, 5, 11); H = (-2, 2, 6)\}$. Every point in 3D space is chosen in such a way that it divides the space into two halves by using planes. Each plane is selected with the median of each coordinate. For example, at the root of this tree, the x-coordinate median of all the points was selected and so the point E and the plane $x = 3$ make the first division. After that, two groups of points are formed: one to the left of the $x = 3$ plane and one to the right of it. Now, in each group y-coordinate median makes the second division and that is $y = 2$ for the left and $y = 3$ for the right. These planes separate points in the other two groups and then the z-coordinate oversees the next divisions, up and down. If having more points, as in this case, the logic for keeping separating in halves starts at the x-coordinate and the explained process is applied until each point has a plane passing through it.

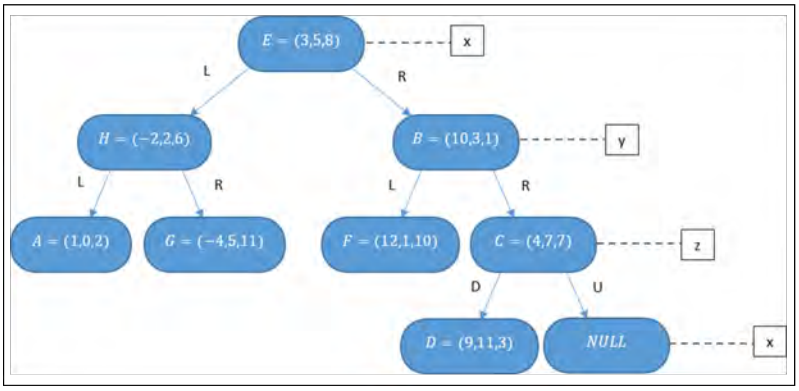


Figure 2.1: K-D tree for a set of 3D points. Based on [20].

Figure 2.2 shows the last example K-D tree graphic representation from two views. As it was mentioned, each point has a plane passing through it and the union of those plane represent the also mentioned boxes.

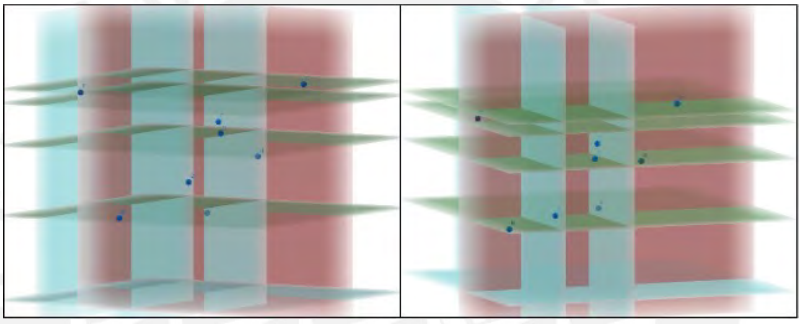


Figure 2.2: Graphic representation of a K-D Tree in 3 dimensions. This graphic was made using the tree in Figure 3. X, Y and Z planes are in light blue, red and green respectively.

For the application of this work, a K-D tree will be created for the model point cloud (using Q points), this is preprocessing, and data point cloud will search nearest neighbors into this object. This means that the tree is only created once with a time complexity of $O(m * \log m)$ and the only part that repeats through iterations is the nearest neighbor searching of data point cloud (P points). As a matter of fact, for a single iteration in the ICP, this last process has a time complexity of $O(n * \log m)$ [22]. Making a comparison between K-D tree and brute force, it can clearly see that, even considering their preprocessing time, K-D trees have a low time complexity than brute force.

2.5.3 Error Metric and Minimization

A model for the standard ICP algorithm error metric was presented in equation 2.6. However, there are other existing metrics: there are more types of them performing slower running times than the standard algorithm [31]. In this subsection, a common error metric variant is presented.

Furthermore, minimizations are shown to use them in the implementation of the ICP algorithm.

2.5.3.1 Point-to-point Error Metric

Equation 2.6 suggests that the error between data and the model point cloud is calculated using the average of the Euclidean distances. This is called the point-to-point error metric and equation 2.6 is the objective function related to it.

According to this definition, minimization can be found using linear algebra, and more specifically SVD. The following steps solve this optimization problem. They are extracted from [13] and the demonstration can be found in the same document.

1. Calculate the centroids of P and Q.

$$\begin{aligned} \bar{p} &= \frac{1}{n}\sum_{i=1}^{n} p_i \\ \bar{q} &= \frac{1}{m}\sum_{i=1}^{n} q_i \end{aligned} \tag{2.8}$$

2. Calculate the deviations of P and Q points from the centroid.

$$\begin{aligned} p_i' &= p_i - \bar{p},\ i = \{1, 2, \ldots, n\} \\ q_i' &= q_i - \bar{q},\ i = \{1, 2, \ldots, m\} \end{aligned} \tag{2.9}$$

3. Compose new point clouds using the deviations of P and Q. These new point clouds will be called P' and Q'. Using the last point clouds, calculate the correlation between them.

$$W = Q \cdot P^T \tag{2.10}$$

4. Apply SVD to W.

$$W = U\Sigma V^T \tag{2.11}$$

5. Calculate the rotation matrix R and the translation vector t that minimizes the error metric.

$$\begin{aligned} R &= VU^T \\ t &= \bar{q} - R\bar{p} \end{aligned} \tag{2.12}$$

2.5.3.2 Point-to-plane Error Metric

One of the most used error metric variants of the ICP is the point-to-plane error metric. Most of the research cited in this work has used it. This preference over the point-to-point metric is mainly because the point-to-plane error metric is much less sensitive to noise and since it causes the ICP algorithm converges in fewer iterations [4].

Equation 2.13 presents the model of the point-to-plane objective function. As it is shown, one element was added to the point-to-point error metric: normals of the model point cloud, Q, which can be found using PCA. These normals are being multiplied for the point-to-point error metric using the dot product. Taking into consideration that the normals are unitary, the point-to-plane error metric symbolizes the scalar projection of the vector $\vec{v_i}$ onto the normal associated to the point q_i. To interpret this metric, one can say that the scalar projection grants the possibility to the data point cloud, P, for finding an error in a whole plane instead of having the error attached to a point of the model cloud, which is the case of the point-to-point error metric. Furthermore, this interpretation is the reason why point-to-plane is more robust against noise than the point-to-point error metric. However, point-to-plane assumes that the points are locally coplanar in order to get the mentioned normals. Experiments will help to decide if the point clouds need a preprocessing step for converting close neighborhoods of points into coplanar points.

$$\hat{R}, \hat{t} = \min_{R,t} \sum_{i=1}^{n} \|[(Rp_i + t) - q_i] \cdot \vec{n_i}\|^2 \tag{2.13}$$

Where:

- $\vec{n_i}$ are the normals associated for each point of the model point cloud Q.
- $\vec{v_i} = (Rp_i + t) - q_i$ is the vector traced from the point $(Rp_i + t)$ to q_i.

A way to minimize this optimization problem is given in the following steps, taken from [13]:

1. Transform the rotation matrix R into the linearized matrix R_L as shown in the following

expression.

$$
\begin{aligned}
&R = R_x R_y R_z \\
&R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{pmatrix} \begin{pmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{pmatrix} \begin{pmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&R = \begin{pmatrix} 1 & -\theta_z & \theta_y \\ \theta_z & 1 & -\theta_x \\ -\theta_y & \theta_x & 1 \end{pmatrix}
\end{aligned}
\tag{2.14}
$$

This linearization is considering that $\cos\theta \approx 1$ and $\sin\theta \approx \theta$ when θ is small. For the application of this work, rotations are expected to be small which means that θ_x,θ_y,and θ_z take small values and so the linear approximation can be applied correctly without affecting significantly to the results.

2. Using R_L, differentiate the equation 2.13 with respect to θ_x,θ_y, and θ_z and the components of the translation vector T_x,T_y, and T_z and equal the result to 0.

3. As a result of differentiating, a system of linear equations comes up. So this last step is simply to solve the system of linear equations using standard methods, as shown in [17].

2.5.4 RMS Error Value

To find the RMS error value, which is the metric that measures how close the two point clouds are by normalizing the sum of errors of each point, this work is using equation 2.15. Notice that P and Q points are attached to the same sub-index i, which means that this q are the correspondence points of p found in the matching step.

$$e_{\text{RMS}} = \sqrt{\frac{\sum_{i=1}^{n} d_i^2}{N}} \tag{2.15}$$

Where:

- $d_i = \sqrt{p_i - q_i} = \sqrt{(x_{p_i} - x_{q_i})^2 + (y_{p_i} - y_{q_i})^2 + (z_{p_i} - z_{q_i})^2}$

2.6 Parallel Programming in CUDA

CUDA (Compute Unified Device Architecture) [28] is a parallel computing platform developed by NVIDIA to enable developers to harness the power of GPUs for accelerating computations.

By providing extensions to popular programming languages such as C, C++, and Fortran, CUDA allows developers to write parallel programs in a familiar environment. At the core of CUDA's programming model are kernel functions, which are small pieces of code that can be configured to run in parallel on the GPU. These functions express parallelism using blocks and threads, where blocks are essentially containers for threads, and threads are independent paths of execution through the code. Each thread runs the same code but with different data inputs, allowing for massive parallelism. Typically, thousands of threads are launched for parallel codes, depending on the application and the capabilities of the GPU.

The key advantage of CUDA is its ability to exploit the parallel processing power of GPUs, which have hundreds or thousands of cores that can work together to solve complex problems much faster than traditional CPUs. This makes CUDA particularly useful for computationally intensive tasks, such as simulations, machine learning, image processing, and digital signal processing, where the ability to process large amounts of data in parallel can significantly speed up computation times. CUDA is also highly flexible, allowing developers to customize kernel functions to optimize performance for specific hardware configurations and computational workloads.

The CUDA programming model is placed in Figure 2.3. At a high level, CUDA architectures work by breaking down computational tasks into smaller units of work that can be executed in parallel on the GPU. To achieve this, CUDA uses a programming model that is based on grids, blocks, and threads. A grid is a two or three-dimensional array of blocks, where each block is a group of threads. The grid defines the overall dimensions of the computation and is typically set up by the host CPU. The size of the grid can be adjusted to match the problem size and hardware capabilities, allowing developers to optimize performance for a specific problem. Each block, on the other hand, is a group of threads that are executed in parallel on the GPU. Blocks are typically smaller than grids and are used to break down the computation into smaller, more manageable tasks. The size of each block can also be adjusted to optimize performance for a specific problem. Finally, each thread is an individual path of execution that performs a small part of the overall computation. Threads within a block can communicate with each other and share data through fast shared memory, while threads in different blocks cannot communicate with each other.

The key to achieving high performance with CUDA architectures is to optimize the use of threads within blocks and blocks within grids to minimize data movement and maximize

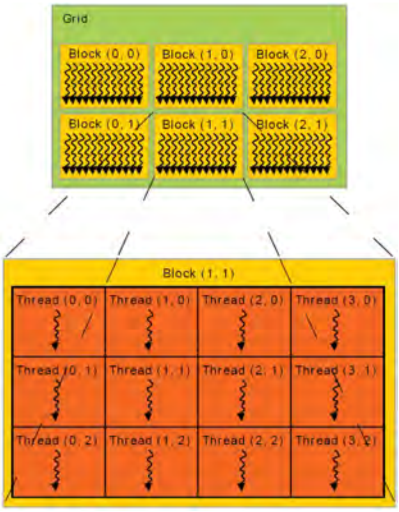


Figure 2.3: CUDA programming model [28]

parallelism. This requires careful consideration of the size of the grid and the block, as well as the way in which data is accessed and shared between threads.

In general, CUDA architectures are designed to handle large-scale parallel computations, such as those required in scientific simulations, machine learning, and image processing. So, by breaking down the computation into smaller units of work that can be executed in parallel, CUDA we can take advantage of the massive parallelism offered by GPUs and achieve significant speedups over traditional CPUs.

Chapter 3

Design and Implementation of the Parallel ICP Algorithm

This chapter begins with some previous considerations about how point cloud data is treated through implementations. After this, a serial ICP algorithm is placed. Finally, the parallel ICP algorithm designing is approached using a flow diagram and each of the processes presented in it are described in detail.

3.1 Previous Considerations

3.1.1 Input Data

Two types of data are used throughout all the tests in this thesis: synthetic and real data. On the one hand, synthetic data is created during the execution of programs and it lets the possibility of easily varying the number of points and inserting noise; in this case, synthetic data intends to model 3D surfaces using functions $z = f(x, y)$ where $f : \mathbb{R}^2 \to \mathbb{R}$. On the other hand, real data comes from the Ouster LiDAR OS1-16 and it is used to test the ICP implementations and to see if they work in a real context.

3.1.2 Data Grouping

Point clouds are stored in memory in one-dimensional arrays, so if a point cloud has n points, then the number of elements in the array is $3n$. To store this data, two types of grouping are employed in either serial or parallel implementations. Figure 3.1 shows how they are constructed. The first type of grouping data consists in holding point clouds in arrays in such a way that the

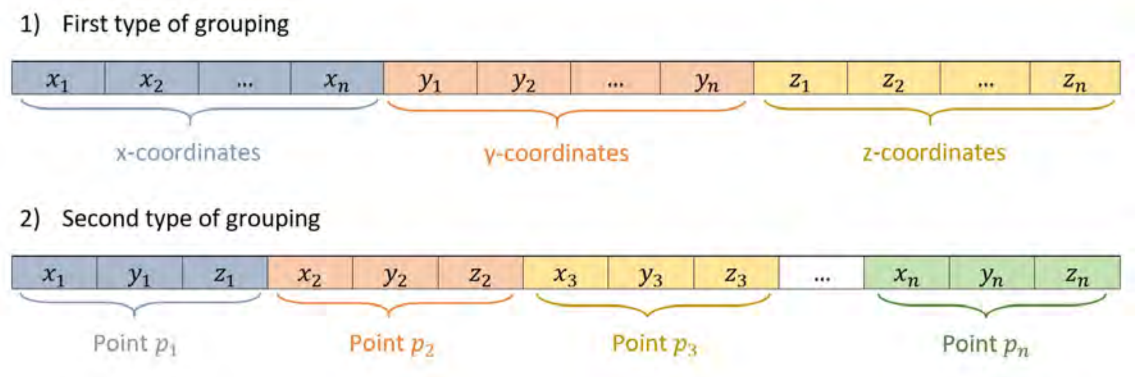


Figure 3.1: Two types of grouping point cloud data.

first n cells store all the x-coordinates and the next $2n$ cells in the y and z coordinates. So, to get the first point of each cloud, the program has to ask memory for the 1^{st}, n^{th} and $(2n)^{th}$ term of the array and a similar process for all points. The second type of grouping data is based on placing the three coordinates of each point one after another. Thus, to get the first point, the program has to ask for memory for the 1^{st}, 2^{nd} and 3^{rd} term of the vector.

Each of the types of grouping data owns some advantages and disadvantages. When trying to access the three coordinates of a certain point using the second type of data induces a single call to the memory due to the closeness of their memory addresses while using the first type provokes three different calls to the memory because of the separation of the memory addresses. However, the first type has also some advantages in comparison to the other, such as the easier way and faster computation for processing vector and matrix operations. In conclusion, due to the fact that some parts of the ICP algorithm work well with the first type of grouping data and some others work well with the second type, for each specific algorithm implementation a type of grouping is analyzed and used depending on what makes the program run faster.

3.2 CPU ICP Implementation

Two ICP algorithms are implemented on the CPU and can run either sequential or parallel (because of the MKL library that easily allows the swapping between these two options). One uses the point-to-point error metric and the other, the point-to-plane error metric, but the two of them utilize the brute force algorithm for the matching step. Also, both of them use synthetic data, because it easily allows finding the time complexity of these implementations. Furthermore, these algorithms were implemented in C based on MATLAB code [14] and optimized using the MKL library to get the highest performance in the CPU.

Figure 3.2 shows the processes involved in the main loop for both algorithms. The orange path traces the flow for sequential ICP with the point-to-point error metric and so the green one with the point-to-plane model.

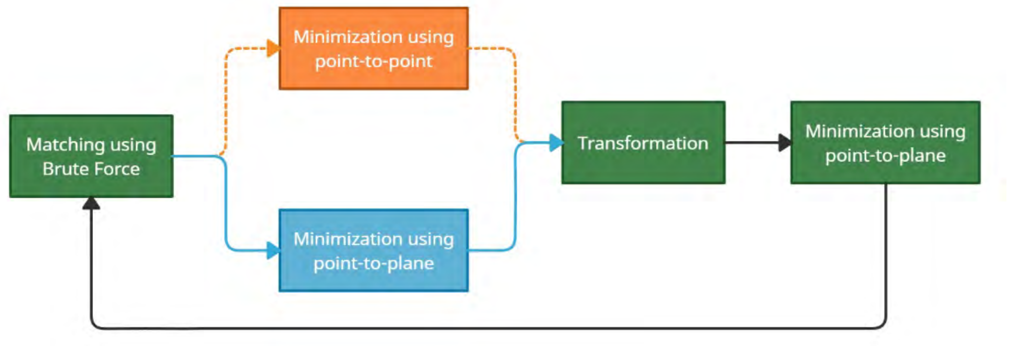

Minimization using point-to-point
Matching using Brute Force
Transformation
Minimization using point-to-plane
Minimization using point-to-plane

Figure 3.2: Sequential implementation flow diagram

3.2.1 MKL Library

Due to the requirement for matrix and vector operations and the need for methods and metrics such as SVD and Least-Squares, a high-performance library is vital for this work.

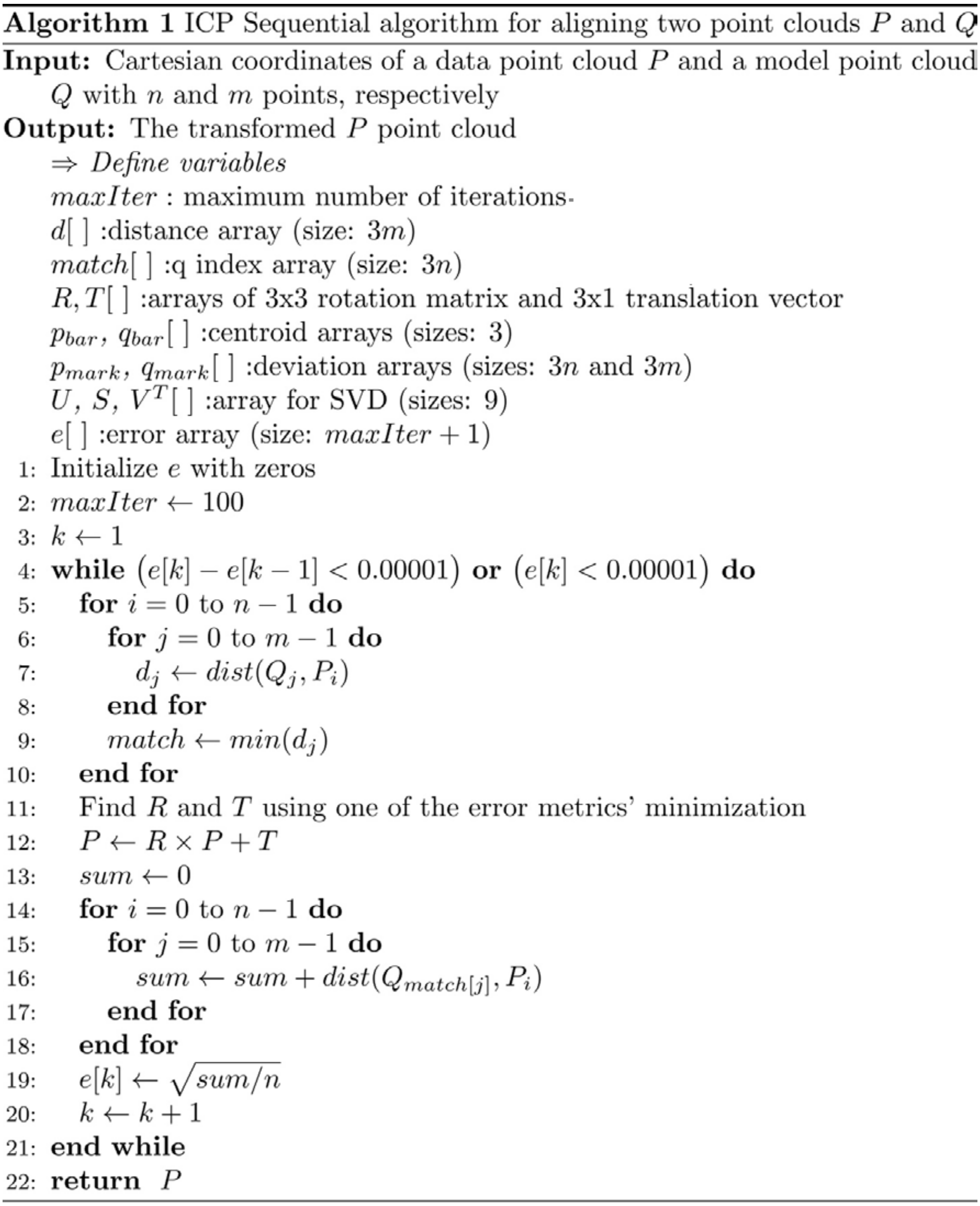
Describing itself as “the fastest and most-used math library for Intel®-based systems” on its main page, there is no doubt about selecting others other than the MKL library [10]. CPU implementations (sequential and parallel) using this library are guaranteed to obtain the greatest possible performance that a CPU (equipped with an Intel processor) can offer, in such a way that comparisons between CPU and GPU implementations are fair enough.

MKL has been designed to provide tools for the implementation of various applications. Linear algebra, fast Fourier transforms (FFT), vector statistics & data fitting, and vector math & miscellaneous solvers are the library packages that Intel provides. Established requirements lead to choosing the linear algebra package.

3.2.2 Sequential ICP Algorithm

The pseudocode for the sequential ICP algorithm is shown in Figure 3.3. Notice that every step described in Figure 3.2 has been traduced to pseudocode with the exception of the minimization step because steps to compute it were already presented in sections 2.5.3.1 and 2.5.3.2.

ET LUX IN TENEBRIS LUCET MCMXVII



```
**Algorithm 1** ICP Sequential algorithm for aligning two point clouds $P$ and $Q$
**Input:** Cartesian coordinates of a data point cloud $P$ and a model point cloud
    $Q$ with $n$ and $m$ points, respectively
**Output:** The transformed $P$ point cloud
    $\Rightarrow$ *Define variables*
    $maxIter$ : maximum number of iterations
    $d[\ ]$ :distance array (size: $3m$)
    $match[\ ]$ :q index array (size: $3n$)
    $R, T[\ ]$ :arrays of 3x3 rotation matrix and 3x1 translation vector
    $p_{bar}$, $q_{bar}[\ ]$ :centroid arrays (sizes: 3)
    $p_{mark}$, $q_{mark}[\ ]$ :deviation arrays (sizes: $3n$ and $3m$)
    $U$, $S$, $V^T[\ ]$ :array for SVD (sizes: 9)
    $e[\ ]$ :error array (size: $maxIter + 1$)
Initialize $e$ with zeros
$maxIter \leftarrow 100$
$k \leftarrow 1$
**while** $\big(e[k] - e[k-1] < 0.00001\big)$ **or** $\big(e[k] < 0.00001\big)$ **do**
    **for** $i = 0$ to $n - 1$ **do**
        **for** $j = 0$ to $m - 1$ **do**
            $d_j \leftarrow dist(Q_j, P_i)$
        **end for**
        $match \leftarrow min(d_j)$
    **end for**
    Find $R$ and $T$ using one of the error metrics' minimization
    $P \leftarrow R \times P + T$
    $sum \leftarrow 0$
    **for** $i = 0$ to $n - 1$ **do**
        **for** $j = 0$ to $m - 1$ **do**
            $sum \leftarrow sum + dist(Q_{match[j]}, P_i)$
        **end for**
    **end for**
    $e[k] \leftarrow \sqrt{sum/n}$
    $k \leftarrow k + 1$
**end while**
**return** $P$
```

Figure 3.3: Sequential ICP algorithm

3.3 GPU ICP: Design and Implementation

Two solutions are proposed for this work and, in this section, the flow diagram of the two of them are illustrated in Figure 3.4 and Figure 3.5, where blocks in blue represent serial processes and blocks in red represent parallelized processes.. They represent algorithms for single alignments of certain data point clouds (P) and model point clouds (Q). If having more point clouds to align, this whole process repeats. In the next subsections, the processes listed in the flow diagram are described.

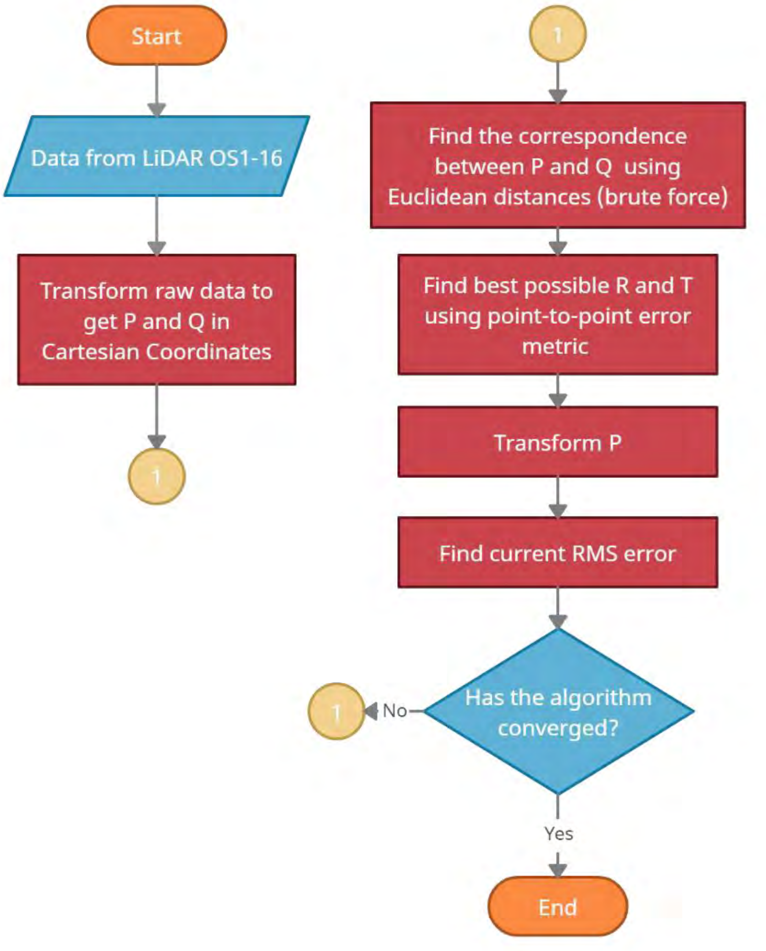

Start
Data from LiDAR OS1-16
Transform raw data to get P and Q in Cartesian Coordinates
1
1
Find the correspondence between P and Q using Euclidean distances (brute force)
Find best possible R and T using point-to-point error metric
Transform P
Find current RMS error
Has the algorithm converged?
No
1
Yes
End

Figure 3.4: Proposed solution model 1.

3.3.1 Receiving Data from the LiDAR OS1-16

Ouster LiDAR sensors [23] offer reliability in the acquisition of data due to their high accuracy and efficiency. One of the most popular models of these Ouster sensors is the OS1, which belongs to the mid-range category. This model presents a variety of number of channels (16, 32, 64, 128)

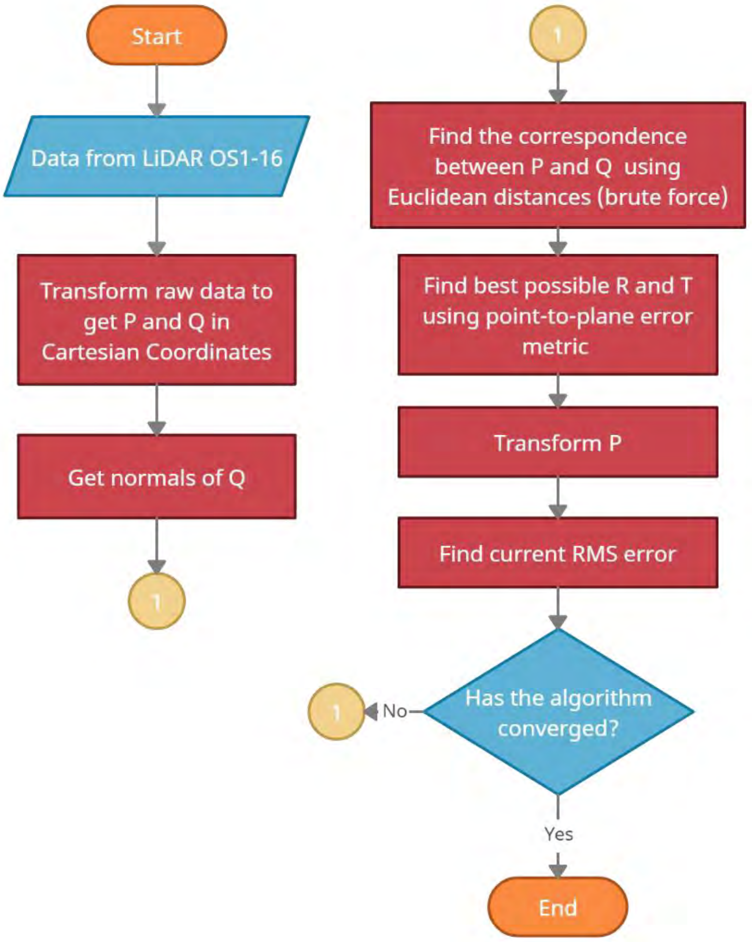

Start
Data from LiDAR OS1-16
Transform raw data to get P and Q in Cartesian Coordinates
Get normals of Q
1
1
Find the correspondence between P and Q using Euclidean distances (brute force)
Find best possible R and T using point-to-plane error metric
Transform P
Find current RMS error
Has the algorithm converged?
No
1
Yes
End

Figure 3.5: Proposed solution model 2.

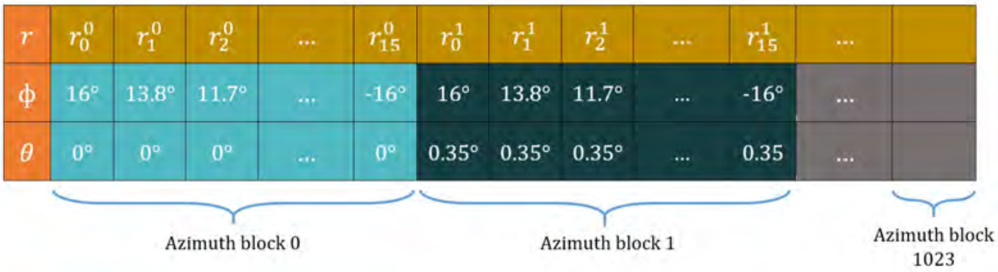
in five different OS1 sensor models. As a matter of fact, the greater number of channels that the OS1 sensor has, the greater number of points scanned. For the application of this work, the OS1 with 16 channels (OS1-16) has been chosen.

Ouster LiDAR sensors have their own graphic environment called Ouster Studio, developed by ParaView [25]. This program allows users to have two modes to see the data: online and offline. Online mode lets users see how data is being captured in real-time, whereas offline mode offers replays of data-capturing moments. Furthermore, this program provides different tools that can be used to see point clouds from various perspectives and colors.

The output data of the OS1 sensors present a variety of information about the scanned scenario, such as the distance, intensity, reflectivity, and noise of each scanned point. Also, due to the fact that this data is in polar coordinates format, information about the azimuth and altitude angles is given. The resolution of this information depends on the number of channels of the sensor, which sets the vertical resolution, and on the LiDAR mode, which establishes the horizontal resolution. In particular, data coming from a full rotation of the OS1-16 (1 donut) has a vertical resolution of 1 azimuth (16 points) and a horizontal resolution of 1024 azimuths. This means that the number of points of a donut is 16384 points, which are stored in an ASCII file of 806912 bytes separated into 64 data packets (1 data packet = 16 azimuth blocks).

The objective of this process (Data from LiDAR OS1-16) is to extract the necessary information from the ASCII file, i.e., ranges, the azimuth, and altitude angles, store it in the CPU memory, and then transfer it to the GPU memory. The simplest way to accomplish this is to assign each variable an array, i.e., an array for the ranges, another for the azimuth angles, and another for the altitude angles. However, this work selects a more efficient solution. Storing the ranges in order according to the number of channels, azimuth block, and data packet, there is no need to store the whole azimuth nor altitude angles arrays, which lets the implementation save time and memory.

Figure 3.6 where ranges are denoted by r, altitude angles by ϕ and azimuth angles by θ illustrates this last explanation; each cell is equal to 1 channel, 16 of these channels are equal to an azimuth block and 16 azimuth blocks are equal to 1 data packet. In this case, the array in mustard would be stored in memory for the ranges. The altitude and azimuth angles data come from a file called beam intrinsics and from an encoder counter that the sensor has. For more information about the LiDAR data format, refer to [24].



r	r_0^0	r_1^0	r_2^0	...	r_{15}^0	r_0^1	r_1^1	r_2^1	...	r_{15}^1	...	
ϕ	16°	13.8°	11.7°	...	-16°	16°	13.8°	11.7°	...	-16°	...	
θ	0°	0°	0°	...	0°	0.35°	0.35°	0.35°	...	0.35	...	

Figure 3.6: LiDAR data storage

The flow diagram of "Data from LiDAR OS1-16" is shown in Figure 3.7. Notice that all of these processes are executed in the CPU. Also, it is important to mention that this algorithm is executed twice: one time for the P point cloud and the other for the Q point cloud.

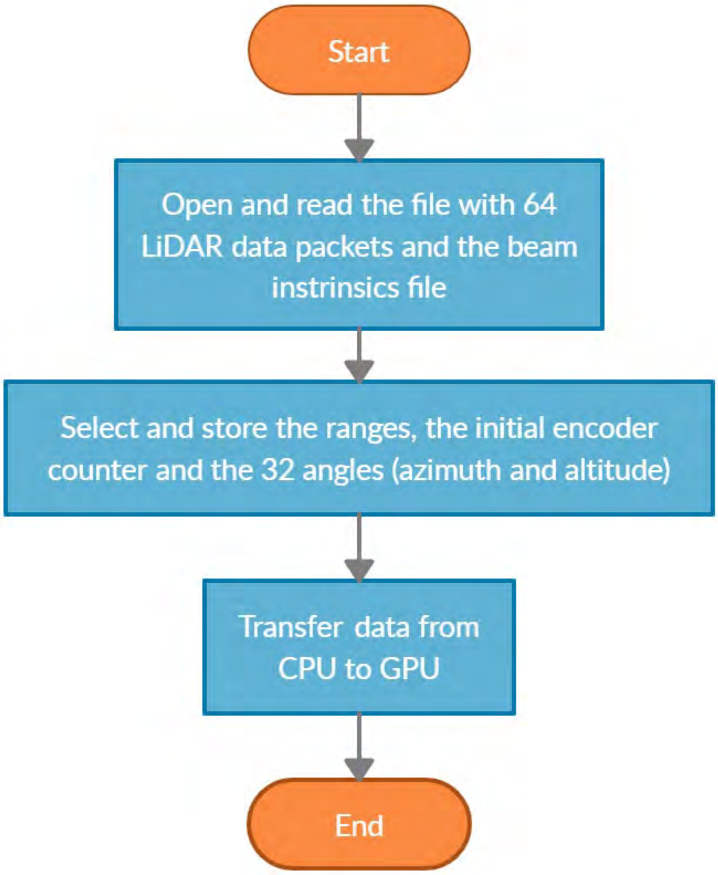
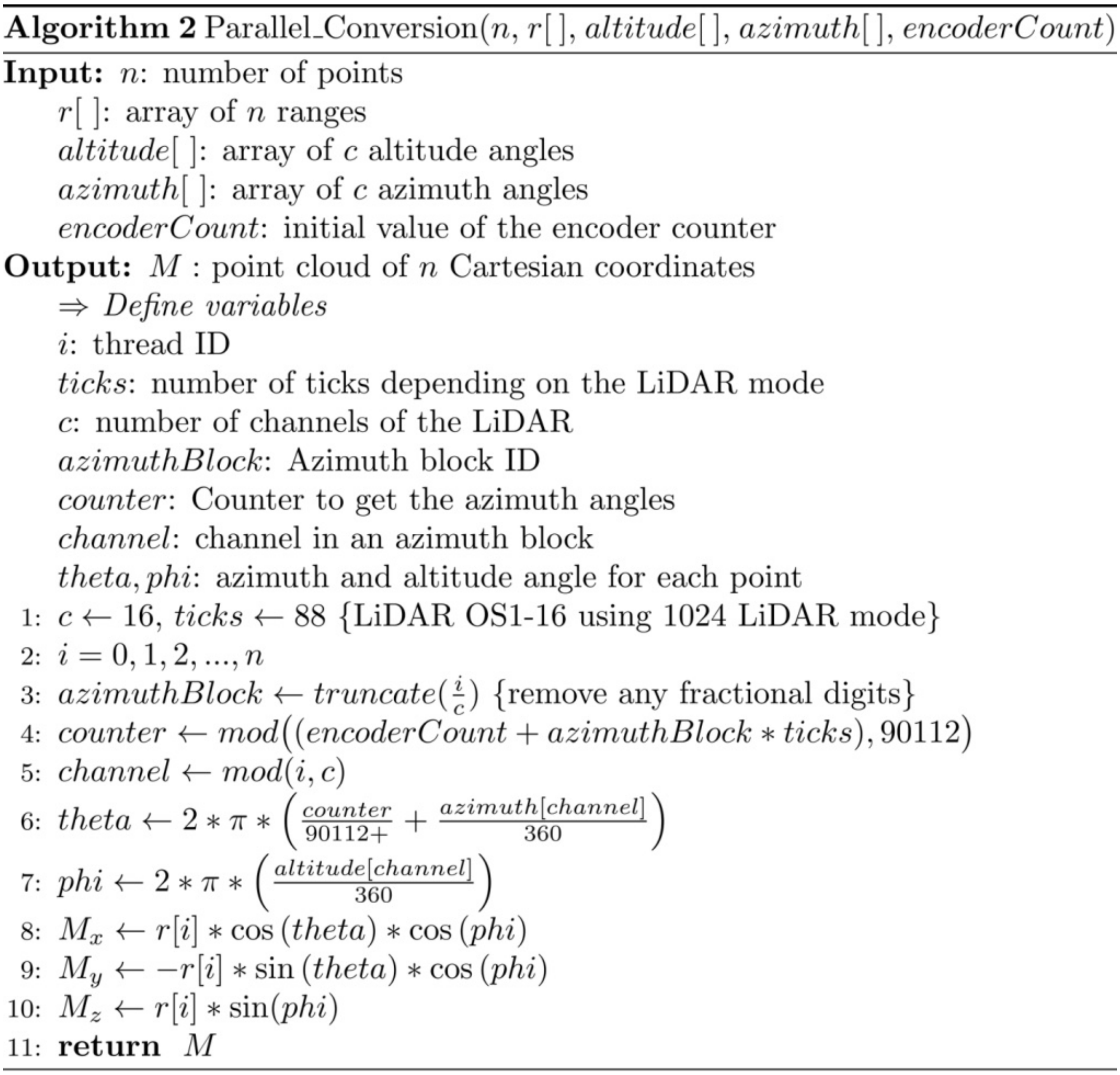

Start
Open and read the file with 64 LiDAR data packets and the beam instrinsics file
Select and store the ranges, the initial encoder counter and the 32 angles (azimuth and altitude)
Transfer data from CPU to GPU
End

Figure 3.7: Data from LiDAR OS1-16 flow diagram.

3.3.2 Transforming Raw Data into Cartesian Coordinates

Now that point clouds are stored in the GPU memory in polar coordinates format, a conversion to Cartesian coordinates is necessary. This is because most of the algorithms and methods the ICP works with are designed for Cartesian coordinates. The algorithm of the kernel that solves this task is presented in Figure 3.8. Each thread is in charge of one conversion and equations to convert from polar to Cartesian was taken from [24]. Also, notice that this algorithm computes the conversion for only one cloud; if having two clouds, then duplicate the lines from 9 to 16. Finally, after this algorithm implementation, if using the point-to-plane model, data must still be stored in the GPU memory in order to calculate the normals in the next procedure; otherwise, data must be moved to the CPU to start with the ICP main loop.



```
Algorithm 2 Parallel_Conversion($n, r[\,], altitude[\,], azimuth[\,], encoderCount$)
Input: $n$: number of points
    $r[\,]$: array of $n$ ranges
    $altitude[\,]$: array of $c$ altitude angles
    $azimuth[\,]$: array of $c$ azimuth angles
    $encoderCount$: initial value of the encoder counter
Output: $M$ : point cloud of $n$ Cartesian coordinates
    $\Rightarrow$ Define variables
    $i$: thread ID
    $ticks$: number of ticks depending on the LiDAR mode
    $c$: number of channels of the LiDAR
    $azimuthBlock$: Azimuth block ID
    $counter$: Counter to get the azimuth angles
    $channel$: channel in an azimuth block
    $theta, phi$: azimuth and altitude angle for each point
$c \leftarrow 16$, $ticks \leftarrow 88$ {LiDAR OS1-16 using 1024 LiDAR mode}
$i = 0, 1, 2, ..., n$
$azimuthBlock \leftarrow truncate(\frac{i}{c})$ {remove any fractional digits}
$counter \leftarrow mod\big((encoderCount + azimuthBlock * ticks), 90112\big)$
$channel \leftarrow mod(i, c)$
$theta \leftarrow 2 * \pi * \left(\frac{counter}{90112+} + \frac{azimuth[channel]}{360}\right)$
$phi \leftarrow 2 * \pi * \left(\frac{altitude[channel]}{360}\right)$
$M_x \leftarrow r[i] * \cos(theta) * \cos(phi)$
$M_y \leftarrow -r[i] * \sin(theta) * \cos(phi)$
$M_z \leftarrow r[i] * \sin(phi)$
return $M$
```

Figure 3.8: Parallel algorithm for coordinate conversion.

3.3.3 Getting Normals of Model Point Cloud Q using PCA

For this process, this work uses the K-Nearest Neighbor PCA (KNN-PCA) method [33] to find the normals of Q which are divided into three steps as shown in Figure 3.9. This flow diagram is designed for finding the normals of each point of Q, so for implementing this algorithm in parallel, each thread is in charge of finding one normal.

Each of the blocks in the flow diagram is explained in the next lines. First, to find the k neighbors of each point, the matching algorithm is described in the next subsection. For this implementation, the number of k nearest neighbors is 4, due to the research made in [33], which suggests following these parameters for reaching the best performance. Although, if changing the value of k improves the performance of the algorithm, this change will be applied. Second,

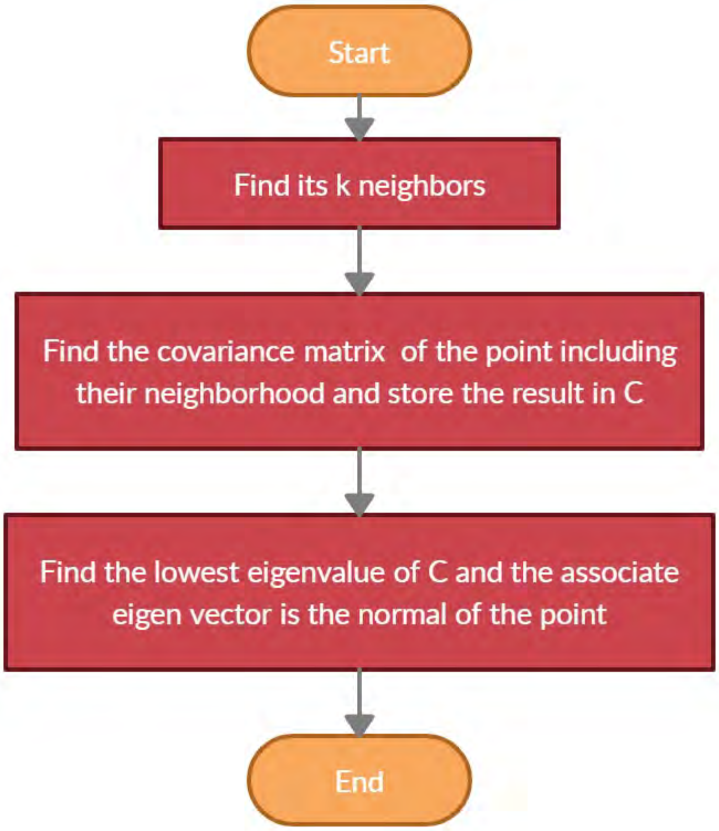

Start
Find its k neighbors
Find the covariance matrix of the point including their neighborhood and store the result in C
Find the lowest eigenvalue of C and the associate eigen vector is the normal of the point
End

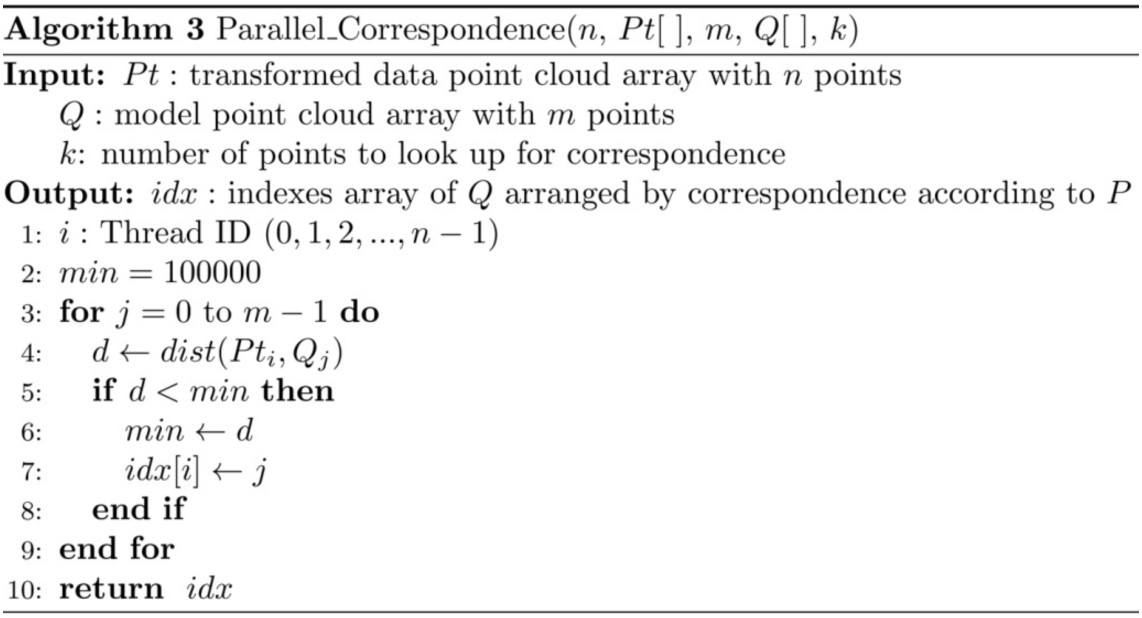
Figure 3.9: Normal estimation flow diagram.

covariance matrices are calculated using the theory described in Chapter 2. Third, since the obtained covariance matrix is a 3×3 symmetric matrix, a simple algorithm for finding the eigenvalues and eigenvectors is employed.

3.3.4 Finding the Correspondence using Brute Force

As has been mentioned in different parts of this document, the matching or correspondence of points process is the most time-consuming step of the ICP. This is why lots of algorithms have been developed and some of them were presented in Chapter 2. Nevertheless, this thesis is using the simplest of them: Brute Force. This algorithm has a very simple way to parallelize, and many techniques can be applied to it to improve performance.

Figure 3.10 shows up the matching algorithm to use in this work. To sum it up, it basically measures the distance between two points and compares it with the current minimum value. This process repeats iteratively as many times as the number of points of Q. For the implementation, each thread is in charge of finding the nearest neighbor of each point of P.



```
Algorithm 3 Parallel_Correspondence($n$, $Pt[\,]$, $m$, $Q[\,]$, $k$)
Input: $Pt$ : transformed data point cloud array with $n$ points
    $Q$ : model point cloud array with $m$ points
    $k$: number of points to look up for correspondence
Output: $idx$ : indexes array of $Q$ arranged by correspondence according to $P$
$i$ : Thread ID $(0, 1, 2, ..., n-1)$
$min = 100000$
for $j = 0$ to $m-1$ do
    $d \leftarrow dist(Pt_i, Q_j)$
    if $d < min$ then
        $min \leftarrow d$
        $idx[i] \leftarrow j$
    end if
end for
return $idx$
```

Figure 3.10: Parallel algorithm for correspondence.

3.3.5 Minimizing the Error Metrics

Equations for minimizing these metrics were already covered in 2.5.3, so, in this case, CUDA routines are employed here to compute matrices operations, decompose matrices using SVD, and solve the system of linear equations. More specifically, the CuBLAS and CuSolver libraries are employed to solve these tasks.

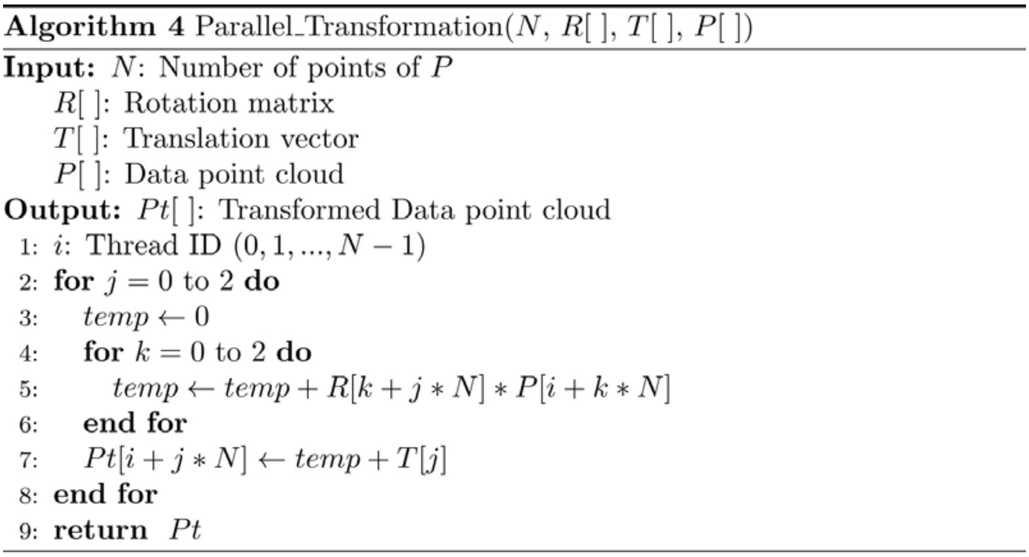
For the implementation of both error metric models, there is necessary to calculate the centroid of the cloud or similar operations where the x-coordinates are summed up and so the y- and z-coordinates. For the solution of this, a matrix-vector multiplication is used and the explanation is shown in the following expression:

$$\begin{pmatrix} x_0 & x_1 & x_2 & \dots & x_n \\ y_0 & y_1 & y_2 & \dots & y_n \\ z_0 & z_1 & z_2 & \dots & z_n \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} x_{\text{sum}} \\ y_{\text{sum}} \\ z_{\text{sum}} \end{pmatrix} \tag{3.1}$$

Where the matrix from the left $3 \times N$ holds the coordinates of the point cloud and the vector from the right $N \times 1$ is filled with ones. The result ends up in the sum of the x-coordinates in the resultant vector followed by the y and z coordinates.

3.3.6 Transforming the Source Point Cloud P

Figure 3.11 shows the parallel algorithm for transforming the P point cloud through iterations given the rotation matrix and translation vector found in the minimization step.



```
Algorithm 4 Parallel_Transformation($N$, $R[\,]$, $T[\,]$, $P[\,]$)
Input: $N$: Number of points of $P$
    $R[\,]$: Rotation matrix
    $T[\,]$: Translation vector
    $P[\,]$: Data point cloud
Output: $Pt[\,]$: Transformed Data point cloud
$i$: Thread ID $(0, 1, ..., N-1)$
for $j = 0$ to $2$ do
    $temp \leftarrow 0$
    for $k = 0$ to $2$ do
        $temp \leftarrow temp + R[k + j * N] * P[i + k * N]$
    end for
    $Pt[i + j * N] \leftarrow temp + T[j]$
end for
return $Pt$
```

Figure 3.11: Parallel algorithm for transformation.

3.3.7 Finding the Current RMS Error Value

Using Equation 2.15, the RMS error can be all implemented using the CuBLAS library. In fact, it is only needed a routine for subtract to vectors and then another one to compute the norm of the resultant vector.

3.3.8 Validation

To know how close P and Q should be, criterions about the error are the following ones. One the one hand, if RMS error stays in the same value from one iteration to another, it might be that the algorithm will not get lower RMS error values in later iterations. In other words, expression $e_k - e_{k-1} < \delta$ becomes the first criterion that will stop the ICP algorithm, where e_k represents the RMS actual error (k iteration) and e_{k-1} the RMS past error ($k-1$ iteration). However, sometimes ICP keeps the RMS through a few iterations and then it converges to lower RMS error values. This is why, another condition is taking in consideration and that is the verifying of the RMS error in comparison to a certain threshold value. So, on the other hand, $e_k < e_{th}$ will be the second criterion for stopping the ICP algorithm, having e_{th} in values around 10^{-6} units. To have a greater precision, RMS threshold error will be obtained in the experiments.

Chapter 4

Results

This chapter presents the results of the sequential and parallel implementations of the ICP algorithm. First, test conditions are presented to bind the implementations of this work. After this, computational results are presented and finally, the analysis of all these results is placed.

4.1 Test Conditions

4.1.1 Implementation Description

All tests were written in C for implementations in CPU and the extension of C in CUDA for implementations in GPU. MKL and CUDA APIs libraries were used for computing some basic linear algebra operations, solving systems of linear equations, and computing eigenvectors. Indeed, BLAS and LAPACK libraries were used in CPU and CuBLAS and CuSolver libraries were used in GPU.

For CPU implementations, it is important to mention that although the MKL library was used as much as it could, the implementation in CPU is somewhat parallel. There was a low percentage of code that had to be sequential because MKL needed to have a specific routine.

For GPU implementations, whenever the kernel to implement was already coded in some of the CUBLAS APIs, the correspondent routine was called. This is mainly because CuBLAS and CuSolver are highly optimized and a lot of effort would be required to surpass those routines with kernel implementations.

4.1.2 Devices

All experiments were evaluated using a CPU Intel® Core i5-9400U equipped with 4 physical cores at 2.9GHz with 32GB of RAM memory and a GPU NVIDIA GeForce RTX 2060 equipped with 1920 CUDA cores at 1755MHz, a total amount of memory of 6GB at 7001MHz with a bus width of 192 bits and based on the Turing architecture.

4.1.3 Input Data

Synthetic data, the Stanford Bunny dataset [33], and real data that comes from a LiDAR OS1- 16 are used in the thesis. Three different datasets were used to show the robustness of the ICP algorithm against different data situations.

Synthetic data is generated using the surface $f(x, y) = x^2 - y^2$, $x, y \in [-2, 2]$ for a variable number of points. Figure 4.1 where the data point cloud is coloured in red and the model point cloud in blue. Synthetic Data shows the form of the synthetic data using 16384 points for both point clouds. Rotations are in the range of 0 to 10 degrees and translations are in the range of 0 to 1.

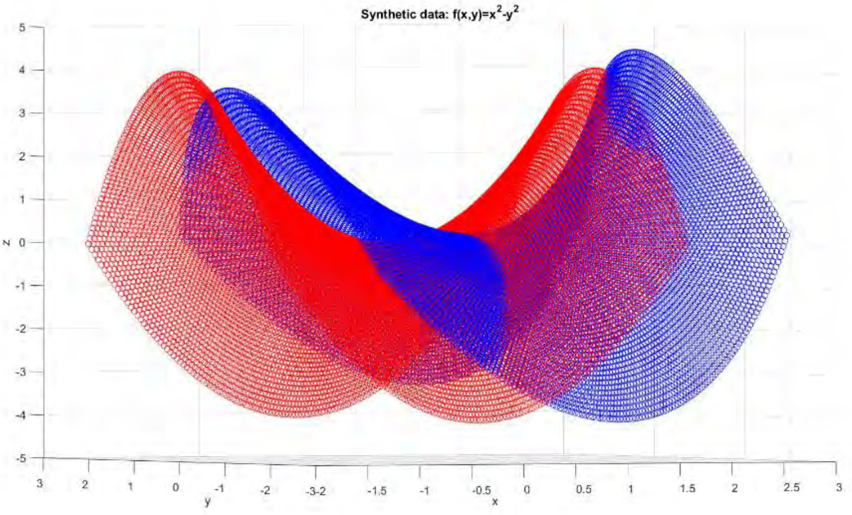


Figure 4.1: Synthetic data.

For the Bunny dataset, a group of 8171 points is selected for the tests. Rotations and translation values applied in synthetic data are used in this dataset. Figure 4.2 shows the two point clouds to align with the Stanford Bunny dataset.

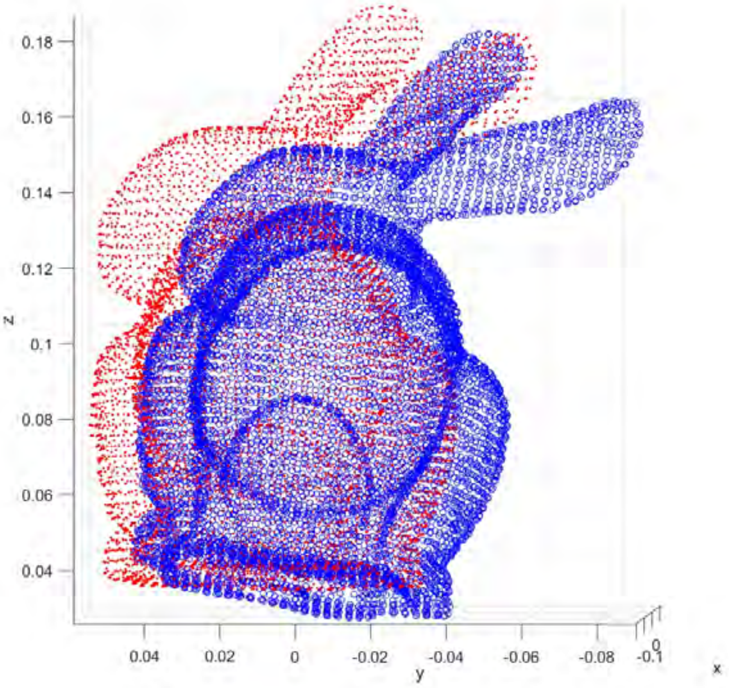


Figure 4.2: Stanford Bunny dataset.

Finally, real data, shown in Figure 4.3, is the result of scanning a hall located in the department O from PUCP university. In this case, 16384 points are received from the LiDAR. To match with the application, which only requires the ICP to compute fine alignments, rotations between scans are very small (less than 1 degree) and so translations (no more than 1 meter).

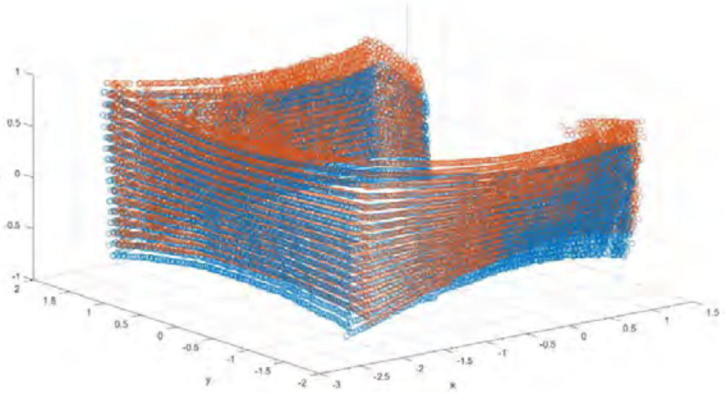


Figure 4.3: Real dataset from LiDAR OS1-16.

4.2 Computational Results

In this subsection, computational results, which are mainly focused on time metrics, are presented.

4.2.1 Time Complexity

For time complexity, synthetic data was utilized because it allows us to easily change in the number of points of the point clouds. In this case, the CPU implementation was changed to use the MKL library completely sequential and the GPU implementation ran using all necessary cores. The result of this test is presented in Figure 21. As can be noticed, CPU implementations have very similar results and so do GPU implementations. This is because the measured times were only recorded for the first iteration of the corresponding implementation, and this is why there are only two equations for tendency lines. The one that is right next to the CPU results is the one for both CPU implementations and the other one corresponds to the GPU implementations.

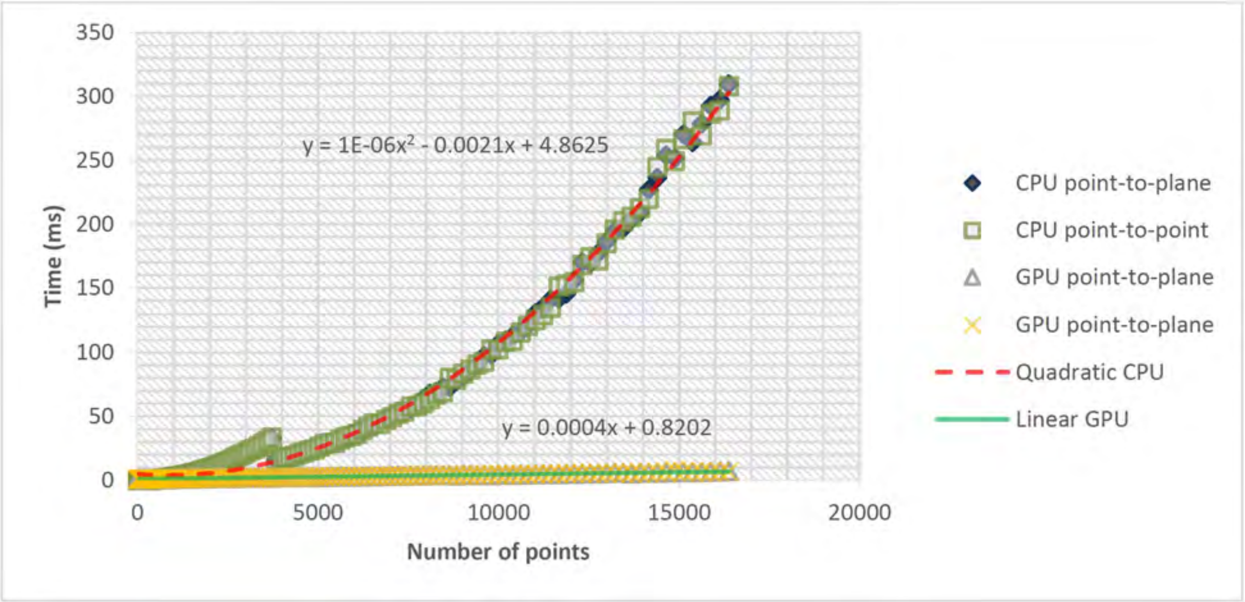


Figure 4.4: Time complexity.

4.2.2 Speedup

Using the time complexity results, the speedup graph presented in Figure 22 is obtained. Amdahl law (presented in equation 4.1) was employed for this purpose.

$$S_p(n) = \frac{10^{-6}n^2 - 0.0021n + 4.8625}{0.0004n + 0.8202} \tag{4.1}$$

It is important to recall that the Amdahl law requires the information for a sequential factor in the parallel implementation, which corresponds to the GPU implementation in this case. For the implementation of this work that factor takes the value of 0 because all the GPU code was parallelized whether using kernels or routines from the CUDA APIs.

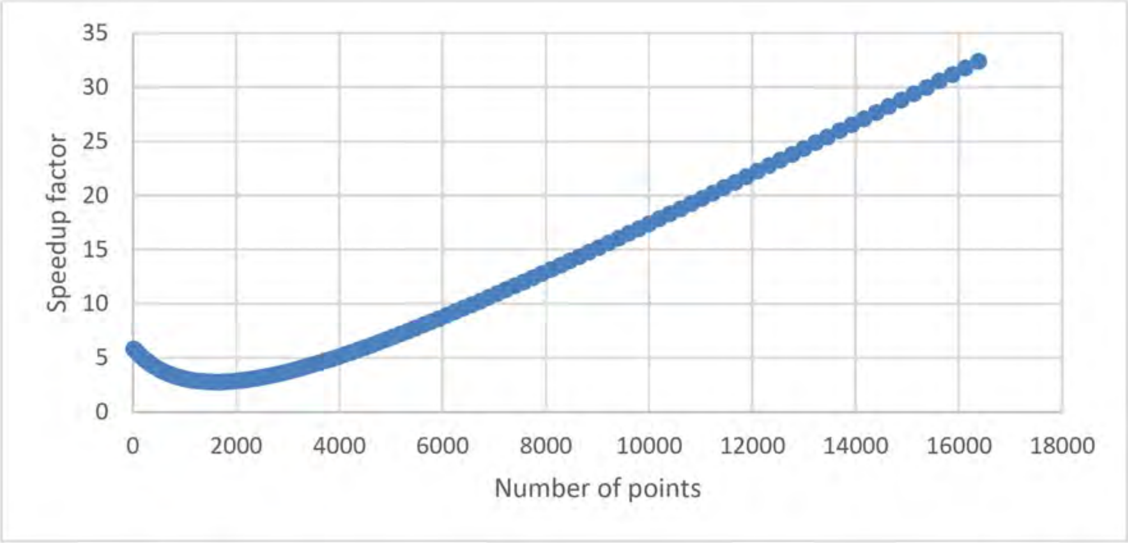


Figure 4.5: Speedup

4.2.3 Running Time

Running times of ICP implementations were recorded in two different ways for 16384 points using synthetic data. Table 4.1 presents the execution times for only one iteration and Table 4.2 the execution time for all the necessary iterations that the ICP took to converge.

Some important details about the following tables are mentioned in the next lines. First, normals were not taken into consideration in Table 4.1 because it belongs to preprocessing of the ICP and it is meant to be used in all necessary iterations. Second, speedup factors in Table 4.2 relate CPU with the GPU versions of the correspondent algorithm. Third, the CPU implementation presented here is the parallelized version, i.e., using the MKL library in parallel mode.

Table 4.1: Synthetic data running time for 1 iteration.

	Matching (ms)	Minimization (ms)	Transformation (ms)	Error (ms)	Total Time (ms)
CPU-point	326.7721	1.0227	0.0854	0.0447	328.0522
GPU-point	6.0571	1.2448	0.1030	0.2057	7.6122
CPU-plane	319.4783	1.4128	0.0955	0.0713	321.1978
GPU-plane	5.0177	0.8470	0.0682	0.1849	6.1182

Table 4.2: Synthetic data running time for all iterations.

	Normals (ms)	Matching (ms)	Minimization (ms)	Transformation (ms)	Error (ms)	Total Time (ms)	Iterations	Speedup factor
CPU-point	—	8579.5574	4.5248	0.7190	1.0432	8585.9870	27	—
GPU-point	—	147.8950	28.5900	2.3582	4.7214	183.5755	27	46.77
CPU-plane	547.5672	1571.1698	9.9679	0.2096	0.1991	2129.1136	4	–
GPU-plane	209.9402	24.1698	3.4347	0.2461	0.7697	238.5621	4	8.92

4.2.4 Error

RMS errors through the iterations of the two ICP algorithms using synthetic data with 16384 points were registered and the results are placed in Figure 4.6.

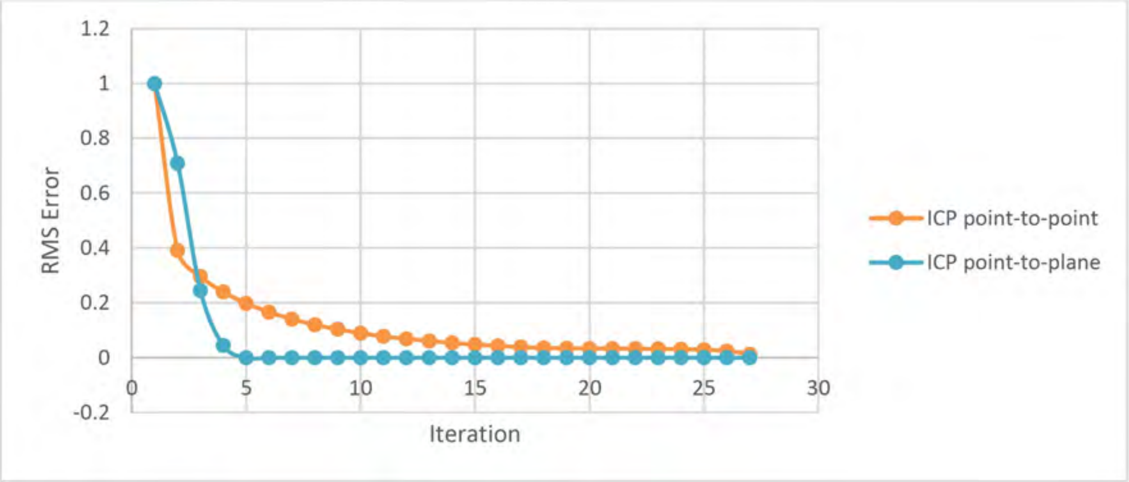


Figure 4.6: RMS error.

4.3 Optimizations

As it has been shown, the matching step consumes around 80% and 99% of each iteration in the ICP algorithm and because of this, a good optimization through the implementation of this Nearest Neighbor algorithm is highly required. The following list shows the optimizations that were made for this algorithm which are placed in Figure 3.10.

1. When calculating the distances between points from the cloud, the square root results not to be necessary for finding the closest points. So, the first optimization is to get rid of this operator.

2. The usage of registers allows access to data in a low number of clocks. Actually, in the range of tens of clocks. So, the second optimization is to use registers for storing the points from P and Q point clouds.

3. The loop-unroll technique divides kernel implementations, in such a way that threads do not have to go inside the loop cycle, but they execute the whole loop in a number of instructions. However, in this case, the for-loops are quite long, so the unrolling of the whole loop cannot be made. Instead of this, the loop can be divided into 2, 4, 8, and so on. Although, as a result of testing with different numbers of divisions, it comes to happen that after 2 divisions, there is no improvement. So, the third optimization is to use the loop-unroll technique dividing the for-loop into 2 halves.

4. The selection of the number of blocks and the number of threads inside each block plays a really important role. It is quite desirable to have as many numbers of blocks as the number of Streaming Multiprocessors (SMs) or a multiple of the available SMs. This is because, blocks are normally reserved in different SMs and if the last condition is not met, bottlenecks are produced inside blocks. In this case, the RTX 2060 has 30 SMs. Different numbers of blocks were tested, but the one that got the greatest performance was 60 blocks. Consequently, the number of threads in each one was the result of dividing the number of points over the number of blocks in such a way that there are as many launched threads as the number of points from clouds. So, for this last optimization, the selected number of blocks is 60 for the implementation of all the kernels.

Figure 4.7 shows the comparison for a different number of points between the non-optimized matching implementation with the optimized version using the mentioned techniques. Here, it can be noticed that the gradient of the non-optimized implementation gets reduced to, approximately, half of its value. Indeed, for 16384 points the execution time comes from 4.07 ms to 2.30 ms, which gives a speedup of 1.77.

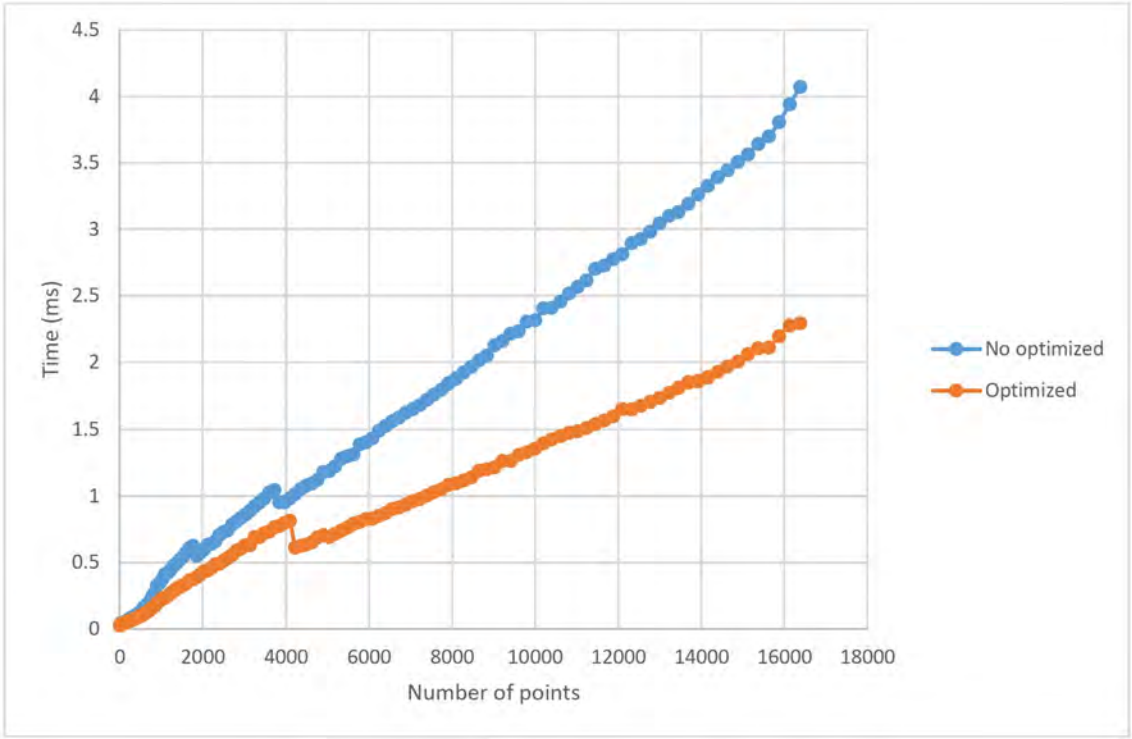


Figure 4.7: Matching optimization for 1 iteration.

4.4 Testing the Optimized Parallel ICP Algorithm

In this section, the three datasets presented at the beginning of this chapter are tested using the final parallelized versions of the ICP algorithm. The results of these experiments are summarized in Table 5. Again, ICP ran using the GPU and the two error metric models (point-to-point and point-to-plane) and the times were measured for the whole number of iterations that took the ICP to converge.

Table 4.3: Optimized ICP results.

Dataset	Error metric model	Number of points	Normals (ms)	ICP main loop (ms)	Total Time (ms)	Iterations
Synthetic	Point-to-point	16384	—	90.2169	90.2169	27
	Point-to-plane		100.9418	14.9802	115.9220	4
Bunny	Point-to-point	8171	—	30.5014	30.5014	17
	Point-to-plane		20.2734	11.5852	31.8586	5
LiDAR OS1-16	Point-to-point	16384	—	47.6056	47.6056	14
	Point-to-plane		106.1020	13.0001	119.1021	3

4.5 Analysis of Results

Figure 4.4 stresses the quadratic tendency of sequential implementations, due to the strong presence of the matching algorithm and the linear tendency of parallel implementations. This is produced by the fact that when using the necessary processors, the time complexity can be reduced in an order of magnitude. In this case, $O(n^2)$ is reduced to $O(n)$ because a number of n processors are available in the employed GPU.

Speedup function presented in Figure 4.5 shows the high computational capability of GPUs for processing big amounts of data. This is why, the function grows up faster as the number of points gets bigger.

Results from Figure 4.4 and Table 4.1 show that there is almost no difference between single iterations between the ICP point-to-point and the ICP point-to-plane. This means that the only difference between them is that the ICP point-to-plane converges in a lower number of iterations than the ICP point-to-point as it is shown in Figure 4.6.

Now, since each step from both algorithms takes more less the same amount of time, it is relevant to note that the Nearest Neighbor Searching algorithm, present in the matching and normals estimation steps, is consuming around 99.9% of the time for CPU point-to-point, 80.6% for the GPU point-to-point, 99.9% for the CPU point-to-plane and 98.1% of the time for GPU point-to- plane. This is why, an optimization was highly required for this algorithm. In here, Figure 4.7 exposed the optimized matching step is almost 2 times faster than the non-optimized version. Indeed, the results presented in Table 4.3 confirm this last statement if comparing the synthetic data results from this table with the GPU results showed in Table 4.2. In fact, not only the optimized parallel ICP implementations surpass the GPU implementations made at the beginning, but also it far exceeds the CPU parallel implementations presented in Table 4.2 with a factor of 95.

From Table 4.3, the speedup factor of the point-to-point implementation surpasses the point-to-plane one and the same situation repeats in Table 4.3. This is due to the fact that, in the point-to-plane, normal estimations require the Nearest Neighbor Searching algorithm, but multiplied by 4 times since it uses the KNN-PCA method. In consequence, the plan to minimize the execution time of the alignment by reducing the number of iterations in the ICP algorithm does not work in this case. Thus, the ICP point-to-point implemented in GPU has the fastest computation time.

Conclusions

- The successful design and implementation of a parallel ICP algorithm on a GPU demonstrates the potential for real-time alignment of misaligned point clouds. This advancement in algorithmic development opens up possibilities for efficient and accurate 3D point cloud registration in various applications.
- GPUs can reduce one order of magnitude from sequential time complexity, going from a time complexity of $O(n^2)$ in the serial implementation to a time complexity of $O(n)$ in the parallel implementation.
- Although time complexity was reduced and the implementation of the parallel ICP algorithm was optimized on GPU, the NNS implementation (Matching step) holds most of the running time on all implementations. The average percentage of the total running time taken by the NNS step in CPU and GPU implementations is 99.9% and 89.4%, respectively.
- The proposed ICP algorithm showcases remarkable resilience and efficiency in diverse scenarios, making it a reliable and efficient choice for point cloud registration tasks when implemented properly. It offers notable speed improvements, resulting in faster point cloud alignment compared to traditional sequential approaches (up to a 95x speed factor).
- The optimized point-to-point GPU implementation ($\approx$ 90 ms) performs 95 times faster than the CPU parallel version (≈ 8500 ms) and the point-to-plane GPU implementation (≈ 115 ms) runs 13 times faster than the respective CPU parallel version (≈ 2100 ms).
- The ICP point-to-point implementations on GPU demonstrate exceptional speed and efficiency, outperforming optimized CPU implementations. Notably, the real-time requirement is met, as evidenced by the alignment of real point cloud data from the OS1-16 LiDAR (16384 points), which was performed with an execution time of only 47ms. This result highlight the significant performance gains achieved by leveraging GPU technology for point cloud registration tasks.

Recommendations and Future Work

- Implementing a more efficient Nearest Neighbor Searching algorithm will enormously reduce computation times of ICP implementations presented in this work. KD Trees are presented as a good option for optimizing this process because, as it was mentioned in Chapter 2, its sequential time complexity of $O(n * \log n)$. So, if implementing this algorithm parallelized in GPU with at least n processors, the time complexity will get reduced to $O(\log n)$, which can reach better results than the parallelized Brute Force with its $O(n)$ time complexity.
- Since the running time of each kernel in GPU parallel implementations depends directly on the number of blocks and threads per block, it is suggested to implement a program capable of finding the best suited values of blocks and threads per block for the GPU to use.

Bibliography

[1] E. Q. ACOSTA, R. A. VELASCO, N. H. ROJAS, H. M. ESPINOZA, W. S. MARROQUIN, AND R. V. VIVAR, *Análisis estadístico de seguridad y compendio ilustrativo de accidentes en el sector de mediana minería y gran minería en 2018*, 2019.

[2] L. L. AMORIM, F. MUTZ, A. F. D. SOUZA, C. BADUE, AND T. OLIVEIRA-SANTOS, *Simple and effective load volume estimation in moving trucks using lidars*, Institute of Electrical and Electronics Engineers Inc., 10 2019, pp. 210–217.

[3] M. ATTIA AND Y. SLAMA, *Efficient initial guess determination based on 3d point cloud projection for icp algorithms*, Institute of Electrical and Electronics Engineers Inc., 9 2017, pp. 807–814.

[4] B. BELLEKENS, V. SPRUYT, R. BERKVENS, AND M. WEYN, *A survey of rigid 3d pointcloud registration algorithms*, 2014. Mathematic foundations are included in this paper.

[5] P. J. BESL AND N. D. MCKAY, *A method for registration of 3-d shapes*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 14 (1992), pp. 239–256.

[6] S. GARGOUM AND K. EL-BASYOUNY, *Automated extraction of road features using lidar data: A review of lidar applications in transportation*, Institute of Electrical and Electronics Engineers Inc., 9 2017, pp. 563–574.

[7] X. GU, X. WANG, AND Y. GUO, *A review of research on point cloud registration methods*, vol. 782, Institute of Physics Publishing, 4 2020, p. 022070.

[8] W. GUAN, W. LI, AND Y. REN, *Point cloud registration based on improved icp algorithm*, Institute of Electrical and Electronics Engineers Inc., 7 2018, pp. 1461–1465.

[9] H. HOPPE, T. DEROSE, T. DUCHAMP, J. MCDONALD, AND W. STUETZLE, *Surface reconstruction from unorganized points*, Association for Computing Machinery (ACM), 1992, pp. 71–78.

[10] INTEL®, *Math kernel library*.

[11] S. IZADI, D. KIM, O. HILLIGES, D. MOLYNEAUX, R. NEWCOMBE, P. KOHLI, J. SHOTTON, S. HODGES, D. FREEMAN, A. DAVISON, AND A. FITZGIBBON, *Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera*, ACM Press, 2011, pp. 559–568.

[12] D. KALMAN, *A singularly valuable decomposition: The svd of a matrix*, 1 1996.

[13] H. M. KJER AND J. WILM, *Evaluation of surface registration algorithms for pet motion correction*, 2010.

[14] ——, *Iterative closest point*, 2013.

[15] P. LI, R. WANG, Y. WANG, AND W. TAO, *Evaluation of the icp algorithm in 3d point cloud registration*, IEEE Access, 8 (2020), pp. 68030–68048.

[16] S. LI, J. WANG, Z. LIANG, AND L. SU, *Tree point clouds registration using an improved icp algorithm based on kd-tree*, vol. 2016-Novem, Institute of Electrical and Electronics Engineers Inc., 11 2016, pp. 4545–4548.

[17] K.-L. LOW, *Linear least-squares optimization for point-to-plane icp surface registration*, 2 2004.

[18] G. MANDLBURGER, K. WENZEL, A. SPITZER, N. HAALA, P. GLIRA, AND N. PFEIFER, *Improved topographic models via concurrent airborne lidar and dense image matching*, d-nb.info, (2017).

[19] S. MAY, A. NUCHTER, D. QIU, AND A. NÜCHTER, *Gpu-accelerated nearest neighbor search for 3d registration*, (2009).

[20] A. W. MOORE AND A. W. MOORE, *Efficient memory-based learning for robot control*, (1990), pp. 64–78.

[21] NVIDIA, *¿gpu vs. cpu? ¿qué es la computación por gpu?*

[22] A. NÜCHTER, K. LINGEMANN, AND J. HERTZBERG, *Cached k-d tree search for icp algorithms*, 2007, pp. 419–426.

[23] OUSTER, *High-resolution os1 lidar sensor: robotics, trucking, mapping*.

[24] ——, *Software user guide of the os1-16/64 high resolution lidar v1.13.0*, 2019.

[25] PARAVIEW, *Ousterstudio*.

[26] C. PEÑARANDA, *Minería y su aporte económico al perú*, 2019.

[27] F. POMERLEAU, S. MAGNENAT, F. COLAS, M. LIU, AND R. SIEGWART, *Tracking a depth camera: Parameter exploration for fast icp*, (2011).

[28] PÉTER, F. H. N. FITZEK, AND VINGELMANN, *Cuda, release: 10.2.89*, 2020.

[29] M. A. Q. ROSALES, *Registro de una secuencia temporal de nubes de puntos utilizando tecnología kinect para la reconstrucción tridimensional de material arqueológico*, 2014.

[30] S. RUSINKIEWICZ AND M. LEVOY, *Efficient variants of the icp algorithm*, Proceedings of International Conference on 3-D Digital Imaging and Modeling, 3DIM, (2001), pp. 145–152.

[31] A. V. SEGAL, D. HAEHNEL, AND S. THRUN, *Generalized-icp*, 2009.

[32] D. SUN AND J. HAN, *Newton and quasi-newton methods for a class of nonsmooth equations and related problems*, SIAM Journal on Optimization, 7 (1997), pp. 463–480.

[33] S. UNIVERSITY, *The stanford 3d scanning repository*, 2013.

[34] G. G. VOSSELMAN AND H.-G. MAAS, *Airborne and terrestrial laser scanning*, Whittles Publishing, 2010.

[35] F. WANG AND Z. ZHAO, *A survey of iterative closest point algorithm*, vol. 2017-Janua, Institute of Electrical and Electronics Engineers Inc., 12 2017, pp. 4395–4399.

[36] T. WEISE, *Global Optimization Algorithms -Theory and Application*, vol. 1, 2 ed., 2009.

[37] C. YANG AND G. MEDIONI, *Object modelling by registration of multiple range images*, Image and Vision Computing, 10 (1992), pp. 145–155.

[38] ÅKE BJÖRCK, *Linear least squares problems*, 2015.