MASTER'S THESIS 2026

GPU acceleration of LiDAR Inertial Odometry

Alve Lindell

ISSN 1650-2884
LU-CS-EX: 2026-41

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-41

GPU acceleration of LiDAR Inertial Odometry

GPU-acceleration av LiDAR-tröghetsodometri

Alve Lindell

GPU acceleration of LiDAR Inertial Odometry

Alve Lindell

June 22, 2026

Master's thesis work carried out at NFA.

Supervisor: Michael Doggett(LTH), Martin Ahrnbom(NFA)

Examiner: Jonas Skeppstedt

Abstract

LiDAR Inertial Odometry (LIO) provides highly accurate state estimation for autonomous robotic systems but suffers from significant computational bottlenecks, especially during the Generalized Iterated Closest Point (GICP) point cloud registration. On resource-constrained edge devices, this high CPU demand can prevent real-time execution and constrain the available computational resources that could be required for other tasks.

This thesis presents a GPU-accelerated implementation of Direct LiDAR-Inertial Odometry (DLIO). Utilizing the Kokkos performance portability framework and the ArborX bounding volume hierarchy (BVH) library, the entire point cloud pipeline was moved onto GPU, which includes voxelization, deskewing, GICP and map handling.

The implementation was evaluated on high-performance hardware and an embedded NVIDIA Jetson edge platform using urban data (Oxford Spires dataset) and forest data. The results demonstrate a performance improvement in most cases. On the Jetson platform, per-frame processing time was reduced from 116.26 ms to 70.76 ms, satisfying the 100 ms real-time constraint of standard 10 Hz LiDAR sensors. Furthermore, CPU usage on the edge device was decreased from roughly 300% CPU usage to 50%, while Absolute Trajectory Error (ATE) was marginally improved. This shows that GPU accelerated LIO can alleviate the CPU requirement of resource intensive robotics pipelines and improve performance.

Keywords: LiDAR, GPU, Odometry, GICP, Point Cloud Registration

Acknowledgements

I would like to thank all the people who helped me at NFA and especially David Gillsjö, Martin Ahrnbom and Rasmus Larsen for the help and discussion which helped me in my work. I would also like to thank my supervisor Michael Doggett for help and feedback given throughout the thesis.

Contents

List of Acronyms

GPU Graphical Processing Unit

ICP Iterated Closest Point

GICP Generalized Iterated Closest Point

LIO LiDAR Inertial Odometry

DLIO Direct LiDAR Inertial Odometry

kNN K Nearest Neighbors

HPC High Performance Compute

BVH Bounding Volume Hierarchy

Chapter 1
Introduction

1.1 Background and Motivation

In the field of robotics, odometry is very important, since in order to accurately control a robot, you usually need a very accurate idea of where the robot is. For this reason, many ways have been developed to estimate movement based on a wide variety of sensors.

1.2 Problem Statement

While LiDAR Inertial Odometry provides very accurate odometry based on LiDAR data, it suffers from one major constraint, its performance requirement. Performing GICP between point clouds is very expensive but highly parallel, so at the cost of several CPU cores it can run in real-time. The parallel nature of this algorithm and the significant cost make this highly suitable as a candidate for running on GPU since it could both increase the performance of and free up computational resources in CPU bound robotics systems.

1.3 Previous Work

LiDAR based odometry has been extensively studied and many different forms have emerged including multi-sensor algorithms such as LIO and those based solely on LiDAR such as GICP.

1.3.1 Direct LiDAR-Inertial Odometry (DLIO)

DLIO is an implementation of LIO that relies on GICP with map2scan matching as well as a complex hierarchical observer model for state estimation. It has superior performance

to many ICP based LIO algorithms and is designed to maintain accuracy during aggressive motion. It also uses this to provide accurate per point deskewing for motion correction during aggressive motions. However, this is still very performance heavy and without multi-threading the algorithm will not run in real time on larger point clouds and even with multi-threading it is still very CPU intensive [2].

1.3.2 Small GICP

GICP is an improvement over ICP that utilizes the fact that all points are in one way or another sampled from something that is locally a plane. This allows representing the points using their normal and this allows for better matching results in the sense that performance is improved since the convergence is better. Algorithms such as Voxelized GICP utilized by fast GICP can also improve the performance and also allows for running it efficiently on GPU. However, for running GICP efficiently on GPU, implementations using BVH has the potential to still further improve performance [4].

Chapter 2
Method

2.1 Sensors

LiDAR inertial odometry rely on several types of sensors in order to perform its function. It utilizes LiDARs which represents the LiDAR part of LIO. Additionally it also has an inertial part done by an inertial measurement unit.

2.1.1 LiDAR

A light detection and ranging sensor utilizes lasers in order to be able to scan an environment and returns a sparse point representation of the 3D environment consisting of many points which we call a point cloud. A LiDAR utilizes the fact that the speed of light is a known constant and therefore based on the round trip time between the emission of the laser and the time at which we detect a return signal from the laser reflecting on the environment we can acquire the distance to this point of reflection. This is done using the formula

$$\frac{c_{air} \cdot t_{round-trip}}{2}$$

where c_{air} is the speed of light in air which is not the same as in vacuum due to the refractive index of air. In rotational LiDARs the laser is rotated at high speed such that it takes one vertical scan utilizing it's n parallel lasers then goes one step horizontally and repeats until it's completed a full rotation. Then these individual scans are combined into one point cloud. The fact that all rays are not sent in the same instance gives rise to the problem of motion distortion. This happens if the LiDAR is moving while we scan which causes each vertical scan to be taken from a slightly different position, this can be corrected for which is discussed later. [11]

2.1.2 IMU

An inertial measurement unit is a sensor capable of measuring linear acceleration as well as orientation and angular acceleration. In some cases magnetic fields are also measured however, this is not of consequence for this use case so it will not be mentioned further. An IMU consists of two sensors an accelerometer for measuring linear acceleration data and a gyroscope for orientation measurement. An accelerometer is typically built using a tiny mass attached with springs. When the accelerometer accelerates then the mass will lag due to its inertia and will cause a displacement measured by the springs, it is worth noting that this will also measure acceleration caused by Earth's gravity since the tiny mass will constantly be pushed down by the gravitational force acting upon it causing a displacement. A gyroscope is built by a small disk that is spinning at a high speed attached to bearings allowing it to move freely in 3D. When this gyroscope is rotated, this spinning disk will resist movement that is against this axis since it will attempt to maintain its angular momentum. This can then be used to calculate how it has moved relative to this disk, however due to friction in the bearings this will drift over time and another note is that you need three of these in order to be able to measure an arbitrary rotation in 3D.

2.1.3 Time Synchronization

In order to be able to perform most LIO algorithms time syncing is usually a requirement for the sensor fusion. Time syncing is done by syncing up the sensors to some external exact time or by one sensor being synchronized against the others. This allows for the sensors to be able to send out data at exact known times relative to each other which gives higher accuracy when running sensor fusion algorithms. This is due to the fact that you know which measurements correspond to each other or the exact time difference between them. Many LiDAR sensors that have a built-in IMU are usually synced to the LiDAR sensor by default.

2.2 LIO

LiDAR inertial odometry is an algorithm for estimating odometry based on IMU and LiDAR data. This is done in several steps and can be done using many different algorithms for state estimation and LiDAR odometry. In this case DLIO is of particular interest since it's the underlying implementation that is GPU accelerated in this paper.

2.2.1 Preprocessing

Before LiDAR data and IMU data can be used preprocessing is done in order to correct for motion distortion in the LiDAR case and IMU bias for the IMU data. Motion distortion is a phenomenon that arises from the fact that the LiDAR sensor moves while a scan is being taken causing the point cloud to look "smeared" compared to a LiDAR scan from a still standing LiDAR. Based on the IMU data and the current state estimation one can estimate the position along the individual times each part of a LiDAR scan was captured which can then be corrected in a process called deskewing. The IMU data is also preprocessed by correcting for IMU bias which is a constant offset in acceleration and the angular speed data

common in IMUs. This bias can be found by measuring the offset of the angular speed and the acceleration while the IMU is at a standstill and then applying this offset to later incoming IMU data. [2]

2.2.2 Mapping

In order to be able to use point cloud data for localization an algorithm called GICP is used in order to find the best transform in a least squares way. In order to improve this, a map of point clouds are created for GICP to compare against. This map is created by capturing keyframes where a keyframe represents a new point cloud that is deemed to contain data not contained in other point clouds, this is done by capturing one new keyframe each time the sensor has moved or rotated a certain distance or angle [2]. Since the map grows in size with travel distance the GICP algorithm is not given the entire map due to the associated cost, instead a sub-map is created based on the closest n keyframes as well as other keyframes selected through heuristics that estimate if they contain relevant information for the current scan. [14] LiDAR Inertial Odometry can also be done without a map construction where you instead only compare the current scan with the previous scan in order to extract the position data from the LiDAR point clouds, in addition LIO can also be performed with alternative point cloud registration algorithms other than GICP such as ICP and NDT however, GICP with map to scan matching remains the most common.

2.2.3 State Estimation

With odometry data from both the IMU and the point cloud registration the two sources will need to be combined in order to get one more accurate odometry estimation. This is done through state estimation algorithms and depending on the use-case and the dynamics of the robot being used different models and filters can be used. The specifics of filter used is not relevant for GPU acceleration since it is neither parallelisable nor particularly costly in terms of performance [14].

2.3 Point Cloud Registration

Point cloud registration is a family of algorithms used to align one set of points with another by finding the transform that most closely align the source with the target see example 2.2. This can be used in Odometry applications since it can be used to estimate how a sensor generating point clouds has moved relative to it's last scan. This transform can be a general transform but in our cases since it represents movement we are only concerned with algorithms that estimate rigid transforms.

2.3.1 Non-linear least squares

Point cloud registration can be presented as an optimization problem were you are trying to minimize the sum of the squares of the errors. For the linear case this can be easily solved in a singular step however, in the case of non-linear systems it becomes hard and iterative

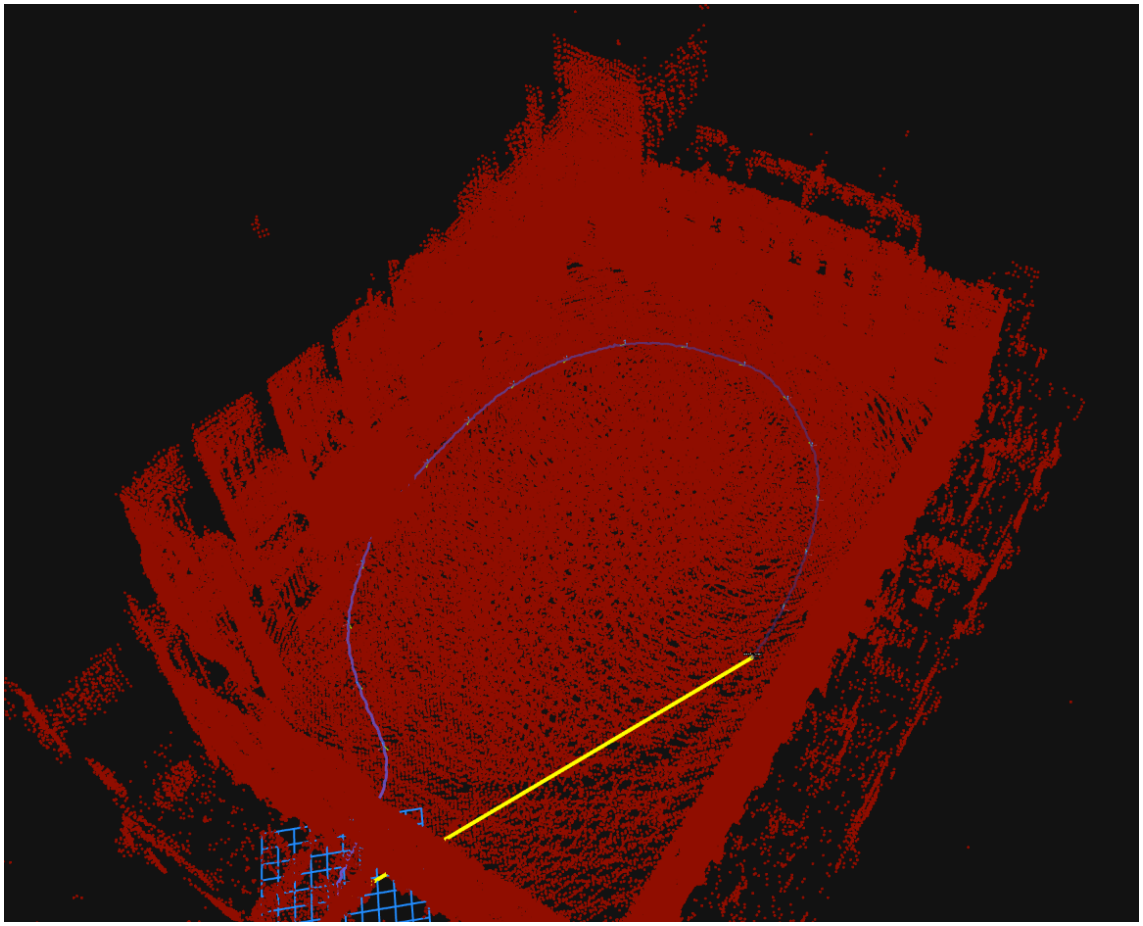


Figure 2.1: Example of the map and the path constructed by DLIO

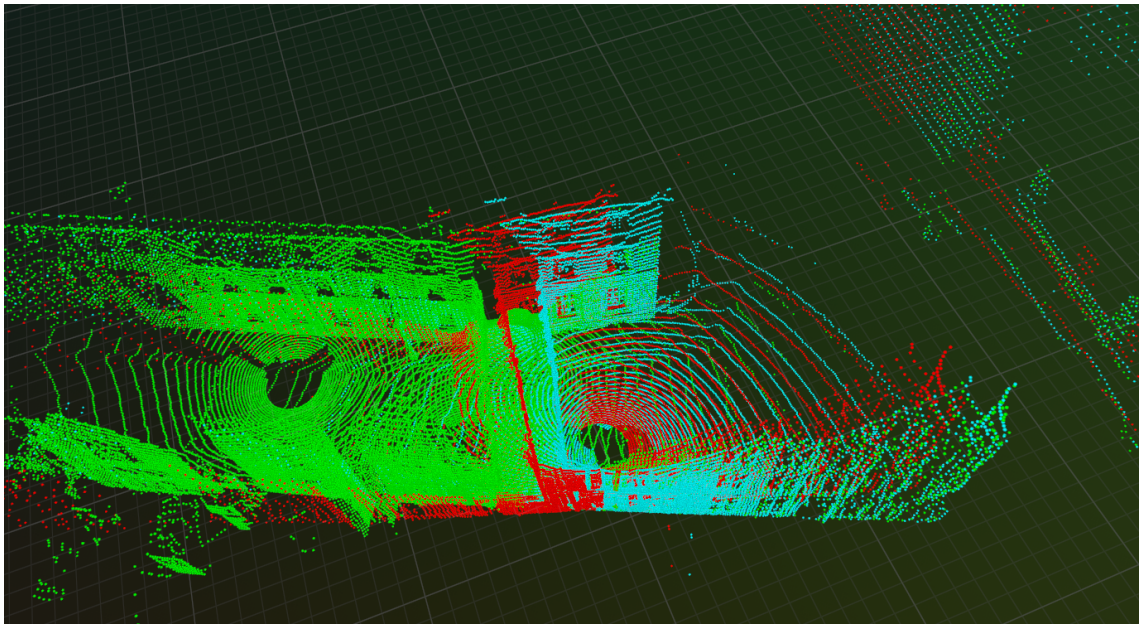


Figure 2.2: Example of a point cloud registration result aligning the red incoming scan with the green map resulting in the turquoise aligned scan

methods are required to approach some local minima. This can be done using several different algorithms although typically rely on linearizing the system and then taking a step towards the minima of the linearlized system and repeating.

Gauss-Newton

The Gauss-Newton algorithm is one of the iterative methods used to solve non-linear least squares. It is a modification of Newton's method for finding the minimum of a function that does not require calculating the Hessian matrix.

Given a vector of residuals $\mathbf{r}(\mathbf{x})$, we aim to minimize the cost function $S(\mathbf{x}) = \frac{1}{2}\|\mathbf{r}(\mathbf{x})\|^2$. By linearizing the residual using a first-order Taylor expansion around the current value of x

$$\mathbf{r}(\mathbf{x} + \delta\mathbf{x}) \approx \mathbf{r}(\mathbf{x}) + \mathbf{J}\delta\mathbf{x}$$

where $\mathbf{J}$ is the Jacobian matrix of the residuals. Substituting this into the cost function and setting the gradient with respect to δx to zero yields

$$(\mathbf{J}^\top\mathbf{J})\delta\mathbf{x} = -\mathbf{J}^\top\mathbf{r}(\mathbf{x})$$

In this formulation, $\mathbf{J}^\top\mathbf{J}$ acts as an approximation of the Hessian matrix $\mathbf{H}$. The update step is then defined as $x \leftarrow x + \delta x$. This method converges quadratically when the initial guess is close to the solution and the residuals are small. [7]

Levenberg-Marquardt

The Levenberg-Marquardt (LM) algorithm is an improvement upon the standard Gauss-Newton method by adding a dampening factor. It does this by interpolating between the Gauss-Newton algorithm and the method of gradient descent. LM is more robust than Gauss-Newton, as it can find a solution even if it starts very far off the final minimum and can provide faster convergence.

The LM algorithm introduces a damping parameter λ to the normal equations:

$$(\mathbf{J}^\top\mathbf{J} + \lambda\mathbf{I})\,\delta\mathbf{x} = -\mathbf{J}^\top\mathbf{r}(\mathbf{x})$$

where $\mathbf{I}$ is the identity matrix. The damping parameter λ is adjusted at each iteration depending on if the iteration reduced the error and by how much. [7] An improvement upon this can be done by using the diagonal elements of the approximate hessian instead of identity when doing the gradient decent in order to take into account the scaling of the actual diagonal elements and provide faster convergence. This yields the equation

$$(\mathbf{J}^\top\mathbf{J} + \lambda \cdot diag(\mathbf{J}^\top\mathbf{J}))\,\delta\mathbf{x} = -\mathbf{J}^\top\mathbf{r}(\mathbf{x})$$

[3]

2.3.2 ICP

Iterated Closest Point (ICP) is an algorithm used for aligning and registering point clouds by iteratively minimizing the distance between corresponding points. It does this in order

to find the transformation that solves the minimization problem

$$\min_{\mathbf{R}\in SO(3),\,\mathbf{t}\in\mathbb{R}^3} \sum_{i=1}^{N} \left\| \mathbf{q}_{c(i)} - (\mathbf{R}\mathbf{p}_i + \mathbf{t}) \right\|_2^2$$

where $S = \{\mathbf{p}_i\}$ is the source set and $T = \{\mathbf{q}_j\}$ is the target set and $c(i)$ is the correspondence function representing the index of the closest point in the target for a given point in the source.

$$c(i) = \arg \min_{j\in\{1,\ldots,M\}} \|\mathbf{p}_i - \mathbf{q}_j\|_2$$

Due to the fact that not all correspondences are of interest since some are too far away to represent real association we introduce the parameter $d_{\max}$ and then C is the set of valid correspondences

$$C = \{i \in \{1, \ldots, N\} \mid d_i \leq d_{\max}\}$$

where $d_i = \|\mathbf{p}_i - \mathbf{q}_{c(i)}\|_2$.
The algorithm proceeds by iterating over the following steps to compute an optimal transformation $\mathbf{R} \in SO(3)$ and $\mathbf{t} \in \mathbb{R}^3$ where initially the transformation is a best guess of the current transformation or identity if none is available.
1. Compute the set of valid correspondences C based on the current set of transformed source points

$$\mathbf{p}'_i = \mathbf{R}\mathbf{p}_i + \mathbf{t}$$

2. Linearize the residual between the transformed source points and their respective valid correspondences

$$\mathbf{r}_i = \mathbf{q}_{c(i)} - \mathbf{p}'_i$$

for a small increment $\delta\mathbf{x} = [\delta\boldsymbol{\theta}, \delta\mathbf{t}]^T \in \mathbb{R}^6$, where $\delta\boldsymbol{\theta}$ represents rotation, the Jacobian of the residual with respect to $\delta\mathbf{x}$ is:

$$\mathbf{J}_i = \begin{bmatrix} -[\mathbf{p}'_i]_\times & -\mathbf{I}_3 \end{bmatrix}$$

where $[\mathbf{p}'_i]_\times$ is the skew-symmetric matrix of $\mathbf{p}'_i$.

$$[\mathbf{p}'_i]_\times = \begin{bmatrix} 0 & -p_z & p_y \\ p_z & 0 & -p_x \\ -p_y & p_x & 0 \end{bmatrix}$$

This gives the linear system

$$\mathbf{H} = \sum_i \mathbf{J}_i^\top \mathbf{J}_i, \quad \mathbf{b} = \sum_i \mathbf{J}_i^\top r_i$$

with the total squared error:

$$\mathbf{E} = \sum_i \|r_i\|^2$$

3. Solve the obtained linear system either by using a Gauss Newton Iteration or by performing a Levenberg Marquardt Iteration depending on which non-linear least squares method you want to use. Use the obtained result to update the current transform

$$\mathbf{R} \leftarrow \exp([\delta\boldsymbol{\theta}]_\times)\,\mathbf{R}, \quad \mathbf{t} \leftarrow \mathbf{t} + \delta\mathbf{t}$$

then check if we have convergence

$$\|\delta\boldsymbol{\theta}\|_2 < \epsilon_\theta, \quad \|\delta\mathbf{t}\|_2 < \epsilon_t$$

where ϵ_θ and ϵ_t represent the parameters for the desired rotational and translation convergence criteria respectively. [1]

2.3.3 GICP

The GICP formulation is built on the same principle as ICP but instead of using the normal point to point matching and residual we instead assume that all points are sampled from what is locally a plane and uses each points covariance regularized to a plane which is essentially the normal, combined with using the covariance weighted Mahalanobis distance instead of euclidean distance. This gives us the optimization problem

$$\min_{\mathbf{R}\in SO(3),\, \mathbf{t}\in\mathbb{R}^3} \sum_{i=1}^{N} \left\| \mathbf{q}_{c(i)} - (\mathbf{R}\mathbf{p}_i + \mathbf{t}) \right\|_{\mathbf{E}_\xi}^2$$

where

$$\|x\|_{\mathbf{E}_\xi} = \sqrt{x^T \mathbf{E}_\xi^{-1} x}$$

and where the covariance is the sum of the corresponding target point covariance and the transformed source point covariance

$$\mathbf{E}_\xi = \mathbf{C}_{q_{c(i)}} + \mathbf{R}\mathbf{C}_{p_i}\mathbf{R}^T$$

The actual steps and optimization part of the algorithm remains the same but using this formulation has significantly better convergence behavior and is therefore significantly faster. However, one additional step required for GICP is a way to calculate the covariance for each point and then the regularization of the covariance matrix [10].

Principal Component Analysis and Regularization

In order to determine the covariances of a point Principal Component Analysis is performed on the k nearest points to the point in question. We first compute the mean of these nearest points and then we calculate the covariance of these closest points in order to get an estimation of the covariance. We can then perform singular value decomposition of the covariance matrix

$$\mathbf{C} = \mathbf{U}\mathbf{E}_\xi\mathbf{V}$$

where $\mathbf{E}_\xi$ contains the singular values of the matrix. In this case the singular values of the covariance matrix essentially represents the principal components of the local point cloud and how strongly the points vary in each of these principal directions [6]. Since we are working under the assumption that all points are sampled from what is locally a plane we expect that two of these singular values will be really big, representing the points being distributed in the plane, and one singular value being really small representing movement in along the normal

which should be very small for a plane. We find this smallest singular value and create a new singular value matrix

$$\mathbf{D} = \begin{bmatrix} \epsilon & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where we replace the smallest singular value with a small number ϵ for example $\epsilon = 0.001$ and fill the rest with ones in order to make the resulting covariance matrix have a very small covariance along the normal of the plane and a high one along the plane. We then reconstruct the regularized covariance matrix as

$$\mathbf{C}_{\mathcal{R}} = \mathbf{UDV}$$

where the covariances have been regularized to fit onto the best plane approximation [10].

2.4 GPU

2.4.1 Kokkos

One challenge regarding GPU programming is the non-uniformity in the API used for programming the GPU and HPCs in general. For programming on large amount of CPU cores OpenMP or threads are common, on NVIDIA GPUs CUDA is used and for AMD GPUs HIP is used. Another disadvantage with many of these programming frameworks is that it requires writing in a separate language that often has a cumbersome learning curve and which all have different syntax's and models for memory and synchronization. In order to solve this problem there are many unifying programming frameworks that create an API on top of the device specific APIs in order to be able to be compatible with all or most devices [13].
Kokkos is one such device agnostic framework for programming HPC applications in a device agnostic way. Kokkos has the advantage that instead of creating a separate language that is capable of being run on device, everything is written in mostly normal `C++`. The way this is done in Kokkos is that functions intended to run on the device are annotated and expressed as `C++` functors or lambdas, which are then passed to parallel dispatch constructs such as parallel_for, parallel_reduce, and parallel_scan. Instead of writing separate kernel code, Kokkos relies on `C++` template meta programming and execution policies to map the same source code to different backends. At compile time, kernel code is generated for the specific backend in use. Kokkos uses this same trick in order to represent memory across different devices through templated memory spaces. This allows for `C++` code written in Kokkos to be compiled to any backend without any code changes and with low overhead since specific optimizations are done depending on the backend compiled for [13]. Example of Kokkos inline `C++` code which can be run on any device by changing the KokkosSpace and KokkosCompute variables 2.1.

```
//memory and execution spaces to run on
using KokkosSpace = Kokkos::CudaSpace;
using KokkosCompute = Kokkos::Cuda;

int main() {
```

```
    //setup Kokkos enviroment
    Kokkos::initialize(argc,argv);{

    //allocate data on the CUDA device
    const int N = 10000;
    Kokkos::View<double*,KokkosSpace> data("data", N);

    //perform a parallel for loop on the CUDA device
    Kokkos::parallel_for("Initialization", Kokkos::RangePolicy<
        KokkosCompute>(0, N), KOKKOS_LAMBDA(int i) {
         data(i) = 0.1;
    });
    //Wait until all kernels are finished executing
    Kokkos::fence();
    //clean up Kokkos enviroment
    }Kokkos::finalize();
}
```

Listing 2.1: Kokkos Parallel Loop Example

2.4.2 ArborX

In order to be able to perform efficient kNN lookup on GPU ArborX was used as a Kokkos implementation of a BVH. In difference from a naive implementation of a kNN which tries all possible combinations resulting in a complexity of $O(n^2)$ when using a BVH one can achieve a complexity of $O(n * log(n))$ allowing for a significant speedup. ArborX is a library containing a device portable implementation of BVHs specifically designed for HPCs allowing high performance and a high degree of parallelism on GPU significantly speeding up. In difference from typical GICP designed for CPUs using kdTrees or Rtrees, BVHs are used in the GPU implementation as it is more adapted to GPUs and as ArborX is a very optimized implementation of kNN using BVHs for Kokkos [5]. See figure 2.3 for an example of the internal representation used by ArborX.

2.5 Implementation

2.5.1 GICP

In order to run GICP on GPU the HPC programming framework Kokkos was used to allow for portability and ease of writing. The registration algorithm is based on GICP and constructed in mostly the same way except all the data is represented on the GPU through Kokkos Views and run through device kernels. The most performance critical part of this algorithm is the kNN lookup used for creating the correspondences which is done using the ArborX library.

Hessian and Gradient construction

Other than the kNN lookup another performance intensive part which can cause a large bottleneck is the construction of the hessian and gradient needed to solve the equation for

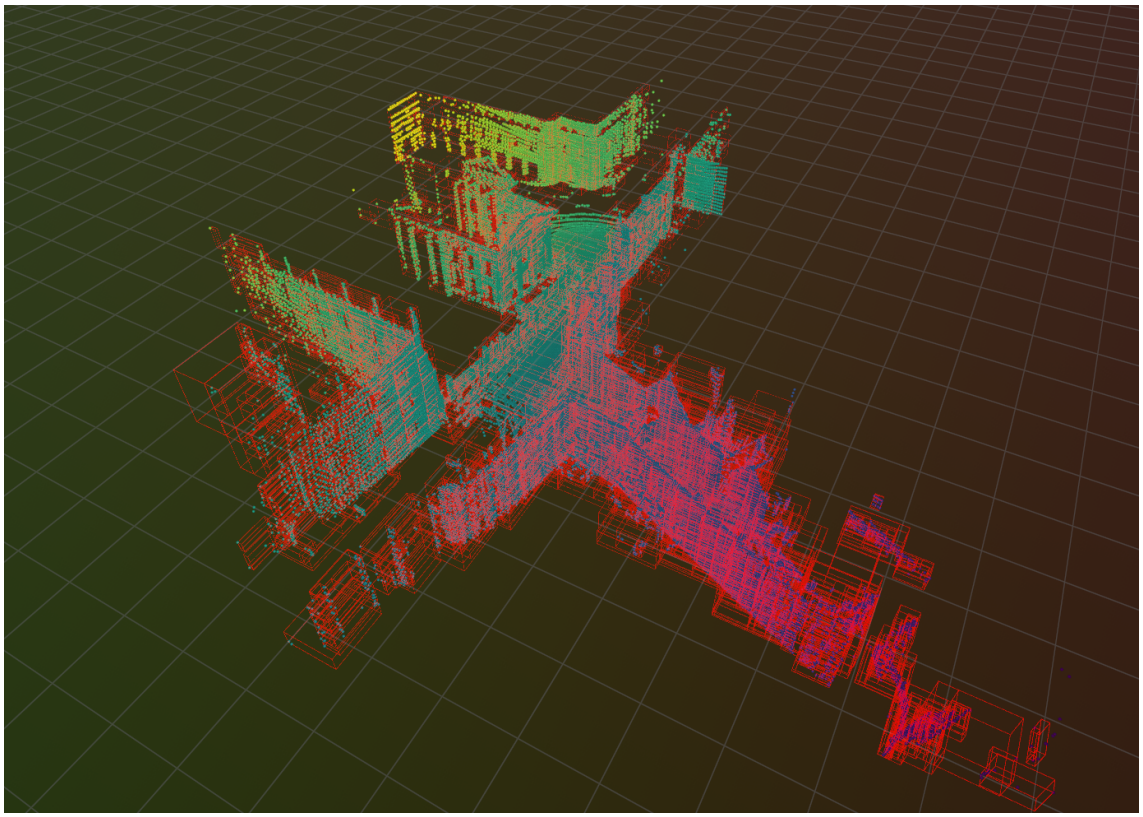


Figure 2.3: BVH representation of pointcloud largest and smallest boxes are removed for clarity

the optimal transform for the current set of correspondences. This can be done efficiently on the GPU by creating a parallel reduction over all points through Kokkos Parallel_reduce which reduces the overhead otherwise created through other synchronization mechanisms such as using atomics.

Covariance Calculation

When it comes to the actual correspondence creation and the covariance estimation the problem is entirely parallel and only dependent on the speed of the kNN lookup and therefore the implementation is not of interest. However, due to the fact that the covariance estimation also has an effect on the convergence of the algorithm if you can create a better estimation though better regularization or covariance estimation that gives faster convergence one can reduce the runtime or improve the accuracy of the algorithm.
Let the (clamped) eigenvalues of the covariance matrix be

$$\lambda_i \leftarrow \max(\lambda_i, \varepsilon), \quad \varepsilon = 10^{-3}$$

and let

$$\lambda_{\min} = \min(\lambda_0, \lambda_1, \lambda_2), \quad \lambda_{\max} = \max(\lambda_0, \lambda_1, \lambda_2)$$

Let kappa be a heuristic representation of how non-planar the covariance is

$$\kappa = \frac{\lambda_{\min}}{\lambda_{\max}}$$

we can then rescale the covariance matrix after regularization

$$\mathbf{C} \leftarrow \kappa \mathbf{C}$$

The logic behind this rescaling is that since we regularize all covariances to be planes since all points are ideally sampled from a plane. However, by scaling up covariances that are far away from being planes we are more likely to trust parts of the point cloud that are sampled from actual planes and less likely to trust parts that aren't planar which are more likely to consists of noise.

2.5.2 kNN

A fast kNN search can make or break an LIO algorithm and making it parallel on the GPU is nontrivial to do efficiently. A brute force method was first implemented however smarter alternatives exists. On GPU BVHs are typical and highly optimized for GPU and therefore the Kokkos implementation ArborX was used. Due to the difficultly of implementing a fast NN lookup and due to the lack of available kNN lookup libraries ArborX was the clear choice and Kokkos was specifically selected with ArborX in mind.

2.5.3 Mapping

In order to reduce the amount of data transferred between the GPU and the CPU the mapping needs to be done on GPU as it would otherwise cause a huge transfer overhead on many

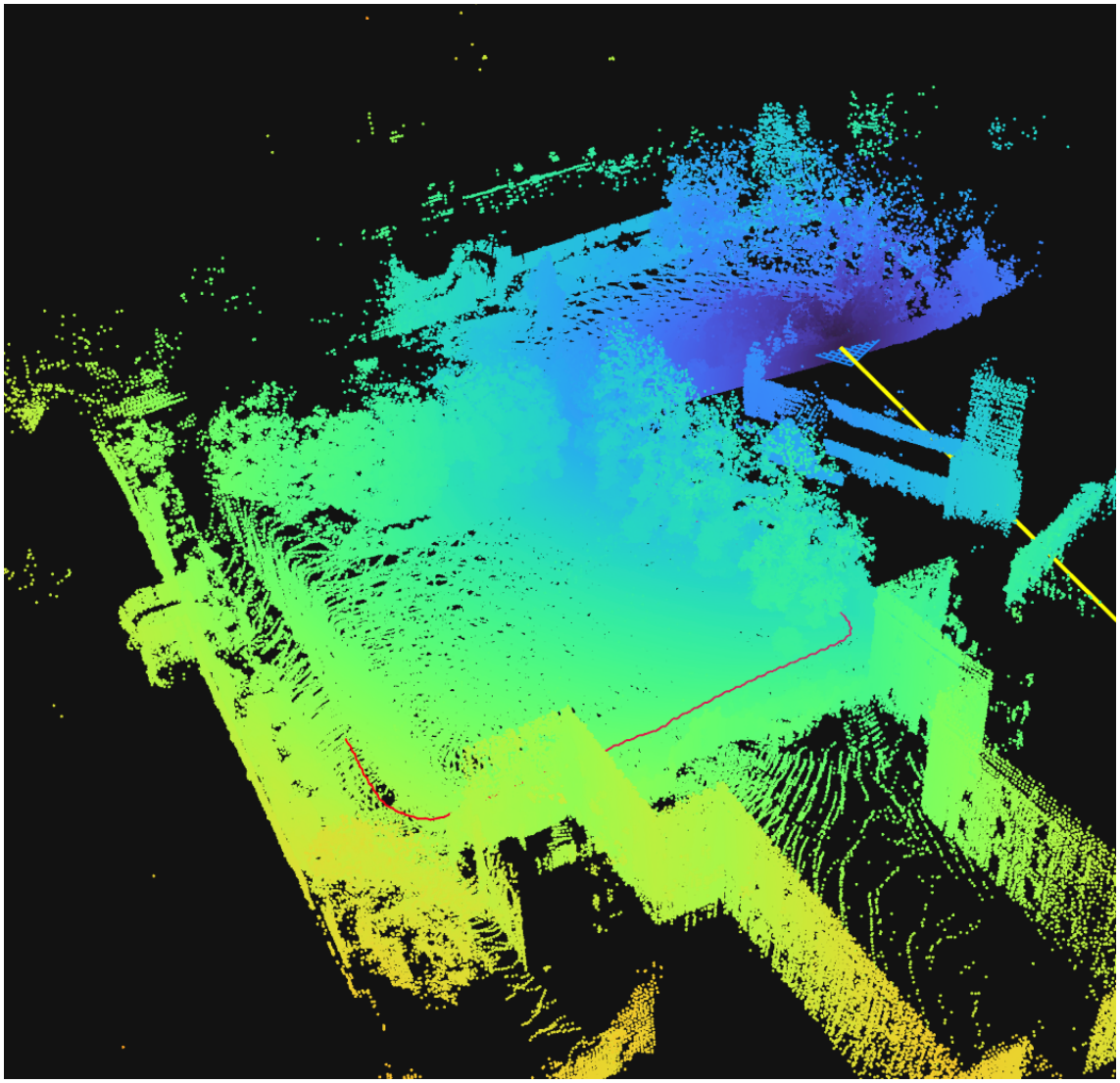


Figure 2.4: Map constructed by GPU DLIO

scans and while constructing the sub-map. By always keeping the point cloud and the covariances for the keyframes generated by DLIO stored on GPU we can avoid this transfer overhead entirely. This allows DLIO to run it's point cloud processing to run entirely on GPU with only the initial transfer to the GPU needed. This also allows new sub-maps to be constructed very quickly siFnce point cloud and covariances for the sub-map can be joined on GPU and due to the efficiency of ArborX:s BVH construction it can be done very quickly. Only the actual underlying representation of the keyframes was changed and all the actual keyframe selection logic was therefore able to remain the same even while running on GPU. See figure 2.4 for an example of a map constructed.

2.5.4 Deskewing

Since a LiDAR sensor is moving while you are taking a scan one will need to deal with the motion-distortion that this movement causes which often ends up skewing the point cloud,

at least if you are moving at a constant speed. A LiDAR scans both vertically and horizontally and it takes one horizontal scan at a time and then moves the laser array to the next step. In order to correct for this we therefore need to know where the LiDAR was during each individual Horizontal line so that we can correct its position. This is done by using the IMU since it can provide accurate short term velocity estimates by integrating the acceleration and angular velocity.

In DLIO this is done by first sorting the points by their timestamps in order to be able to extract every single unique time in a given scan so that the position estimate can be calculated based on these timestamps. Then the IMU is integrated up until every single timestamp that was present in the scan, if a timestamp is in between two IMU timestamps partial integration is done such that a continuous estimate for each position is obtained. This is very accurate but it's hard to do quickly since you need sorting and then a correction on all sorted points, this also makes it difficult to run on GPU due to the sorting. A discrete estimate can be used instead by finding the time span of the scan and integrating up until n different equally distributed timestamps with in the time span of the scan. This can then be used to interpolate between the two closest timestamps in order to get a discrete estimate of the position. This approach gives slightly lower accuracy; however, as long as you sample enough points it will be negligible; however, it does still provide a significant speed up and allows it to be run on the GPU more easily.

2.5.5 Voxelization and Downsampling

One major performance bottleneck for most registrations algorithms is how performance scales with the number of points and the fact that if certain regions contain a high density of points they can be unnecessary since they contain the same information. In order to reduce this somewhat unnecessary work we can down sample the point cloud in order to have a smaller number of points that still represents the entire scan. This can be done via a process called voxelization where you divide the scan into $n \times n \times n$ cubes and represent multiple points that are in the same cube by their centroid [9]. This can be efficiently done on GPU calculating the voxel they are located in by calculating $\lfloor \frac{x}{n} \rfloor$ which will give three numbers representing the voxel. We then create a hash function based on the three coordinates of the voxel and put it into an unordered map, which there exists an efficient GPU implementation of in Kokkos. We can then extract all the centroids of each occupied slot in the unordered map and reconstruct the down sampled point cloud. There are alternative ways of constructing this down sampling such as sorting the point cloud by the hash key which reduces contention however, no other method tried was faster in testing. This downsampled scan is only used for the purposes of odometry and since one of the outputs of the LIO algorithms is a corrected pointcloud the non-downsampled deskewed and position corrected pointcloud is outputted from the LIO algorithm in order to not lose information in the output. A comparison between a downsampled pointcloud and a non-downsampled pointcloud can be seen in figure 2.5 and figure 2.6 respectively.

2.5.6 Profiling

In order to identify which part that was the bottleneck in the algorithm and in order to identify any potential slow code profiling was used. This was done using several tools and

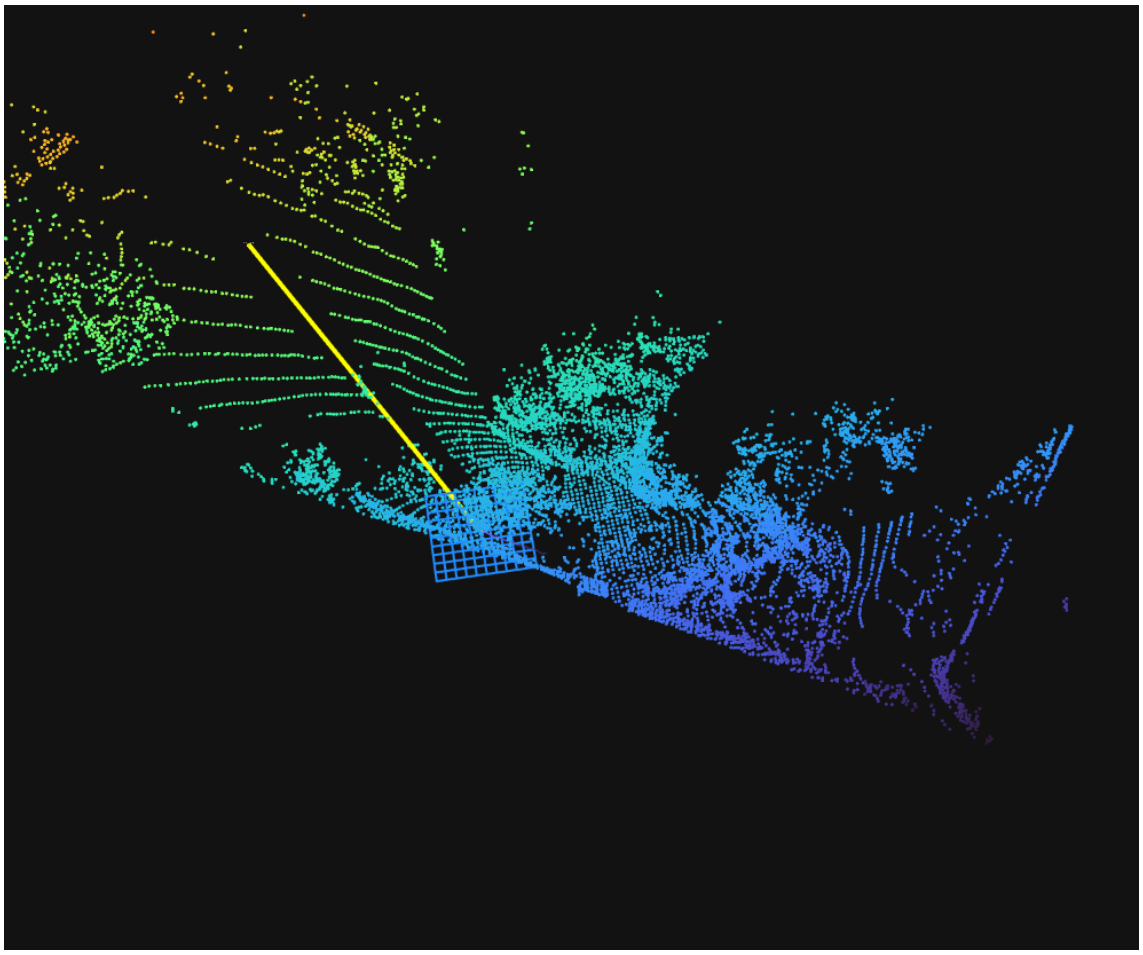


Figure 2.5: Pointcloud downsampled using a voxel size of 0.5

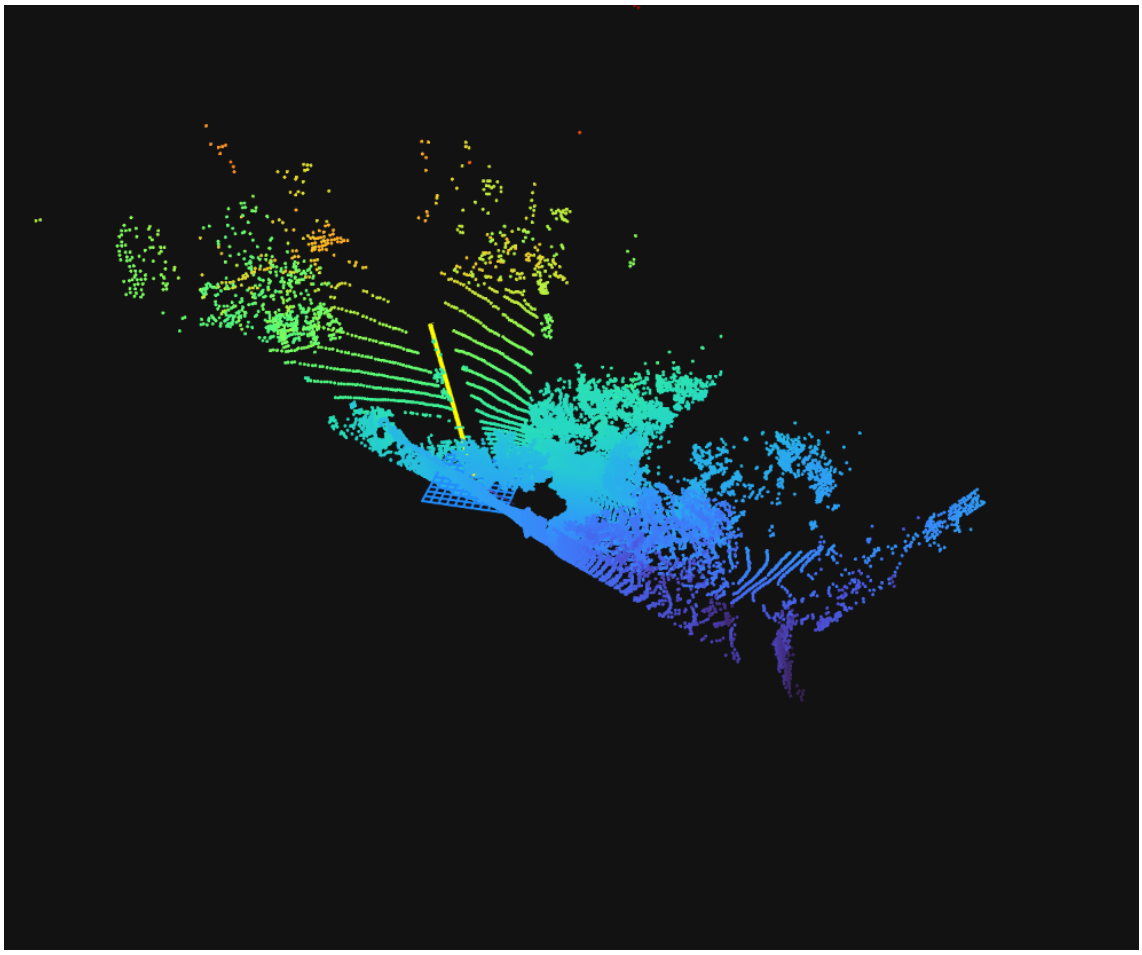


Figure 2.6: Pointcloud with no downsampling

methods, the main tool used was Kokkos own profiling tools which provide info about which kernels are taking a large percentage of the total runtime and other statistics such as memory usage. In addition to this there are tools provided by NVIDIA to allow for more granular and extensive profiling, however this is quite hard to use in Kokkos since you will mainly see internal Kokkos handling instead of your actual code which makes it hard to parse. In addition to this explicit instrumentation e.g. code timings were also used to determine which sections were slower.

2.6 Validation

2.6.1 Data Sets

In order to evaluate both GICP:s and LIO:s performance a dataset with LiDAR data is required. However, evaluating just the performance is not enough since we also need to know how accurate the algorithms are, therefore we also need data with ground truth so that we can compare the true trajectory or transform with our algorithms estimate. In the case of validating LIO, IMU data is also necessary. One additional caveat is that we require data from varying terrain since the structure of the point clouds can have a large effect on the results.

For evaluating both accuracy and the performance of GICP the Oxford Spire dataset was used. This dataset contains both LiDAR scans as well as IMU data. In addition to this they also provide ground truth data used for evaluating the accuracy of the algorithm. However, one problem with this dataset is that it mainly contains one type of environment consisting of buildings and mainly flat ground. [12]

For evaluating LIO the dataset need to have the LiDAR scans be synchronized with the IMU and thus a dataset with an ouster LiDAR was used since they provide this by default. Therefore the Newer Collage dataset was used for evaluating the performance of the LIO algorithm [8].

For more challenging data containing a different environment data from NFA forestry machines were used. This data has both IMU and LiDAR scans. However, ground truth data is not present in this recording. This data was mainly recorded in forests on uneven terrain giving a different challenge compared to data recorded in simpler terrain. One caveat with this data is that accuracy can't be determined and it can only be used for performance evaluation since no ground truth data exists for the data.

2.6.2 Hardware

Since a GPU based implementation and a CPU based implementation is compared the comparisons between the implementations are not necessarily true when running on other hardware since it's dependent on the difference of speed between the GPU and the CPU. These evaluations where done on a NVIDIA Jetson Orion which offers a NVIDIA Jetson Orin NX 16GB GPU and an ARM Cortex-A78AE CPU.

2.6.3 Point Cloud Registration Validation

In order to validate point cloud registration algorithms in a way that is consistent with the way it is used in typical LiDAR inertial odometry we need to create a map for the algorithms to compare each scan to. The map and its accompanying scan are constructed by combining a subsets of all current scans. These scans are first transformed to the ground truth's frame by looking at where the LiDAR is by using the ground truth data provided. Then every time the LiDAR has moved one meter or more we add this scan to the current map, such that the map is built incrementally in the same way that typical LIO algorithms do. Then for the scan we want the algorithm to match with the map we transform it based on the ground truth information from the previous scan. This gives us the current scan based on where we think the LiDAR is currently i.e at the previous scans position. Then when the algorithm is validated we take the relative transform between the scans real position and the old transform that we used to transform it. This gives us both a ground truth map as well as a scan taken from where a LIO algorithm would think the scan is placed i.e at the latest registered position. Another advantage given by this approach is that as the recording goes longer the map grows as more scans are added to it which allows validation of the algorithms at different point cloud sizes.

Accuracy Metrics

In order to evaluate the accuracy of the point cloud registration algorithm we need to calculate the distance error and the angle error between how the ground truth moves the LiDAR compared to how algorithms estimates the movement of the LiDAR which is calculated in the following way. Let the ground truth pose of a scan in the map frame be:

$$\mathbf{T}_{\text{GT}} = \begin{bmatrix} \mathbf{R}_{\text{GT}} & \mathbf{t}_{\text{GT}} \\ \mathbf{0}^{\top} & 1 \end{bmatrix} \in SE(3)$$

where $\mathbf{R}_{\text{GT}} \in \mathbb{R}^{3\times3}$ is the rotation and $\mathbf{t}_{\text{GT}} \in \mathbb{R}^3$ is the translation. The GICP-estimated transform aligning the scan to the map is:

$$\mathbf{T}_{\text{ICP}} = \begin{bmatrix} \mathbf{R}_{\text{ICP}} & \mathbf{t}_{\text{ICP}} \\ \mathbf{0}^{\top} & 1 \end{bmatrix} \in SE(3)$$

. Let $\mathbf{p}_{\text{prev}}$ be the previous scan position. Then transformed positions are:

$$\mathbf{p}_{\text{ICP}} = \mathbf{R}_{\text{ICP}}\,\mathbf{p}_{\text{prev}} + \mathbf{t}_{\text{ICP}}$$
$$\mathbf{p}_{\text{GT}} = \mathbf{R}_{\text{GT}}\,\mathbf{p}_{\text{prev}} + \mathbf{t}_{\text{GT}}$$

We can then compute the euclidean error comparing the ground truth movement of the sensor origin and the point cloud registration algorithms estimated movement of the sensor origin as:

$$e_{\text{dist}} = \|\mathbf{p}_{\text{ICP}} - \mathbf{p}_{\text{GT}}\|_2$$

Additionally we calculate the transforms angle error as:

$$\mathbf{R}_{\text{Rel}} = \mathbf{R}_{\text{GT}}\,\mathbf{R}_{\text{ICP}}^{\top}$$
$$e_{\theta} = \arccos\Big(\frac{\text{tr}(\mathbf{R}_{\text{GT}}) - 1}{2}\Big)$$

Over N scans, the average distance and RMSEs are:

$$\text{RMSE}_{\text{dist}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(e_{\text{dist}}^{(i)})^2}, \quad \text{RMSE}_{\theta} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(e_{\theta}^{(i)})^2}$$

Timing Metrics

To evaluate the performance of the different components used in the evaluated point cloud registration algorithms, we compute the mean execution time for each stage of the GICP algorithm: source construction, target construction, and matching. For our use case the mean matching time and the mean source construction time is of most importance. The reason for this is that the target construction time which is the map and therefore a much larger point cloud is not reconstructed every time when used in LIO since we can keep the calculated normals from when the point cloud was first used as the source.

2.6.4 LIO Validation

Accuracy Metric

For validating LiDAR Inertial Odometry we do not estimate the position over a single scan since we need it to stay consistent after a large amount of scans and IMU data. In order to validate the LIO algorithm, the trajectories generated from the algorithm when running it on a recording of IMU and LiDAR data and compare it to the ground truth trajectory from the same data, for example of a part of a trajectory see figure 2.7. We can then calculate the absolute trajectory error which measures the root mean square position error. Let $\mathbf{p}_{\text{lio}}^{i}$ be the i:th position as calculated by the LIO algorithm and $\mathbf{p}_{\text{gt}}^{i}$ be the true position.

$$\text{ATE}_{\text{RMSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left\|\mathbf{p}_{\text{lio}}^{i} - \mathbf{p}_{\text{gt}}^{i}\right\|^2}.$$

Timing metrics

For evaluating the performance of the LIO algorithm the time taken to process a point cloud is most important since it dominates the compute usage of the algorithm. This timing is split into several parts based on the different steps that take a significant amount of time in the algorithm which are the de-skewing, the source construction and the matching times. The target construction occurs on a separate thread which is also timed however, it is not significant since it is infrequent. In addition, since this runs on a real-time system we are interested in how much of the available CPU resources it utilizes which is measured by the mean CPU load.

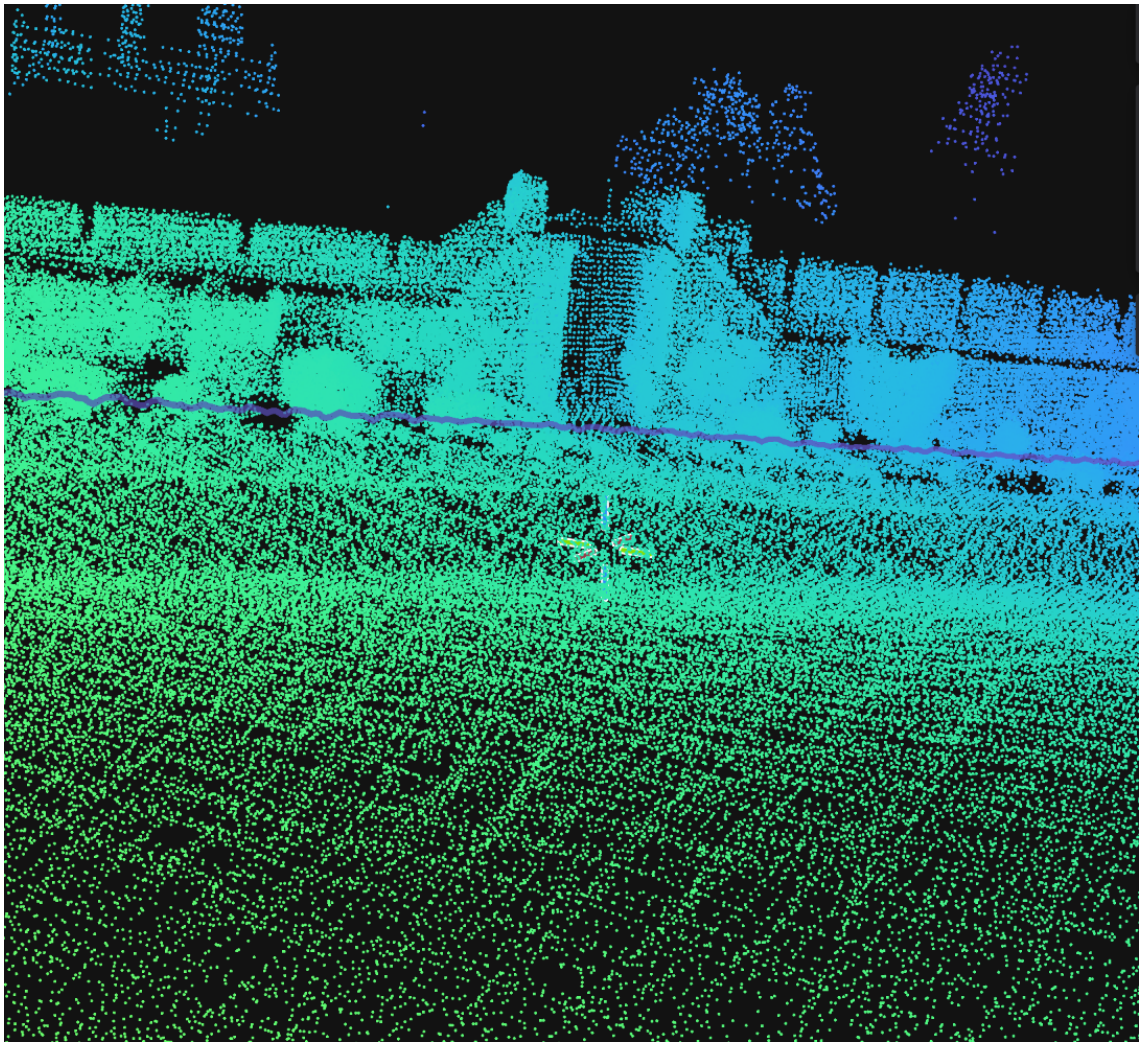


Figure 2.7: Example of a trajectory (oscillation are from the person holding the LiDAR walking)

Chapter 3

Results

3.1 Point Cloud Registration Results

The point cloud registration algorithms were evaluated independently of the LiDAR inertial odometry algorithm on both a Jetson of the same model that NFA uses in the field as well as a Laptop with a powerful CPU and GPU. The algorithms was compared with all of the algorithms offered by the Fast GICP package and the Small GICP package as well as DLIO:s GICP implementation called Nano GICP. The point clouds that were evaluated were extracted from the oxford spire dataset and were taken spaced roughly 1 meter per scan and the map used was based on the combination of the previous 1 meter spaced scans. The algorithms were evaluated on one set of 500 small point clouds were the map consists of the two most recent point clouds added to the map and therefore keeping the map size at around 60k with a scan size of 30k. The algorithm was also evaluated on a dataset of 500 large point clouds where the map encompassed the last 30 scans, giving a point cloud size of 900k and a scan size of 30k. All algorithms ran on the same setting for parameters where possible, for example correspondence distance and the number of points to estimate covariances with were 1.5 and 20 respectively, this could otherwise have a huge performance and accuracy effect and they were not tuned in any way to not misrepresent any algorithms.
Based on the results from the small point cloud set we can see that the GPU GICP implementation is dominating in the matching times both on the laptop GPU and Jetson. We can also see that where my implementation loses is in the construction time where Small GICP dominates see figure 3.1 and figure 3.4. The slowness of the construction for my implementation is even more noticeable on the Jetson due to its proportionally weaker GPU compared to its CPU when compared with the laptop GPU and CPU.

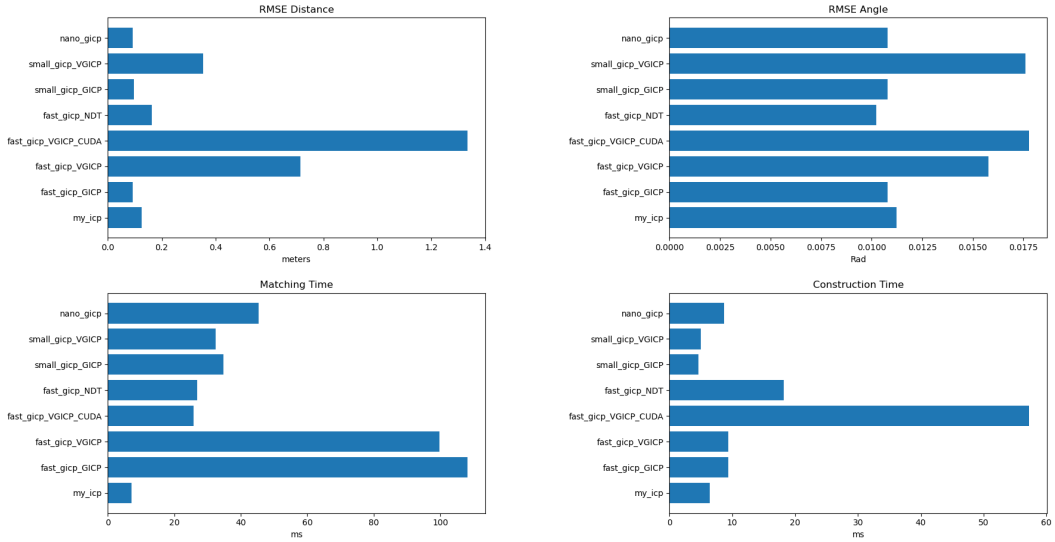


Figure 3.1: GICP evaluation on 500 small point clouds 60k map size on i7-12700H and RTX 3080 Mobile Max-Q

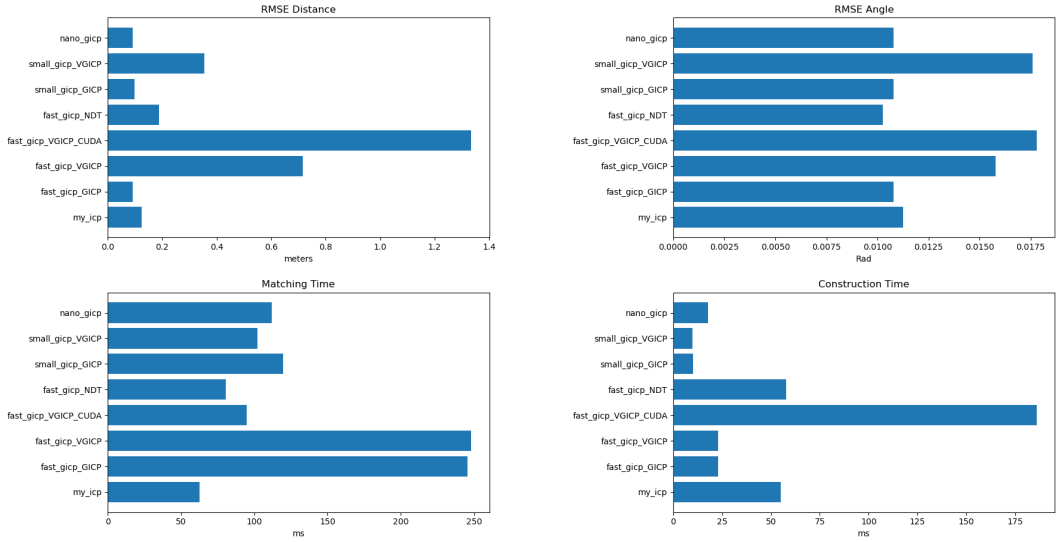


Figure 3.2: GICP evaluation on 500 small point clouds 60k map size on Jetson with Cortex-A78AE and NVIDIA Jetson Orin NX 16GB

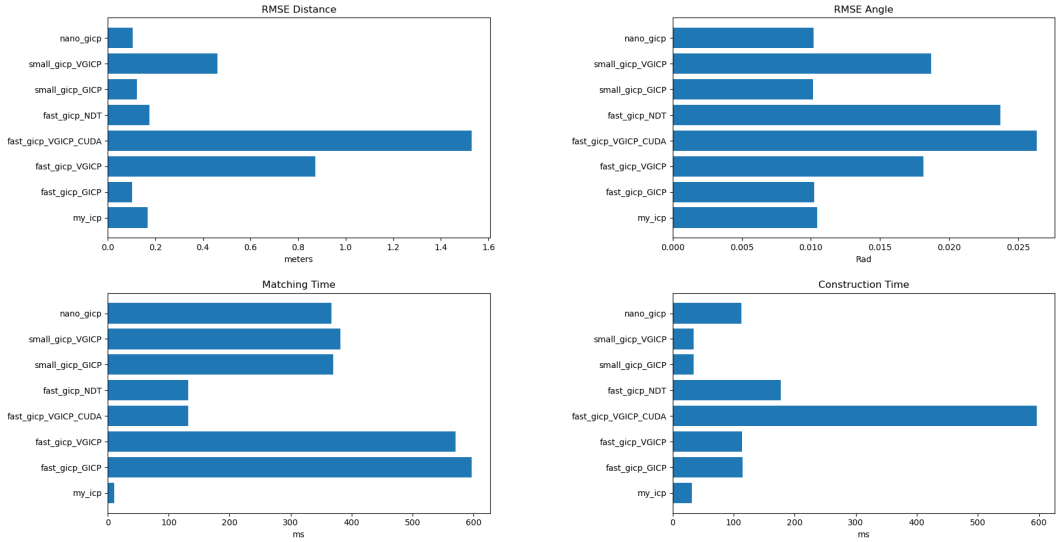


Figure 3.3: GICP evaluation on 500 large point clouds 900k map size on i7-12700H and RTX 3080 Mobile Max-Q

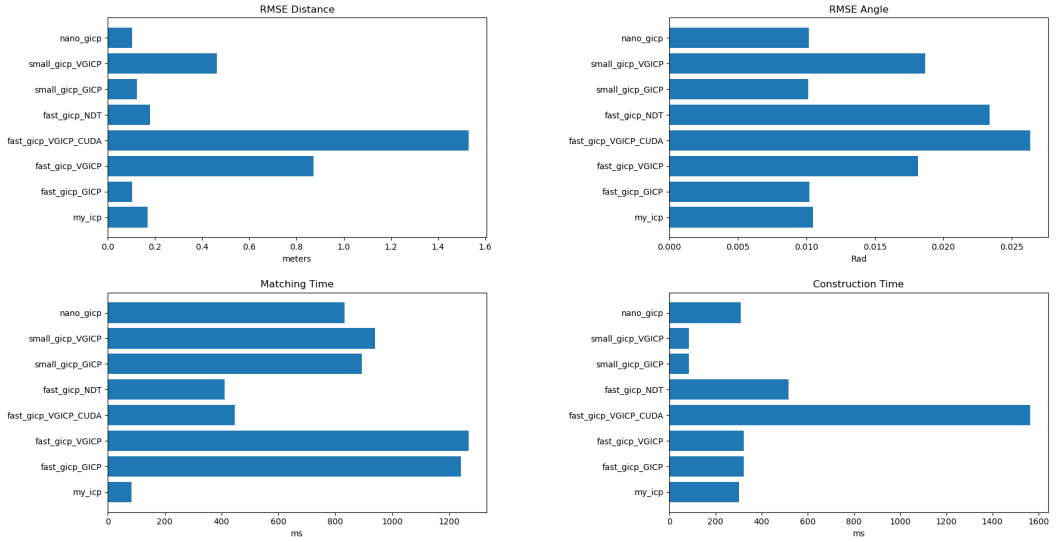


Figure 3.4: GICP evaluation on 500 large point clouds 900k map size on Jetson with Cortex-A78AE and NVIDIA Jetson Orin NX 16GB

3.1.1 Downsampling

In order to determine if downsampling has any negative effects on accuracy or positive effects on performance the results with and without downsampling were compared see table 3.1.

Metric	Downsampling On (0.25 voxel size)	Downsampling Off
RMSE Distance Errors (m)	0.1680	0.1365
RMSE Angle Errors (Radians)	0.0101	0.0105
Mean Source Construction Time (ms)	3.3534	4.1648
Mean Target Construction Time (ms)	6.2836	28.2473
Mean Matching Time (ms)	7.0635	10.9193

Table 3.1: Comparison of Performance: Downsampling On vs. Off

3.2 DLIO Evaluation

DLIO was evaluated according to its time to process each scan and the trajectory error of the estimation along with the average CPU usage, this was done on several recordings from the oxford spires dataset and the averaged results are presented in table 3.2.

Platform	Impl.	Time / Frame (ms)	ATE (m)	CPU utilization
i7-12700H	CPU	29.67	0.2333	268%
RTX 3080	GPU	10.76	0.2171	22%
Cortex-A78AE	CPU	116.26	0.2955	299%
NVIDIA Jetson Orin NX 16GB	GPU	70.76	0.2604	50%

Table 3.2: Average DLIO Evaluation Results (Oxford Spires Dataset)

For the forest recording provided by NFA, the results were for running on the laptop with i7-12700H and RTX 3080 Mobile Max-Q. In this case the only metrics are CPU usage and frame time since no ground truth data is available for this recording. However, the resulting maps and trajectory were still inspected to check that they were reasonable, which they all were. The results are presented in table 3.3.

Platform	Impl.	Time / Frame (ms)	ATE (m)	CPU utilization
i7-12700H	CPU	7.80	N/A	73%
RTX 3080	GPU	10.17	N/A	17%
Cortex-A78AE	CPU	68.10	N/A	240%
NVIDIA Jetson Orin NX 16GB	GPU	60.02	N/A	51%

Table 3.3: DLIO Evaluation Results (NFA Forest Dataset)

Chapter 4
Discussion

4.1 Point Cloud Registration Algorithm

We can see based on the results that utilizing the GPU has the potential to give a big performance increase but we can also see it's dependent on the relative performance of the GPU and the CPU. My ICP had the fastest matching times in all cases which indicates that it will be able to have a much higher frame rate when utilized in a LiDAR Inertial Odometry algorithm. We can also see that the construction time for the GPU based algorithms loses out to the non-GPU based on the Jetson but is able to perform at the same pace as the best CPU algorithms in the Laptop case. Another caveat regarding the Construction time results is that it is not a good indicator of the speed that a LIO algorithm will perform at since in these cases the construction involves the reconstruction the entire map which is only done when a key frame is added instead of every frame when used in DLIO so we would mainly pay the cost of constructing the scans data structures which is smaller than the map. We can also see that the GPU based algorithm scales better on larger point clouds both in terms of the construction time and the matching time.

When looking at the RMSE of the angular error and the distance error we can see several problems. Many of the algorithms have a very bad results in terms of RMSE such as fast VGICP which has an awful RMSE in most cases. This is not mainly due to poor accuracy of the algorithm it is instead dominated by when the algorithm fails to find a reasonable match and gives an extreme error, the results of these algorithms can be very binary when the conditions are hard. The point clouds that are compared are also quite a hard and worst case condition since it's constructed with the scans being of by on average 1.3 m away from its correct position and no initial guess is provided which is a very hard case. We can see that the GPU based GICP performs roughly the same as the other top algorithms in terms of RMSE in all cases, this indicates that they have roughly similar behavior which is the expected results since they are implementing the same algorithms.

The results from evaluating with vs without downsampling shows that it is a clear trade-

off. While the inaccuracy was increased by roughly 21%. The performance was also affected decreasing the matching time by 37% and decreasing construction time by 67%. This tradeoff can be worth it depending on the requirements and by further increasing the voxel size or by decreasing the voxel size we can shift the tradeoff depending on what we desire.

These results show that using a relatively simple but efficient nearest neighbor lookup method such as BVHs can compete in terms of performance with approximate nearest neighbor algorithm such as VGICP and also with algorithms designed for speed on GPU such as NDT even when both of these algorithms are run and made for GPU. One can't draw the conclusion that BVHs are faster than other nearest neighbor methods for this use case since no equivalently optimized implementation exists for GPU for these algorithms however BVHs do show promising results.

4.2 DLIO

4.2.1 Performance

The evaluation of the Direct LiDAR-Inertial Odometry (DLIO) implementations highlights a performance gain when utilizing the GPU version. On the laptop hardware, the GPU implementation reduced the per-frame processing time by nearly a factor of three, dropping from 29.67 ms to 10.76 ms. While the CPU was already able to run CPU DLIO in real time since it was under the 10Hz input rate the LiDARs used it does reduce resources usage and allow for increased point cloud sizes while maintaining real time viability.

On the embedded Jetson which is highly representative of actual field robotics—the CPU implementation averaged 116.26 ms per frame. This exceeds the 100 ms threshold of a 10 Hz LiDAR, meaning the CPU-only version would likely drop frames or fall behind. In comparison the GPU implementation on the Jetson had a processing time of 70.76 ms. This brings the algorithm back into the real-time processing range, showing that the ArborX and Kokkos-based GPU acceleration is a useful improvement for certain use cases.

In the case of forest data we do not see as big of an improvement as we did with other environments see figure 4.1 for reference. This is likely due to the fact that in that dataset half of the field of view of the LiDAR is covered due to mechanical construction. This mean that roughly only half as many points will be registered by the LiDAR. One additional factor that contributes to this is the fact that many of the rays will not hit anything since the trees are much sparser when compared to the typical environment of the Oxford spire dataset which mainly consists of large buildings. This will in turn improve the performance of both algorithms and the overhead of the GPU algorithm competes with the gain. This can most clearly be seen in the laptop comparison where the GPU version decreases the performance from 7.8 ms to 10.17 ms and we can see that this benchmark requires very little compute as the average CPU utilization is only 73% and therefore GPU acceleration mainly contributes to higher overhead slowing down the algorithm. In the Jetson case due to the fact that its CPU is much slower the GPU is still able to compete since it still requires a lot of compute for that CPU which we see by the CPU utilization being 240% which in turn explains why the GPU has a slight performance improvement over the CPU.

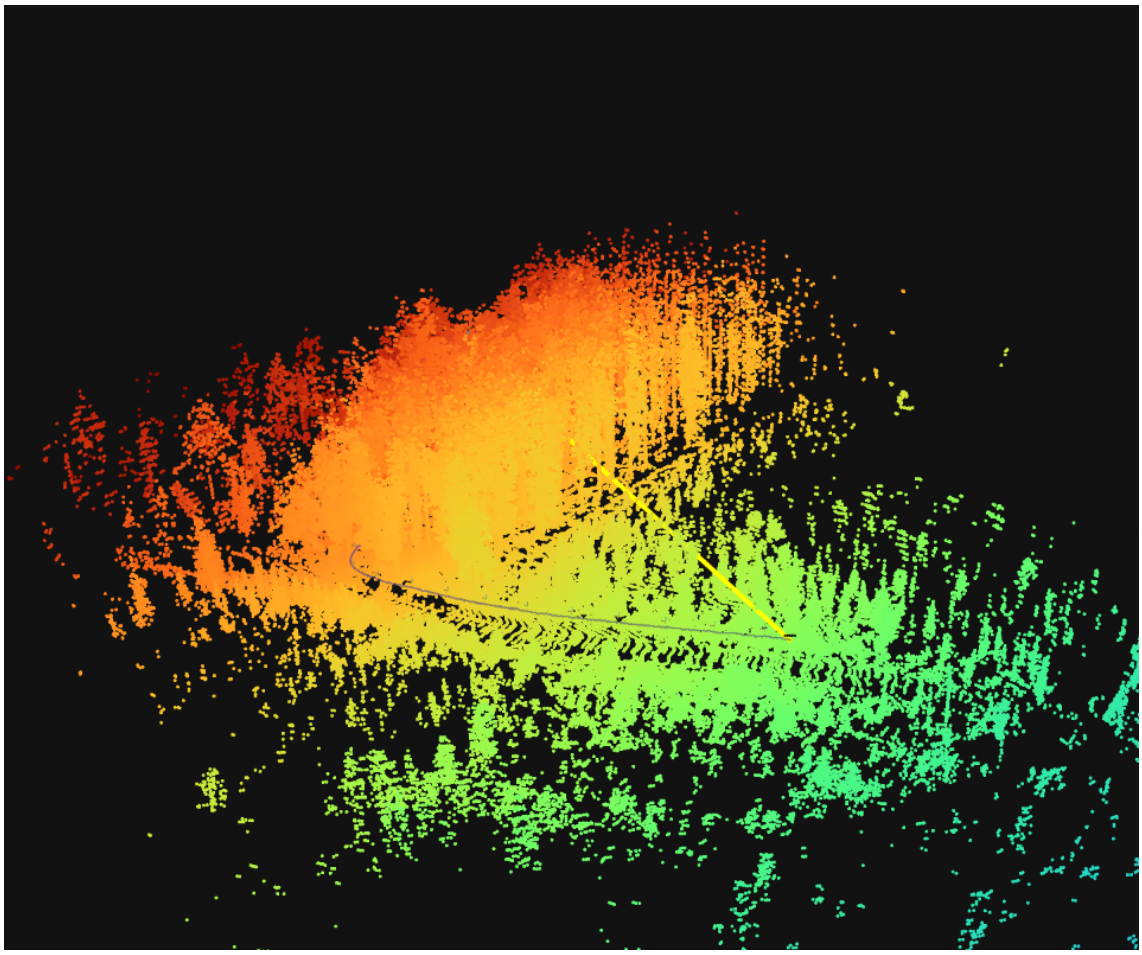


Figure 4.1: Map constructed from the NFA dataset

4.2.2 CPU Offloading

In both the case of the Laptop and Jetson hardware the average CPU core utilization dropped from 2.68 to 0.22 and 2.99 to 0.5 respectively. This shows that utilizing a GPU implementation can provide major offloading of CPU resources which can be of great use in certain computational environments. Another improvement in the same vein is that RAM usage was also decreased since the map is now entirely stored on VRAM instead of the systems RAM providing another potentially useful resource offloading.

4.2.3 Trajectory Accuracy

On both tested devices the absolute trajectory error was slightly improved on the GPU version of the algorithms going from 0.2333 m to 0.2171 m and 0.2955 m to 0.2171 m on the Laptop and Jetson respectively. The reason for the slight improvement is likely due to the faster processing time leading to fewer missed frames or slightly increased accuracy of the point cloud registration algorithm.

4.3 Future Work

While this work does provide a performance improvement in many cases there is still potential for an even faster approach. The most expensive part of the current algorithms is the requirement to construct the covariances for each point. If you were to run point to plane ICP there could potentially be a speed up in the LIO case since you would then only need to compute these covariances for the scans added to the map. So even if this algorithm would traditionally be slower in this use case it could be relevant.

Bibliography

[1] P.J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[2] Kenny Chen, Ryan Nemiroff, and Brett T. Lopez. Direct lidar-inertial odometry: Lightweight lio with continuous-time motion correction, 2023.

[3] Frank Dellaert and Michael Kaess. *Factor Graphs for Robot Perception*. Now Publishers Inc., August 2017.

[4] Kenji Koide. small_gicp: Efficient and parallel algorithms for point cloud registration. *Journal of Open Source Software*, 9(100):6948, August 2024.

[5] D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery. ArborX: A performance portable geometric search library. *ACM Trans. Math. Softw.*, 47(1), December 2020.

[6] Andrzej Maćkiewicz and Waldemar Ratajczak. Principal components analysis (pca). *Computers Geosciences*, 19(3):303–342, 1993.

[7] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, New York, NY, 2. ed. edition, 2006.

[8] Milad Ramezani, Yiduo Wang, Marco Camurri, David Wisth, Matias Mattamala, and Maurice Fallon. The newer college dataset: Handheld lidar, inertial and vision with ground truth. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4353–4360, 2020.

[9] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011. IEEE.

[10] Aleksandr V. Segal, Dirk Hähnel, and Sebastian Thrun. Generalized-icp. In *Robotics: Science and Systems*, 2009.

[11] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, Cambridge, MA, 2 edition, 2011.

[12] Yifu Tao, Miguel Ángel Muñoz-Bañón, Lintong Zhang, Jiahao Wang, Lanke Frank Tarimo Fu, and Maurice Fallon. The oxford spires dataset: Benchmarking large-scale lidar-visual localisation, reconstruction and radiance field methods. *International Journal of Robotics Research*, 2025.

[13] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022.

[14] Wei Xu, Yixi Cai, Dongjiao He, Jiarong Lin, and Fu Zhang. Fast-lio2: Fast direct lidar-inertial odometry, 2021.

EXAMENSARBETE GPU acceleration of LiDAR Inertial Odometry
STUDENT Alve Lindell
HANDLEDARE Michael Doggett (LTH), Martin Ahrnbom(NFA)
EXAMINATOR Jonas Skeppstedt (LTH)

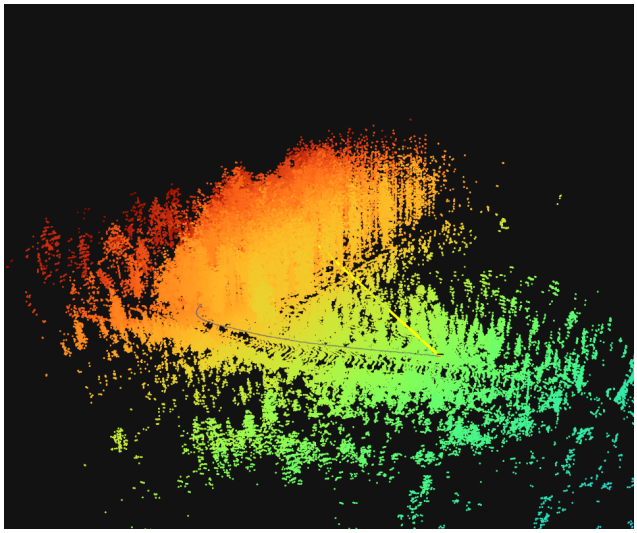
När GPU möter LiDAR, snabbare och smartare positionsbestämning

POPULÄRVETENSKAPLIG SAMMANFATTNING **Alve Lindell**

GPU kan användas för att snabba upp LiDAR-tröghetsodometri, en teknik som hjälper robotar och självkörande system att avgöra sin position i omgivningen. Genom att utföra många beräkningar parallellt kan tidskrävande punktmolnsmatchning accelereras utan att försämra noggrannheten. Arbetet visar hur modern grafikprocessorhårdvara kan möjliggöra snabbare och mer responsiv robotnavigering.

Inom robotik är det av stor viktig att kunna uppskatta var en robot befiner sig med hög precision för att kunna kontrollera roboten. Detta problemet kan lösas med flera olika typer av sensorer och algoritmer. Inom robotik är det vanligt att LiDAR(laser scanners) används då de kan ge dig en bild av robotens miljö i 3d. En LiDAR fungerar genom att skicka ut tusentals laser pulser samtidigt som den roterar 360° i 10Hz. Den mäter sedan hur lång tid det tar för laserstrålen att komma tillbaka till sensorn vilket den använder för att beräkna avståndet. Detta resulterar i ett moln av alla punkter LiDARn har mätt upp som representerar hur miljön ser ut i 3D.

Denna LiDAR sensorn kan användas för odometri med hjälp av metoden vid namn LiDAR tröghets odometri. Denna metoden använder sig av LiDAR sensorer, accelerometerar samt gyroskop för att uppskatta en robots rörelse. Detta görs genom att matcha succesiva laser scanningar för att uppskata hur laser scannern har rört sig. Detta kombineras sedan med accelerations och rotations data från de andra sensorerna. Denna metod ger otroligt hög precision men den har nackdelen att matching av laser scanningar är väldigt beräknings intensivt. Denna matchingen är dock väldigt paralleliserbar och i detta papret så används GPU för att kunna göra hundratals av dessa beräkningar samtidigt och därmed snabba på prestandarden på algoritmen.



Resultatet visar att GPU accelerering kan öka prestandarden hos LiDAR-tröghets odometri utan att ha någon negativ inverkan på precisionen hos algoritmens lokaliserings prestandard.