CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

Department of Measurement

Bachelor's thesis

FPGA-based Processing of LiDAR Data

Filip Kučera

May 2022

Supervisor: Ing. Petr Čížek

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kučera Filip**	Personal ID number:	**492267**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Measurement**		
Study program:	**Open Informatics**		
Specialisation:	**Internet things**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

FPGA-based Processing of LiDAR Data

Bachelor’s thesis title in Czech:

Zpracování dat z laserového skeneru pomocí FPGA

Guidelines:

1) Get familiar with development for FPGA boards such as DE10 nano [1] with a focus on the High Level Synthesis using C code [2].
2) Get familiar with the Ouster OS-0 LiDAR [3] and its communication interfacing.
3) Propose and develop FPGA-based reading of the raw data stream of the LiDAR sensor and its processing to point cloud for single and multiple LiDAR units.
4) Benchmark the developed architecture in terms of processing speed, latency, and power consumption and compare its performance with the baseline CPU-based implementation [4].
5) Investigate existing localization techniques based on LiDAR data processing, such as [5,6] and FPGA implementations [7,8] and select a suitable method of LiDAR-based incremental localization for the deployment on the FPGA.
6) Deploy the selected localization method on the FPGA.

Bibliography / sources:

[1] DE10 nano get started guide, available: https://software.intel.com/com/content/www/us/en/develop/articles/terasic-de10-nano-get-started-guide.html [cited on 2022-14-01].
[2] Intel High Level Synthesis Compiler Pro Edition - User Guide, available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls.pdf [cited on 2022-14-01].
[3] Ouster OS0 lidar documentation, available: https://ouster.com/downloads/ [cited on 2022-14-01].
[4] Ouster OS0 lidar drivers, available: https://github.com/ouster-lidar/ouster_example.git [cited on 2022-14-01].
[5] F. Pomerleau, F. Colas, R. Siegwart, 'A Review of Point Cloud Registration Algorithms for Mobile Robotics,' Foundations and Trends in Robotics, 4 (1): 1–104, 2015.
[6] J. Zhang and S. Singh, “LOAM: Lidar Odometry and Mapping in Real-time,“ Robotics: Science and Systems (RSS), 2(9), 2014.
[7] M. Eisoldt, M. Flottmann, J. Gaal, P. Buschermöhle, S. Hinderink, M. Hillmann, A. Nitschmann, P. Hoffmann, T. Wiemann, and M. Porrmann, “HATSDF SLAM – Hardware-accelerated TSDF SLAM for Reconfigurable SoCs,” European Conference on Mobile Robots (ECMR), 2021.
[8] M. Palieri et al., ”LOCUS: A multi-sensor lidar-centric solution for high-precision odometry and 3D mapping in real-time,” IEEE Robotics and Automation Letters, 6(2):421-8, 2021.

Name and workplace of bachelor’s thesis supervisor:

Ing. Petr Čížek Department of Computer Science FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **09.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until:
by the end of summer semester 2022/2023

Ing. Petr Čížek Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Declaration

I declare that the presented work was developed independently and that I have listed all sources of the information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 20, 2022

. . . .
Filip Kučera

Acknowledgement

I would like to thank Ing. Petr Čížek for supervising me and guiding me through the interesting field of robotics and perception. Also, I'd like to thank my mom for always supporting me.

Abstrakt

Tato práce se zabývá zpracováním dat z 3D laserového skeneru pomocí programovatelného hradlového pole (FPGA). Práce předkládá kompletní návrh pipeline pro zpracování surových dat z 3D laserového skeneru až do podoby point cloudu a využití dat v odhadu odometrie robota, efektivně využívající hradlovou (FPGA) i procesorovou (CPU) část vývojové desky DE10-Nano. Pro zrychlení vývoje jsme využili nástroje High Level Synthesis, umožňující napsat jádro algoritmu v jazyce C++ a přeložit ho do HDL.

Jádrem práce je implementace metody odhadu odometrie založená na použití surových hloubkových obrazových dat z LiDARu za účelem nalezení nejvhodnějšího rešení pro architekturu FPGA a také ukázání slibného směru v této oblasti. Navrhovaná architektura je modulární a není tak limitována na specifický výběr algoritmu nebo senzoru. V závěru práce demonstrujeme funkčnost námi navrhované architektury a ukazujeme, že navzdory využití relativně levného FPGA dokáže soupeřit i s moderními CPU.

Klíčová slova: Programovatelná Hradlová Pole, Odometrie, Point Cloud, High Level Synthesis, LiDAR

Abstract

In this thesis, we propose a general purpose sensor data processing pipeline, implemented as a SoPC utilizing both the FPGA and the CPU part of the DE10-Nano Development Board. The proposed pipeline processes LiDAR data and outputs a 3D point cloud and an odometry estimate. To accelerate the development we used the High Level Synthesis toolchain allowing us to write core algorithm in C++ and translate it into HDL. In this work, we study all the necessary steps of the LiDAR point cloud processing pipeline and we explore an unconventional way of approaching the odometry estimation by using the raw depth image data from the LiDAR, in order to find a best fit solution for the FPGA architecture and also to show a promising direction to the field. We also discuss the steps necessary to gain the biggest advantage from the use of the FPGA over the traditionally used CPU(s). We also show that the pipeline design is modular and thus isn't limited to any specific choice of algorithm(s) or sensor(s). In the last section, we show that the pipeline is functional and that even when using relatively cheap FPGA, the results are competitive with modern CPU(s).

Keywords: Field Programmable Gate Array, Odometry, Point Cloud, High Level Synthesis, LiDAR

Contents

1 Introduction **1**

2 Problem Statement and Background **3**

2.1 LiDAR Depth Image to Point Cloud Transform . . . 3
2.2 Odometry . . . 4
2.2.1 LiDAR Odometry . . . 4
2.2.1.1 Feature Based Methods . . . 4
2.2.1.2 Grid Based Methods . . . 4
2.2.1.3 Dense Methods . . . 5
2.2.2 Features from Accelerated Segment Test . . . 5
2.2.3 Binary Robust Independent Elementary Feature description . . . 5
2.2.4 Least Squares Fitting of Two 3D Point Sets . . . 6

3 Hardware & Software Setup **8**

3.1 Ouster OS0-128 . . . 8
3.2 Field Programmable Gate Array . . . 9
3.2.1 DE10-Nano Development Board . . . 10
3.3 High Level Synthesis . . . 11
3.3.1 Intel HLS . . . 12
3.4 u-dma-buf . . . 12

4 Architecture **13**

4.1 Point Cloud Generator . . . 13
4.1.1 HPS Part . . . 13
4.1.2 FPGA Part . . . 15
4.1.2.1 lidar2pntcloud . . . 16
4.2 LiDAR Odometry . . . 16
4.2.1 Data Preprocessing . . . 17
4.2.2 FAST feature detector . . . 18
4.2.3 Binary Robust Independent Elementary Feature description . . . 18
4.2.4 Feature matching & tracking . . . 18
4.2.5 Feature Registration . . . 19

5 Results **20**

5.1 Latency Experiments . . . 20
5.1.1 Receiveing the LiDAR Data . . . 20
5.1.2 Point Cloud Generation in FPGA . . . 20
5.1.3 Feature Detection in FPGA . . . 22
5.1.4 Feature Description . . . 22
5.1.5 Feature Matching . . . 23
5.1.6 SVD Solver . . . 23
5.1.7 Comparison to the CPU Reference Implementation . . . 24
5.2 Odometry Experiments . . . 25
5.2.1 Translational error evaluation . . . 25
5.2.2 Rotational error evaluation . . . 26
5.3 FPGA Resource Usage, Precision & Frequency Ratings . . . 27

6 Conclusion **30**

References 31

A Content of the Attachment 35

List of Figures

1 Pixels used by the FAST detector. 6
2 Schema of the switched network. 8
3 The used Ouster OS0 LiDAR. 9
4 Simplified schema of the logic block. 10
5 The used DE10-Nano devlopment board. 11
6 Overview of the architecture of the proposed pipeline. 13
7 Results of the proposed point cloud generating pipeline. 14
8 Screenshot of the ModelSim simulation of the lidar2pntcloud component. 16
9 Screenshot of the Quartus Platform Designer showing the SoPC schema. 17
10 Depth image before (top) vs. after (bottom) data skew correction. 18
11 Boxplots of the latency measurements. 22
12 Boxplot of the description times using various hyperparameters. 23
13 Boxplot of the matching times using various hyperparameters. 24
14 Photos of the used robotic platform and total station. 26
15 Trajectories as measured by a total station and estimated by our odometry pipeline. 26
16 The RPE_{trans} of the second trajectory experiment. 27
17 The Clearpath Husky A200™ robotic platform inside a hallway. 28
18 Measured and estimated orientations of the robot along with their respective RPEs. 29

List of Tables

1 Latency statistics 21

List of Abbreviations

ADC	Analog to Digital Converter
AHRS	Attitude Heading Reference Sensor
ALM	Adaptive Logic Module
API	Application Programming Interface
ATE	Absolute Trajectory Error
BRIEF	Binary Robust Independent Elementary Features
CPU	Central Processing Unit
CSR	Control & Status Register
DFF	D-Flip Flop
DMA	Direct Memory Access
FAST	Features from Accelerated Segment Test
FIFO	First In, First Out
FSM	Finite State Machine
FoV	Field of View
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input & Output
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HDMI	High-Definition Multimedia Interace
HPS	Hard Processor System
HW	Hardware
IC	Integrated Circuit
ICP	Iterative Closest Point
IMU	Inertial Measurement Unit
IO	Input & Output
LB	Logic Block
LE	Logic Element
LED	Light Emmiting Diode
LiDAR	Light Detection and Ranging

LUT	Look-Up Table
mSGDMA	Modular Scatter & Gather Direct Memory Access
NIR	Near Infrared
PC	Point Cloud
RAM	Random Access Memory
RGBD	Red, Green, Blue, Depth
RPE	Relative Pose Error
RTL	Register Transfer Level
RaDAR	Radio Detection and Ranging
SDRAM	Synchronized Dynamic Random Access Memory
SLAM	Simultaneous Localization and Mapping
SoPC	System on Programmable Chip
SVD	Singular Value Decomposition
SW	Software
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XOR	eXclusive OR

Chapter 1

Introduction

Light Detection and Ranging (LiDAR) sensors are one of the key elements of robotic sensing hardware. They provide 3D scans of their surroundings in a form of a "cloud" of 3D points, called "point cloud" (PC). Point cloud plays a major role in many higher level robotic algorithms such as localization and mapping, odometry, planning, detection and collision avoidance, to name a few. It allows the robot to perceive the notion of depth, distance and 3D shapes in its surroundings as opposed to the data provided by the often used 2D cameras. Another key benefit to using the LiDAR(s) is the fact that their output is almost completely light invariant, meaning the change in lighting (e.g. during day & night cycles) doesn't affect the perceiveing ability of the sensor, as opposed to cameras in which the same scene can look drastically different under various lighting conditions [1].

The LiDAR doesn't provide only the 3D point cloud though. It produces raw data in a form of a depth map, intensity image and ambient light image. The depth map is usually the most interesting, as it can be fused with a known intrinsic parameters of the LiDAR unit to produce the expected point cloud. The other information provided can also prove useful, especially the intensity image, as it provides an idea of a detected material's reflectance, which can be used to distinguish objects and deepen the understanding of the sensor's surroundings; e.g., in the automotive industry where vehicle registration plates are usually highly reflective and can be used as a clue that the perceived object is a car [2].

Aquiring all of the above described information does come at a cost of highly demanding computations required, as the LiDAR produces data at a rate of hundreds of Mbits per second and to provide us with the point cloud, it needs to be processed first using non-trivial projection calculations. Even when all of the data is processed in time, the resulting point clouds contain such a big number of points that often up to 90% of the points are not considered during the execution of the higher level algorithms [3].

This is where we saw the opportunity for Field Programmable Gate Array (FPGA) technology to be utilized to exploit the highly parallelizable nature of the above described task. The FPGA is a technology that slowly becomes more and more adopted by various fields, robotics included, for its ability to implement custom hardware design for solving specific tasks efficiently, with a comparatively low cost and power usage, which are often among the major considerations of various battery powered systems. The power of the FPGA for the task of generating point clouds comes mainly from the fact that all of the LiDAR measurements are independent of each other and thus can be projected simultaneously, which the FPGA can encompass by implementing multiple point projection components running in parallel. This is in contrast to the traditionally used CPUs, as their nature of execution is "step-by-step"; i.e., point-by-point (excluding the comparatively little parallelism provided by multiple hardware threads). This parallelism provided by the FPGA can in turn bring power consumption and speed improvements, even though the frequency of the FPGA fabric is usually orders of magnitude slower than that of the traditionally used CPUs (50 MHz vs. 4.7 GHz in our case). Also, another key motivation for the usage of the FPGA for this task is to free up the CPU resources, which can then be utilized for the higher level tasks, which often better fit the CPU architecture. This in effect makes the entire system more efficient.

However, the goal of this work is twofold. Not only we aim to process the incomig LiDAR data into a resulting point cloud using the FPGA, we also explore an unconventional way of estimating odometry from the LiDAR data. As described in Section 2.2.1, most ego-motion estimating algorithms utilize the point cloud for that purpose. We, inspired by [1], wanted to explore the odometry computed on the raw, depth image data from the LiDAR, as it's a perfect fit for the FPGA architecture due to its simple, low latency pipeline, low memory requirements, and high paralellizability potential. We thus propose a general odometry pipeline for solving this task in Section 4.2. The goal of the pipeline design was to not be reliant on a specific algorithmic choice but rather to prove the concept of the odometry pipeline as suitable for the FPGA & CPU combination we used in this work.

We structured this work to present the above described goals as follows: Next Chapter 2 states the problem and describes the background necessary to understand the presented problems. Chapter 3 describes the hardware & software setup used to accomplish the set goals, after which the main Chapter 4 describes our proposed architecture in detail. Results of the experimental evaluation are dedicated to Chapter 5 which demonstrates the real-world performance under various experimental setups and with different measurements executed. The end is marked by a conclusion Chapter 6 in which we reflect back at the work we did in the past year while working on this thesis, and propose areas and ideas for future work.

Chapter 2
Problem Statement and Background

The goal of this thesis is twofold - processing raw data from LiDAR unit(s) to produce 3D point cloud and utilize the raw sensor data to implement odometry. The raw 2D LiDAR scan that the LiDAR produces doesn't carry all the available information as it doesn't encompass the sensor's intrinsic parameters. Therefore, the 2D scan is fused with the LiDAR's intrinsics via a 2D to 3D transform to produce precise reconstruction of the sensor's surroundings in the format of cloud of points with XYZ coordinates - the Point Cloud. This process is described in Section 2.1. The point cloud is then used by other, higher level algorithms for; e.g., localization, planning, mapping, etc.

On the other hand, the raw depth data can also be used, on its own. Inspired by the localization method [1] we chose to explore this area by implementing proof of concept (PoC) odometry via classical image methods (e.g., the FAST feature detector) which later utilize the LiDAR's capability of providing 3D coordinates of a given feature to solve the odometry task. The description of the odometry task is in Section 2.2 including the review of the existing LiDAR based localization approaches in Section 2.2.1 and the description of the used visual feature based detection and description method Features from Accelerated Segment Test (FAST) (Section 2.2.2) and Binary Robust Independent Elementary Feature description (BRIEF) (Section 2.2.3) used to implement the odometry in this thesis.

2.1 LiDAR Depth Image to Point Cloud Transform

Raw LiDAR data comes in a form of 2D depth (or intensity, near-infrared, etc.) image that needs to be transformed into 3D Point Cloud via a projection parametrized by the LiDAR's intrinsic parameters. This poses a huge problem for power constrained robots running on batteries. But even without this constraint it's hard to process such amount of data in real time on a CPU, given it's non-parallel procedural nature of execution. This makes solving of the above described problem inefficient as the points could all be independently transformed at once as opposed to the linear "step by step" way of computing on a classical CPU.

FPGA and other highly parallel architectures, such as GPUs are thus more suitable for such task. We opted for the FPGA because it allowed us to tailor the hardware to match our needs exactly, keep the power consuption low and provide us with a complete design freedom.

The parallelizability of the projection comes mainly from the fact that the measurements (points) are independent and thus can be computed simultaneously. The projection from depth image into the 3D point cloud is done by utilizing the measured range at each LiDAR beam, the beam's location at the time of measurement, and known intrinsic parameters of the LiDAR, which describe the beam's physical placement and angle inside the LiDAR unit. With this knowledge, the transform is specified in the Ouster's datasheet [4] and the intrinsics are queryable via the LiDAR's TCP API. To calculate the measured point's XYZ coordinates, the necessary calculations are given:

$$\begin{aligned}
x &= (r-|\vec{n}|)\cos(\theta_{encoder}+\theta_{azimuth})\cos(\phi)+|\vec{n}|\cos(\theta_{encoder}),\\
y &= (r-|\vec{n}|)\sin(\theta_{encoder}+\theta_{azimuth})\cos(\phi)+|\vec{n}|\sin(\theta_{encoder}),\\
z &= (r-|\vec{n}|)\sin(\phi),
\end{aligned} \tag{1}$$

where r is the distance measured by the LiDAR's beam, $\vec{n}$ is the distance vector pointing from the LiDAR's coordinates' origin to the LIDAR's front optics, $\theta_{encoder}$ is the angle of rotation of the inner rotor, $\theta_{azimuth}$ is the angle of the beam source as physically angled and mounted to the LiDAR's rotor and ϕ is the pitch angle of the beam.

The developed architecture that computes the projection is described in Section 4.1.

2.2 Odometry

Odometry is the task of understanding robot's position change in time using various sensor data, such as camera(s), LiDAR(s), RaDAR(s) etc. [5] It's one of the fundamental tasks to be solved on robotic platforms to fulfill the need for ego-motion understanding to further utilize this knowledge in higher level algorithms, such as localization and mapping. There are many ways of approaching this task, depending on the sensors available and data in general, computing resources as some methods are more computationally demanding than others, power capacity (similar to the aforementioned constraint), precision required, and so on. The result of the odometry estimation is a 3D transform encompassing the rotation and translation between two time frames.

In this work we focus mainly on the odometry computed purely from LiDAR data and we explore a rather unconventional way of approaching it. We hope for it to be a proof of concept necessary for the field to grow in this direction and explore new possibilities which this approach opens up.

2.2.1 LiDAR Odometry

LiDAR odometry is odometry done from LiDAR data. This is usually but not necessarily approached by first projecting the raw sensor data into 3D point cloud upon which various odometry methods are applied. In general, we can categorize these approaches into **feature based**, **grid based** and **dense** [3].

2.2.1.1 Feature Based Methods

Feature based LiDAR odometry works by finding various features in the LiDAR scans and matching them together to yield odometry estimate. The features can be calculated using the raw data or more conventionally, the calculated 3D point cloud. The raw data is in a form of 2D depth image, similar to what; e.g., RGBD camera would produce so all the classical image algorithms can be utilized. Algorithms for calculating the features on the raw LiDAR data include detectors such as FAST [6], SURF [7], SIFT [8] and matched using descriptors such as BRIEF [9] in combination with their proximity in the depth image. [1] Point cloud features include edge features [10], ellipsoidal surfels [11] and ground features [12] and can be matched based on their proximity [13], type [10], or descriptor [11].

Our proposed solution fits into this category as we're using FAST feature detector to find and register features using the LiDAR depth image. We chose this approach because it's been successfully applied in the past by [1], to prove the concept and also because it's the perfect fit for the FPGA architecture, given it's low memory requirements and simplicity of the feature extraction pipeline [14]. Also, our pipeline serves the purpose of a proof of concept (PoC) and the individual elements, such as the FAST feature detector, can be readily exchanged for other feature detectors, such as the above mentioned point cloud feature detectors. In this work, we show that the pipeline is working and is highly modular and thus does not rely on any given algorithmic choice.

2.2.1.2 Grid Based Methods

Grid based registering methods build discrete grids estimating the probability of occupancy from the LiDAR point clouds and use Newton's optimization method to estimate the ego motion change in between consecutive scans [15]. The Newton's method searches for a root of a function f (i.e., find x_r such that $f(x_r) = 0$) by an iterative process $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, with some initial guess x_0 until

$f(x_i)$ approaches 0 (or maximum iterations is reached, signifying the algorithm failed to find the root). Because Newton's method uses division, it's not well suited for the FPGA architecture as divider circuit is both resource demanding and slow (with an exception to division by a factor of 2^n for $n \in \mathbb{Z}$). This can be overcome by using a look-up table (LUT) - a precomputed array of values for $\frac{1}{x}$ with precision depending on given requirements and memory available inside the FPGA. The value $\frac{y}{x}$ can then be calculated as $y \cdot \frac{1}{x}$ where $\frac{1}{x}$ will be a value from the precomputed LUT. The multiplication is usually accelerated by digital signal processors (DSPs) inside the FPGA (as described in Section 3.2). The multiplication can be precomputed as well, although the memory available in the FPGA is usually very constrained and might be insufficient.

2.2.1.3 Dense Methods

Dense methods' name is derived from the fact that they utilize the point cloud itself rather than the information extracted from it, thus working with *dense* cloud of points. Dense methods are usually based on a variation of Generalized Iterative Closest Point (GICP) algorithm [16], which iteratively finds the least-squares solution of fitting two point clouds onto each other. There are also non-iterative algorithms such as [17] that solve the point cloud matching in terms of least-squares using singular value decomposition (SVD) [18]. Dense methods are the most common but also computationally demanding and usualy, only a fraction of points from point clouds are considered [3].

We didn't choose this method because we didn't deem it a good fit for the FPGA architecture given its memory requirements and computational requirements. We chose to pursue performance, latency and low power usage and chose to implement efficient FAST feature detector from [19] inside the FPGA pipeline, track the features frame-to-frame and then use the SVD to find the best fit (in the sense of least-squares) transform to match the set of points as descibed in [17].

2.2.2 Features from Accelerated Segment Test

Features from Accelerated Segment Test (FAST) [6] is an image corner detection algorithm designed to be lightweight and fast, hence the name. It works by comparing the intensity difference between pixels lying on the Bresenham circle [20] with the radius of 3 and their center pixel (see Fig. 1). When this difference is bigger than the set threshold t, the pixels are counted as a 1 and when a large enough consecutive set of 1s is found, the center pixel is considered to be a corner.

The architecture implementing the FAST feature detector has been adapted from [19] and it is described in Section 4.2.2. We are also aware of the fact that the corners are error-prone for the LiDAR sensors as the laser beam can easily miss the corner and return a distance measurement of an object *behind* the corner. This can have negative impact on the odometry as the features, paired with the noise of the sensor can become unstable and inconsistent. This theoretical flaw is also deemed in the odometry results Section 5.2 to be one of the key sources of imprecision. As the architecture proposed in Chapter 4 is designed to not rely on a specific algorithmic choice, this can be easily fixed by different choice of a feature detector, which can be swapped with the FAST; e.g. the maximally stable extremal regions (MSER) [21] which would search for stable surfels.

2.2.3 Binary Robust Independent Elementary Feature description

The Binary Robust Independent Elementary Feature description (BRIEF) [9] algorithm is a feature description method which works by comparing pairs of pixels in a window surrounding a found feature and producing a descriptor vector, which can later be utilized to measure the similarity of found features on the scan-to-scan basis. This allows us to track their motion in time and calculate the sensor's ego-motion in time. The choice of pairs can impact the performance of the description and

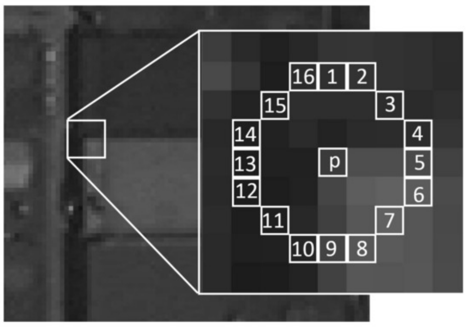


Figure 1: Pixels used by the FAST detector. [22]

is usually randomly generated, but can even be genetically optimized for a specific environment, as shown in [23].

2.2.4 Least Squares Fitting of Two 3D Point Sets

The task of odometry is to find a linear 3D transform expressing the rotation and translation of the sensor between two time frames. After we successfully match the features found by the feature detector like the one described in Section 2.2.2, we can proceed onto the odometry estimation. We chose to follow the work of [17] and implement their solution to this task. For that, we first project the depth image 2D features into 3D and then try to find the optimal (in the sense of least squares) rigid body transform between the two sets of points, from the previous and the last scans. We do this non-iteratively, as opposed to the traditionally used GICP as partially described in Section 2.2.1.3.

Let $\boldsymbol{A}, \boldsymbol{B} \subseteq \mathbb{R}^{3\times n}$ be matrices consisting of n 3D projected feature coordinates of the features matched in the previous and last LiDAR scan, respectively. Then, we assume that the only difference between these matrices is a rigid body transform with the addition of a noise produced by the sensor; i.e., it holds that $\boldsymbol{B} = \boldsymbol{RA} + \boldsymbol{t} + \boldsymbol{N}$, where $\boldsymbol{R}$ is the rotation matrix, $\boldsymbol{t}$ is the translation vector (note: we use matrix-vector addition as a shorthand for adding the vector to the matrix column wise; i.e., to each individual column. In other words, the addition can be expressed as $+ : \boldsymbol{A} \subseteq \mathbb{R}^{m\times n} \times \boldsymbol{t} \subseteq \mathbb{R}^{m} \to \boldsymbol{B} \subseteq \mathbb{R}^{m\times n}$ and $\boldsymbol{A} + \boldsymbol{t} \mapsto \boldsymbol{B}$ such that it holds that $\boldsymbol{B}_i = \boldsymbol{A}_i + \boldsymbol{t}$ for $i = 1, 2, \ldots, n$, where $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$ are the columns of matrix $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively.) and $\boldsymbol{N}$ is the noise matrix between the two frames. Our task is to find the best fit (in a sense of least-squares) matrix $\boldsymbol{R}$ and vector $\boldsymbol{t}$ to provide us with the rotation and translation part of the motion of the sensor between the two consecutive frames. This can be formulated as finding the least-squares solution to the task

$$\underset{\boldsymbol{R}\in\mathbb{R}^{3\times 3}, \boldsymbol{t}\in\mathbb{R}^3}{\operatorname{argmin}} \quad ||\boldsymbol{B} - (\boldsymbol{RA} + \boldsymbol{t})||^2. \tag{2}$$

It was shown in [24] that $\boldsymbol{R}'\boldsymbol{c} + \boldsymbol{t}' = \boldsymbol{c}'$ where

$$\begin{aligned} \boldsymbol{c} &= \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{A}_n, \\ \boldsymbol{c}' &= \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{B}_n, \end{aligned} \tag{3}$$

and $\boldsymbol{R}'$ and $\boldsymbol{t}'$ are the optimal solutions to eq. 2; i.e., the mean of the columns of $\boldsymbol{B}$ equals to the transformed mean of the columns of the matrix $\boldsymbol{A}$. Therefore, we can subtract the means from the points in matrix $\boldsymbol{A}$ and $\boldsymbol{B}$ and reformulate the task in eq. 2 as

$$\underset{\boldsymbol{R}\in\mathbb{R}^{3\times 3}}{\operatorname{argmin}} \|(\boldsymbol{B}-\boldsymbol{c'})-\boldsymbol{R}(\boldsymbol{A}-\boldsymbol{c})\|^2 \tag{4}$$

where we use a matrix-vector subtraction similarly to the matrix-vector addition described above. This divides finding the optimal $\boldsymbol{R}$ and $\boldsymbol{t}$ into two tasks. First we find the optimal $\boldsymbol{R'}$ using eq. 4 and then calculate the optimal $\boldsymbol{t'}$ as $\boldsymbol{t'} = \boldsymbol{c}' - \boldsymbol{Rc}$. As derived in [17], the optimal matrix $\boldsymbol{R'}$ can be found with the following steps:

Step 1: Calculate $\boldsymbol{H} = \boldsymbol{AB}^T$.

Step 2: Find the SVD decomposition of $\boldsymbol{H}$ so that $\boldsymbol{H} = \boldsymbol{U\Lambda V}^T$.

Step 3: Calculate $\boldsymbol{X}$ as $\boldsymbol{X} = \boldsymbol{VU}^T$.

Step 4: Calculate the determinant of $\boldsymbol{X}$ $det(\boldsymbol{X})$.

Step 5: If $det(\boldsymbol{X}) = 1$, then $\boldsymbol{R'} = \boldsymbol{X}$. Else if $det(\boldsymbol{X}) = -1$ and one of the singular values found by the SVD is zero, then $\boldsymbol{V'} = [\boldsymbol{v}_1 \boldsymbol{v}_2 - \boldsymbol{v}_3]$ where $\boldsymbol{v}_i$ for $i = 1,2,3$ are the column vectors of the matrix $\boldsymbol{V}$ and $\boldsymbol{R'} = \boldsymbol{V'U}^T$. Else the algorithm failed (see [17] for details).

After sucessfully finishing the above listed steps, we use the optimal rotation matrix $\boldsymbol{R'}$ to find the optimal translation vector as $\boldsymbol{t'} = \boldsymbol{c'} - \boldsymbol{R'c}$.

Chapter 3

Hardware & Software Setup

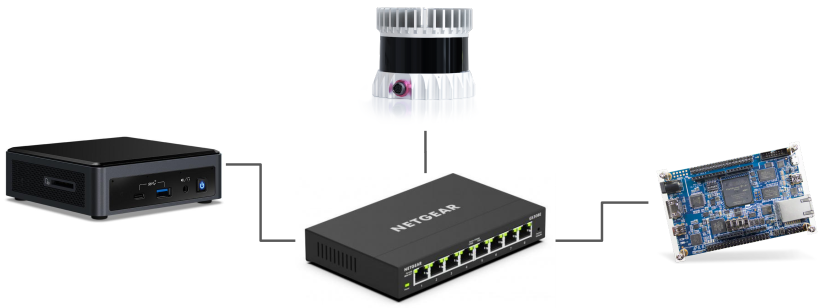


Figure 2: Schema of the switched network hosting the DE10-Nano board, Ouster OS0 LiDAR and Intel NUC. (Images from [25], [26], [27], [28].)

This Chapter describes the hardware and software used to develop the proposed system. Our hardware setup consists of the FPGA board DE10-Nano connected to the Ouster OS0-128 LiDAR, and debugging and programming computer Intel NUC connected using switched gigabit ethernet network as it is shown in Fig. 2. As the development of a custom FPGA processing cores is challenging and often also time consuming process, we have opted to prospect the possibility of rapid FPGA prototyping using the High Level Synthesis (HLS) toolchain. The HLS allows for the description of the architecture (respectively the dataflow) using the C++ programming language which is then translated into a functional FPGA design. The following sections list the properties of the Ouster OS0 LiDAR (Section 3.1) and the DE10-Nano development board (Section 3.2.1) together with the description of the High Level Synthesis principles (Section 3.3), and u-dma-buf DMA buffers (Section 3.4) that has been used in the development process of this thesis.

3.1 Ouster OS0-128

Light Detection And Ranging (LiDAR) sensor is one of the key sensors when it comes to robot's orientation in 3D space. It actively scans its surroundings using lasers and after projection outputs 3D coordinates of obstacles it detected. The Ouster OS0-128 LiDAR depicted in Fig. 3 actively fires near infrared laser beams around itself to scan its surroundings. It provides a large vertical field of view (FoV) of 90° - 45° up and down - and a range of approx. 50 meters. It provides up to 2,621,440 points per second in a format of 128 rows by 512, 1024 or 2048 columns at a rate of 10 Hz or 20 Hz. The standard deviation of the Ouster OS0's measurements is specified by the manufacturer to be ±3 cm for lambertian targets and ±10 cm for retroreflectors. In our testing, we used the maximum resolution available (128 by 2048) at 10 Hz, as the 20 Hz is only available for lower resolutions. This means that we need to process approx. 254Mbits of data per second [4]. This data comes in a raw form that can be treated as a 128 rows by 2048 columns image, which needs to be transformed according to intrinsic parameters of the LiDAR to produce the desired cloud of 3D points (Point Cloud). Because of the sheer amount of the data received, the computing requirements are not negligible and can impose a big challenge for battery powered robots. That's where our custom FPGA solution comes in to tackle

Figure 3: The used Ouster OS0 LiDAR. [29]

this challenge by utilizing the parallelizability of the operations to free up expensive CPU and power resources for other tasks such as odometry / SLAM, planning, movement control, detection from cameras, etc.

The Ouster LiDAR comes packed with an inertial measurement unit (IMU) which we didn't use in our tests as our robots usually have dedicated, more precise IMUs and it's data is utilized in higher level algorithms, running on the CPU. The IMU is capable of providing measurements of forces acting upon the device and describe them in terms of rotation and translation in 3D space.

The Ouster data comes in a form of UDP packets of 24,896 bytes in our case, containing 16 columns of 128 measurements. This means that to cover one revolution of the LiDAR, it takes 2048 / 16 = 128 of those packets. The packet contains information about range in millimeters (32 bit number, only 20 bits used) of the hit surface, signal photons intensity of the return signal, and near infrared (NIR) natural environmental illumination. We considered only the depth data in our work but others can be utilized for other similar tasks as well and their addition would follow similar procedure as shown in this thesis. The Ouster provides the necessary steps to project the 2D points into 3D Point Cloud and the necessary intrinsic parameters are available via TCP API from the LiDAR unit itself.

3.2 Field Programmable Gate Array

Field Programmable Gate Array (FPGA) [30] is a circuit that can be (re)configured to serve specific computing needs. Unlike traditionally used integrated circuits (ICs) like the CPU or GPU, the FPGA isn't constrained to any specific subset of functions it can provide and can be configured to provide specifically needed capabilities. This is done by the means of Hardware Description Language (HDL) such as Verilog or as in our case, VHDL. HDL is used to describe specific digital circuit made up from simple logic gates and memory elements (usually D-Flip Flops). This provides very wide range of potential use cases and allows us to highly accelerate the tasks at hand as opposed to general purpose processing units. FPGAs can even implement CPUs or GPUs, albeit it will be significantly less performant than using the in-sillicon implemented hardware. FPGA provides this reconfigurability by allowing the designer to configure so called Logic Blocks (LBs) or Logic Elements (LEs) and the interconnection in between these LBs. Logic Blocks can function as a simple logic function (e.g., XOR, NOR, ...) or a memory element (usually few bits in size). With the capability of expressing logic functions, memory elements and connecting them in fully configurable fashion, the FPGA is able to implement wide range of digital circuits. In addition to this, many FPGAs as well as the one

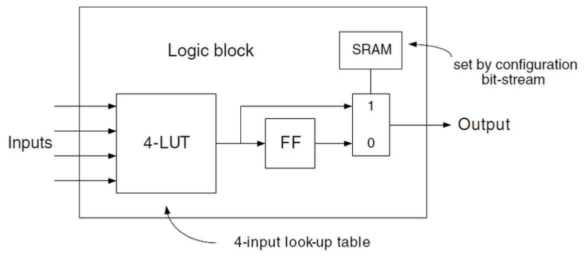


Figure 4: Simplified schema of the logic block. Four input LUT can be seen as well as 1 bit flip flop (FF) and SRAM controlled multiplexor at the output of the LB. [31]

we used (described in Section 3.2.1) hosts various digital signal processors (DSPs) to accelerate tasks such as integer multiplication and addition. We took advantage of those using fixed point arithmetic, as described in 4.1.2.1.

The logic blocks are usually made up from a look-up table (LUT) and a D-Flip Flop (DFF). The LUT is a small, usually SRAM memory capable of emulating any logical function of 4 to 8 binary inputs. It works by specifying values of the function for every possible input encoded as a memory address. For example, the 2-input AND function would have the value '1' written on the address of 0x3 as it's '11' in binary and the AND is '1' if both of the inputs are '1'. The logic block can also be configured to function as a memory element by utilizing the in-sillicon made D-Flip Flop. Simplified schema of the LB can be seen in Fig. 4.

We chose this technology for it's powerful ability to solve computing tasks in parallel, from which both of our tasks can benefit immensely. It also allowed us to keep the power consumption low and free up resources of the CPU, which was traditionally used to perform the described tasks.

3.2.1 DE10-Nano Development Board

For our implementation we chose the Terasic DE10-Nano Development Board [32] depicted in Fig. 5 for its small power footprint and reasonable cost. In spite of the small cost, the Cyclone V 5CSEBA6U23I7NDK [33] chip features sufficiently large number of 41 910 adaptive logic modules (ALMs) - the basic element of the Cyclone's FPGA fabric equaling to roughly 110K logic blocks - which allowed us to design the custom hardware without any performance hurting compromises in mind. The Cyclone V also hosts 553 M10K memory blocks, each 10 kB in size. There are also 112 DSPs accelerating integer multiplications. The FPGA, running at 50 MHz is accompanied by two ARM A9 cores running at 925 MHz, 1 GB of DDR3 SDRAM memory, and 64 kB of low latency on-chip memory. The chip is semantically divided into FPGA and Hard Processor System (HPS - Arm cores) area. The FPGA is connected to GPIO ports, LEDs, HDMI, ADC, and buttons. The HPS is connected to all the other peripherals, including the DDR3 memory and an ethernet port. This design constrained us to utilize the Linux running HPS for all of the necessary communications via the ethernet port and leave to the FPGA only the sole computations.

All the HPS-FPGA communication is done via DMA thanks to *bridges*. The Cyclone V features three distinct bridges to utilize HPS-FPGA communication - HPS-to-FPGA bridge, its lightweight variant and FPGA-to-HPS brigde. The reason for having two bridges in one direction (HPS-to-FPGA and Lightweight HPS-to-FPGA) is to distinguish between full-fledged high-bandwith DMA transfers and low-bandwith Control & Status Register (CSR) reads and writes. CSRs are special purpose registers exposed by various hardware (be it 'real' HW such as network cards or GPUs or 'soft' hardware, such as ours implemented on the FPGA) to provide means of communications between software and hardware. CSRs are used to control the hardware (e.g. send commands or pointer to memory area) and read it's state (e.g. various errors, busy & free flags, etc.). This kind of communication is extremely

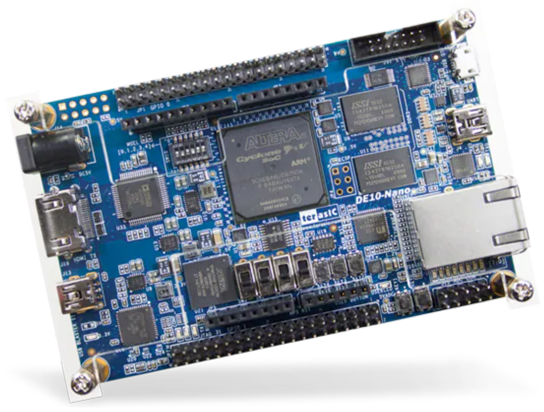


Figure 5: The used DE10-Nano devlopment board. [28]

common in hardware that the Cyclone V has a specific low-latency low-bandwith (up to 32 bits as opposed to 1024 bits in the normal HPS-to-FPGA) bridge to provide means for this kind of communication. We use only the two above mentioned HPS-to-FPGA bridges in our architecture as the data transfers from FPGA to HPS are done via DMA directly connected to the external DDR3 SDRAM. More detailed info can be found in [34].

3.3 High Level Synthesis

High Level Synthesis (HLS) is a way of converting code written in high level programming language into Hardware Description Language (HDL). It allows designers to be freed of low-level details and hardware considerations when tackling a problem and focus solely on its efficient algorithmic solving. HLS toolchain then converts the high-level design, usually written in C(++) to Register Transfer Level (RTL) [35] design expressed in HDL. One of the main benefits of HLS, apart from its huge development time savings and scalability and maintainability is the ease with which the developed components can be unit tested by utilizing pure C(++) and thus all the software methods normally employed in SW development. It can be used to determine accuracy, latency, the ability to efficiently pipeline the operations and overall correctness of the design. The tests can be run in pure software emulation, which is many times faster (similar to classical software speeds) than the HDL synthesis and RTL simulation inside a tool such as ModelSim. This boosts the development time and provides advanced debugging and profiling abilites. Only after all the necessary tests are done and constraints fulfilled, the RTL simulation is run to make sure everything will work after implemented inside the FPGA fabric.

Because we used the Intel's Cyclone V chip, we were bound to the Intel's software / hardware development toolchain, including the Intel HLS [36]. This also determined our HLS language of choice - C++. We chose the HLS for implementing the 2D to 3D transformation as it allowed us to write very readable and scalable code in C++, which clearly computed the transformation by the steps described in the official Ouster OS0 documentation [4] and stay clear of low-level details like timing closures and pipelining while doing so. It also allowed us to use preimplemented Intel libraries for sine and cosine calculations which saved us a considerable amount of development time. It does all of this by making every designated C++ function an independent HDL component, which can then be "plugged" into an existing HDL design by; e.g., VHDL's structural modeling.

3.3.1 Intel HLS

The desired component to be compiled into HDL is written in C++ as an ordinary function, marked with the "component" keyword in front of the declaration. The parameters to this function are translated into input ports on the resulting component. Basic types (or structures consisting of basic types, such as integers or floats) are translated as an input of the necessary width. Pointers or arrays are translated into Avalon memory-mapped interface. The return value is translated similarly to the input parameters.

The Intel also provides basic HLS libraries, such as definitions of datatypes for fixed-point arithmetic, basic linear algebra and trigonometric functions. We used the datatype "ac_fixed" extensively and all the necessary trigonometric functions (as described in Section 2.1) were implemented in the HLS libraries using fixed point arithmetic (see Section 4.1.2.1 for details). Another advantage to using Intel HLS is the automated report generation that provides readable, HTML formatted information about the pipeline order, maximum frequency rating, logic block usage, DSP usage as well as line by line analysis of the resource usage.

One downside of the Intel's HLS toolchain is the fact that it hasn't got robust linear algebra libraries (apart from a simple matrix multiplication and Cholesky's decomposition), which posed a challenge on us as we relied on the SVD in our odometry. We were thus limited to computing the SVD inside of the HPS as implementing it in the HLS or the VHDL from scratch is well outside of the scope of this thesis and is unnecessary for the proof of concept. The details of the developed architecture are further described in Section 4.1.

3.4 u-dma-buf

U-dma-buf is a Linux kernel module designed to allocate physically contiguous memory chunks intended to be used as a direct memory access (DMA) memory buffers [37]. These buffers can then be used from Linux user space as a regular files and their physical address can be retrieved and used by the FPGA to access it. The module also ensures that the buffer is contiguous in physical address space and not only in virtual, otherwise the memory would be unusable for the FPGA as it uses physical memory addressing directly. Another necessary funcionality of this module is the ability to disable the CPU cache and thus keep the memory consistent between the HPS and FPGA without the necessity of flushing the cache periodically. We used the u-dma-buf allocated buffers to store the incoming LiDAR data and the FPGA outputs (point cloud and feature coordinates). The details of the usage follows in Chapter 4.

Chapter 4
Architecture

This section introduces our custom designed architecture meant both as a production ready and proof of concept design. We utilized both the HPS and FPGA parts to build a system on programmable chip (SoPC). We divided the processing pipeline into part that makes sense to be implemented in the FPGA and part that should be processed in the HPS.

The architecture's overview can be seen in Fig. 6. It comprises of two parallel pipelines - point cloud generation and odometry estimation. The leftmost block represents the connected LiDAR unit (described in Section 3.1) which sends its data over UDP to the HPS (described in Section 3.2.1) which then parses it into the DMA buffers for point cloud generation, described in Section 4.1.1 and preprocesses it for the FAST feature detector using process described in Section 4.2.1. Then the data is transferred using a DMA into the FPGA pipeline, marked in the schematic as a yellow box, described in Section 4.1.2. The two units - "lidar2pntcloud" generating point cloud, described in Section 4.1.2.1, and FAST feature detector, described in Section 4.2.2 - are running in parallel and transferring the results via DMA back to the HPS. The resulting point cloud is then visualized using Intel NUC 10i7FNK and the features found by the feature detector are used in the rest of the odometry pipeline. The HPS implemented feature description, matching and registration are described in Sections 4.2.3, 4.2.4, and 4.2.5, respectively.

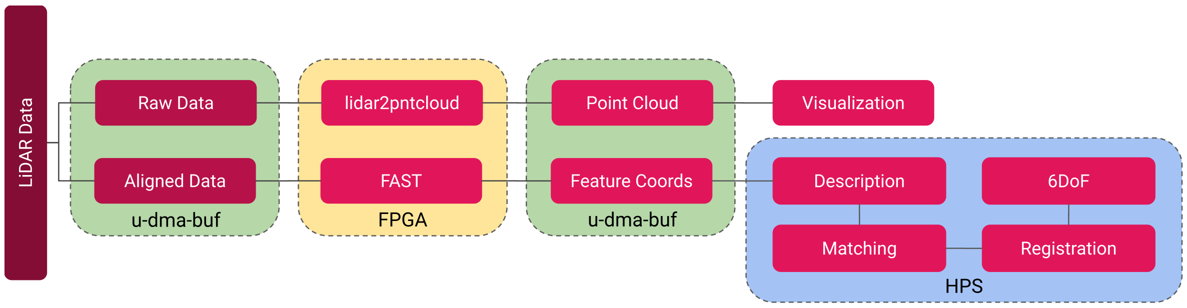


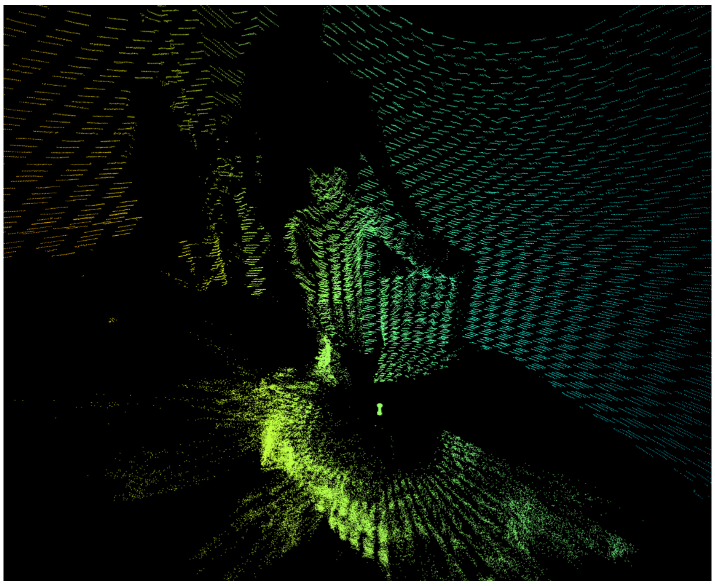
Figure 6: Overview of the architecture of the proposed pipeline.

4.1 Point Cloud Generator

This section describes the architecture of the point cloud generation pipeline in greater detail. The pipeline is divided into a HPS and FPGA part. The HPS implements the LiDAR communication itself as well as orchestration of the DMA transfers. The FPGA then hosts a "lidar2pntcloud" component developed using the HLS implementing the projection described in Section 2.1.

4.1.1 HPS Part

The HPS part consists of a software running on the CPU that would be too complex or too expensive; e.g., in terms of development time or resources used to be deployed in the FPGA fabric. Also, because the ethernet port is connected directly to the HPS, it's necessary for receiveing the LiDAR data.


(a) Point cloud depicting a person standing near the LiDAR unit and leaning on a chair.

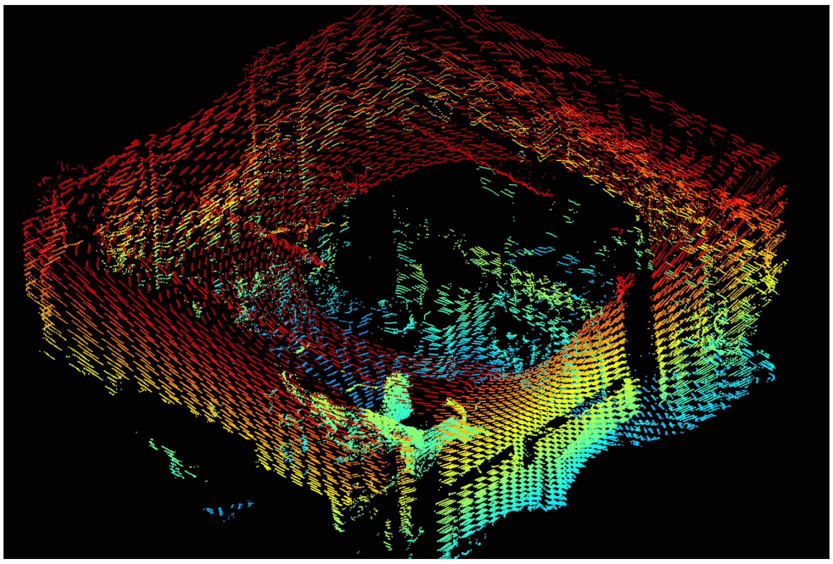

(b) Point cloud depicting our laboratory as seen from above.

Figure 7: Results of the proposed point cloud generating pipeline visualized by custom visualization software.

First, four u-dma-buf buffers are instantiated - one 1 MB for the raw LiDAR data, second 4 MB for the resulting Point Cloud, third 1 MB for the LiDAR's geometry induced skew compensated raw data and fourth 4 MB for the odometry output. After that, a Python script runs to query the LiDAR's intrinsic parameters. It parses the response and converts all the angles from floating point degrees to a custom fixed point integer representation. The fixed point representation works by rescaling the degrees from a range of 0 – 360 to 0 – 128 and multiplying it by 4096, before casting them to integers. This provides sufficient accuracy while keeping the number width at 20 bits. After parsing the intrinsics it stores them in a binary file to later be used by the main Orchestrator script written in C.

Then the main Orchestrator script can be run. It memory maps all the CSRs exposed by the

FPGA into its virtual memory space, as well as the on-chip RAM and the u-dma-buf buffers for array-like access. It also resets the SDRAM controller and initializes it for the FPGA to be able to use it. It then retrieves the DMA buffers' physical addresses available as a text file at "/sys/class/u-dma-buf/<*buffer_name*>/phys_addr", where *"buffer_name"* is a name chosen for the buffer at an instantiation time. It writes this address to a CSRs of mSGDMA components, used for managing the DMA transfers inside the FPGA. The mSGMDA IP developed and maintained by Intel provides scalable and standardized way of implementing a DMA. It provides many useful capabilities, such as read & write buffering, scatter & gather modus operandi, burst mode, streaming or memory-mapped transfers, aligned / un-aligned memory accesses, error readings, packet mode, and FIFO-like execution of multiple DMA requests. The main motivation behind using the mSGDMA IP was to stay abstracted from the low-level details of the SDRAM controller and the fact that it uses the standard Avalon interface, integrating seamlessly into our design which already uses Avalon, the standard bus developed by Intel specifically for use in their FPGAs. The mSGDMA instances in our SoPC design and their respective Avalon interfaces are visible in Fig. 9.

After the u-dma-buf addresses are written in the mSGDMA CSRs, the Orchestrator transfers the intrinsic parameters onto the FPGA's on-chip memory for use by the "lidar2pntcloud" component. The Orchestrator also uses the intrinsics for skew correction of the raw LiDAR data, necessary for the odometry pipeline (further detailed in Section 4.2.1). After that, it starts an infinite loop of listening to LiDAR data sent over the UDP on port 7501, parsing them in real time and starting the DMA transfers of the data into the FPGA fabric. It's also capable of sending the resulting point cloud over the UPD to the Intel NUC for visualization using our custom Python visualizer.

4.1.2 FPGA Part

The FPGA hosts the main component "lidar2pntcloud" for calculating the raw data projections, developed using the HLS toolchain. The FPGA system implements a simple finite state machine (FSM) with two states: Reading the intrinsics and processing the incoming LiDAR data. Upon initial configuration of the FPGA fabric by the bootloader, the FSM starts in the intrinsics state and waits for a signal from the HPS notifying it about the intrinsics being written in the on-chip RAM. After that, it reads the intrinsics into separate arrays for faster and parallel access, dividng the intrinsic parameters into azimuth and altitute angles, both of which need to be accessed in parallel by the "lidar2pntcloud" component. After finishing reading the intrinsics, it writes a specific sequence (0x101010) into the on-chip RAM at the address of 0x0, which is used by the HPS as a signal that it can start receiving the LiDAR data itself. The FSM also changes its state to LiDAR data processing and waits until the DMA transfer of the LiDAR data is started.

After the HPS parses the LiDAR UDP packets, it starts the DMA transfers of those data into the FPGA and the "lidar2pntcloud" component immediately starts processing it and, with deterministic pipeline delay of 61 cycles, as measured by the GHDL simulation (presented in Fig. 8), outputting the projected data via DMA back to the HPS. The component processes the LiDAR data in a batch manner of 32 size each, after which it pauses the pipeline for 22 cycles to buffer in another batch of the necessary intrinsics that it needs for computing the transform. The HLS generated component exposes its Input / Output (IO) as a standard Intel Avalon bus interface which is almost seamlessly incorporated into the system, which uses the Intel's mSGDMA IP Core as a DMA processor, that also uses Avalon to communicate, as seen in Fig. 9.

After each point is calculated it's immediately transferred via DMA into the SDRAM and available to the HPS. The result of processing a single LiDAR scan can be seen in Fig. 7a and Fig. 7b.

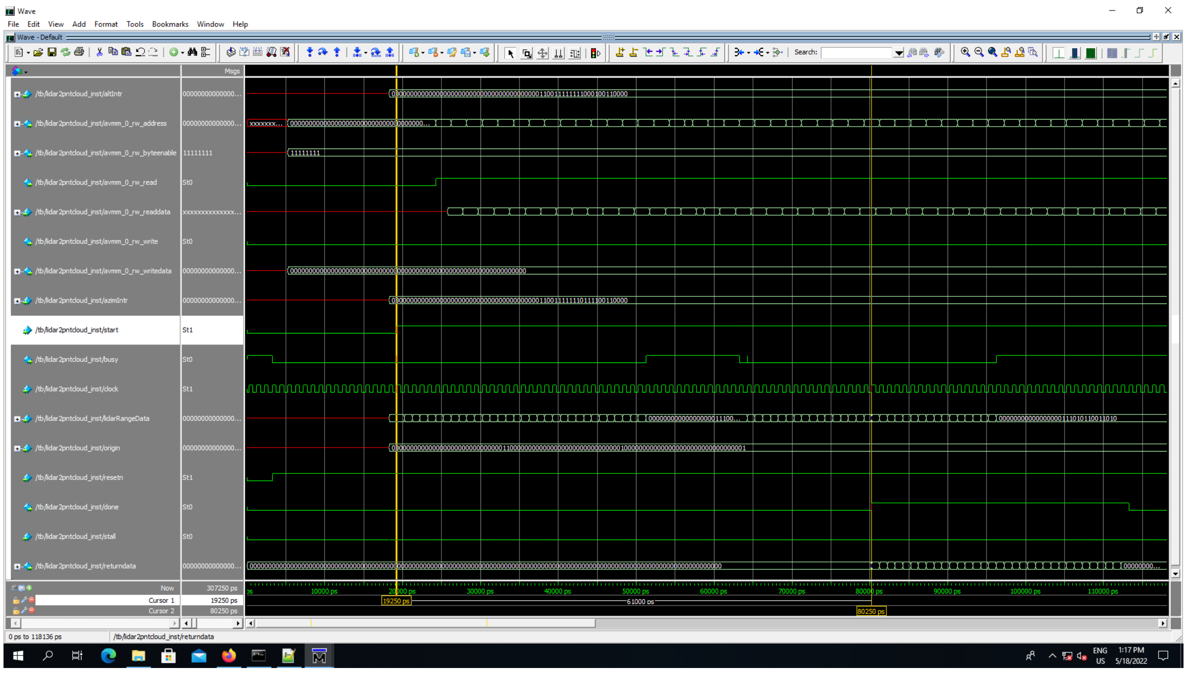

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/tb/lidar2pntcloud_inst/altIntr
/tb/lidar2pntcloud_inst/avmm_0_rw_address
/tb/lidar2pntcloud_inst/avmm_0_rw_byteenable
/tb/lidar2pntcloud_inst/avmm_0_rw_read
/tb/lidar2pntcloud_inst/avmm_0_rw_readdata
/tb/lidar2pntcloud_inst/avmm_0_rw_write
/tb/lidar2pntcloud_inst/avmm_0_rw_writedata
/tb/lidar2pntcloud_inst/azimIntr
/tb/lidar2pntcloud_inst/start
/tb/lidar2pntcloud_inst/busy
/tb/lidar2pntcloud_inst/clock
/tb/lidar2pntcloud_inst/lidarRangeData
/tb/lidar2pntcloud_inst/origin
/tb/lidar2pntcloud_inst/resetn
/tb/lidar2pntcloud_inst/done
/tb/lidar2pntcloud_inst/stall
/tb/lidar2pntcloud_inst/returndata
Now 307250 ps
Cursor 1 19250 ps
Cursor 2 80250 ps
61000 ps
0 ps to 118136 ps

Figure 8: Screenshot of the ModelSim simulation of the lidar2pntcloud component, showing the pipeline propagation delay using two cursors with space of 61 000 ps, equating to 61 cycles at a simulation frequency of 1 GHz.

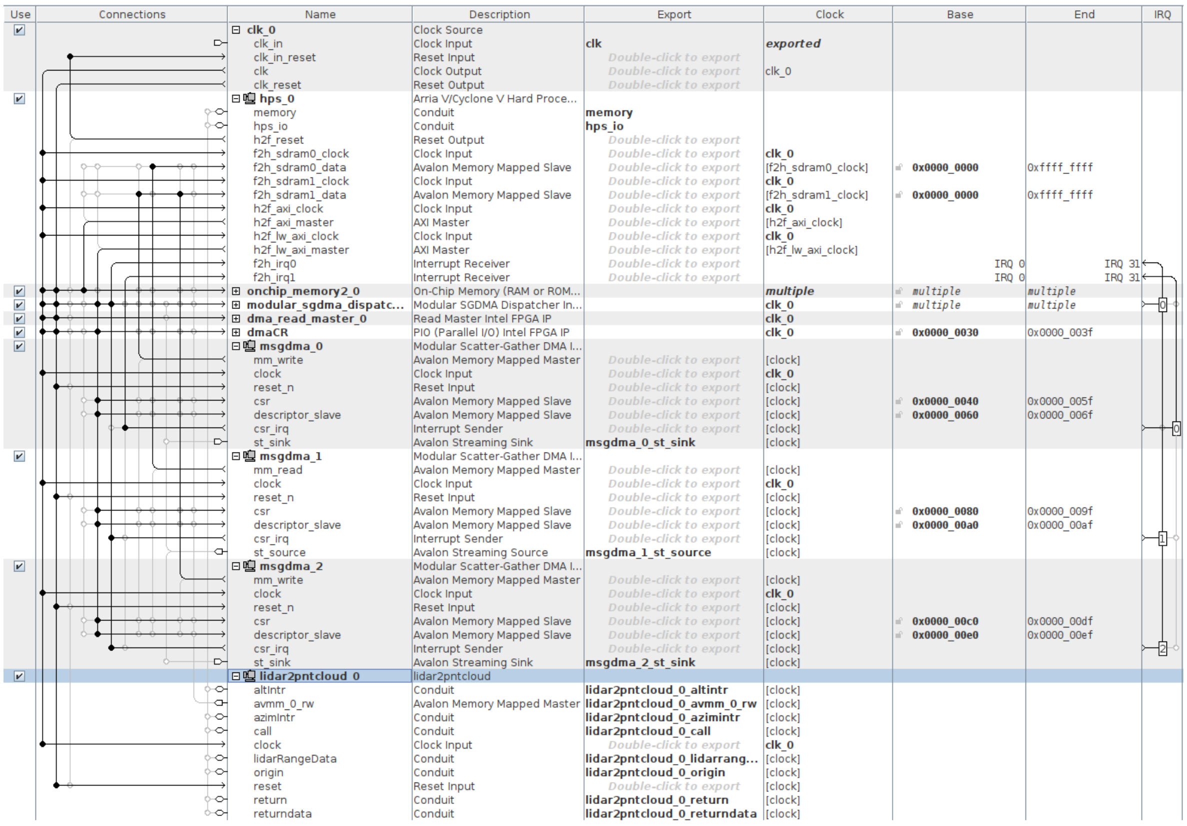
4.1.2.1 lidar2pntcloud

In this section, we present the HLS created, point cloud processing component "lidar2pntcloud" in greater detail. To implement the equations as stated in Section 2.1, we opted to use fixed-point arithmetic, as the floating point arithmetic requires hardware support from the FPGA fabric, otherwise it is very demanding in terms of logic block usage and propagation delays through the generated circuits. Fixed point arithmetic allowed us to use integer multiplication accelerating DSPs of the Cyclone V (see Section 3.2 for details) as well as simple full adders instantiated from the logic blocks. The number of bits to respresent the fixed point numbers and its integer / fractional part ratio was determined by extensive unit testing of the developed component and examining its error rate. Most of the fixed point numbers were 32 bits wide with 20 bits of integer part. The result is resource utilization / speed tradeoff rated at ± 2 mm error on approx. 1 km distance, which our LiDAR (described in Section 3.1) will never produce but which is the theoretical maximum distance expressed in the Ouster's UDP packets (see Section 3.1 for details). This allows for swapping the LiDAR unit for a different one without having to recompile the FPGA configuration files.

We also implemented an optional parameter to the component allowing the origin of the LiDAR coordinate frame to be translated in space, allowing to use the same component for multiple LiDAR units and only specifying their relative offset to the component in order to produce a correctly merged point cloud.

4.2 LiDAR Odometry

This subsection introduces our custom designed odometry pipeline, running in parallel to our Point Cloud generating pipeline. With this architecture, we show modular, highly modifiable odometry pipeline which can be easily modified to suite various odometry needs and constraints (e.g., different feature detector, different data pre/post-processing, etc.). We thus propose a general-purpose odometry



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✓		⊟ **clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	*Double-click to export*				
		clk	Clock Output	*Double-click to export*	clk_0			
		clk_reset	Reset Output	*Double-click to export*				
✓		⊟ **hps_0**	Arria V/Cyclone V Hard Proce...					
		memory	Conduit	**memory**				
		hps_io	Conduit	**hps_io**				
		h2f_reset	Reset Output	*Double-click to export*				
		f2h_sdram0_clock	Clock Input	*Double-click to export*	**clk_0**			
		f2h_sdram0_data	Avalon Memory Mapped Slave	*Double-click to export*	[f2h_sdram0_clock]	**0x0000_0000**	0xffff_ffff	
		f2h_sdram1_clock	Clock Input	*Double-click to export*	**clk_0**			
		f2h_sdram1_data	Avalon Memory Mapped Slave	*Double-click to export*	[f2h_sdram1_clock]	**0x0000_0000**	0xffff_ffff	
		h2f_axi_clock	Clock Input	*Double-click to export*	**clk_0**			
		h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_clock]			
		h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**			
		h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi_clock]			
		f2h_irq0	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
		f2h_irq1	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
✓		⊞ **onchip_memory2_0**	On-Chip Memory (RAM or ROM...		***multiple***	*multiple*	*multiple*	
✓		⊞ **modular_sgdma_dispatc...**	Modular SGDMA Dispatcher In...		**clk_0**	*multiple*	*multiple*	0
✓		⊞ **dma_read_master_0**	Read Master Intel FPGA IP		**clk_0**			
✓		⊞ **dmaCR**	PIO (Parallel I/O) Intel FPGA IP		**clk_0**	**0x0000_0030**	0x0000_003f	
✓		⊟ **msgdma_0**	Modular Scatter-Gather DMA I...					
		mm_write	Avalon Memory Mapped Master	*Double-click to export*	[clock]			
		clock	Clock Input	*Double-click to export*	**clk_0**			
		reset_n	Reset Input	*Double-click to export*	[clock]			
		csr	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_0040**	0x0000_005f	
		descriptor_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_0060**	0x0000_006f	
		csr_irq	Interrupt Sender	*Double-click to export*	[clock]			0
		st_sink	Avalon Streaming Sink	**msgdma_0_st_sink**	[clock]			
✓		⊟ **msgdma_1**	Modular Scatter-Gather DMA I...					
		mm_read	Avalon Memory Mapped Master	*Double-click to export*	[clock]			
		clock	Clock Input	*Double-click to export*	**clk_0**			
		reset_n	Reset Input	*Double-click to export*	[clock]			
		csr	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_0080**	0x0000_009f	
		descriptor_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_00a0**	0x0000_00af	
		csr_irq	Interrupt Sender	*Double-click to export*	[clock]			1
		st_source	Avalon Streaming Source	**msgdma_1_st_source**	[clock]			
✓		⊟ **msgdma_2**	Modular Scatter-Gather DMA I...					
		mm_write	Avalon Memory Mapped Master	*Double-click to export*	[clock]			
		clock	Clock Input	*Double-click to export*	**clk_0**			
		reset_n	Reset Input	*Double-click to export*	[clock]			
		csr	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_00c0**	0x0000_00df	
		descriptor_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_00e0**	0x0000_00ef	
		csr_irq	Interrupt Sender	*Double-click to export*	[clock]			2
		st_sink	Avalon Streaming Sink	**msgdma_2_st_sink**	[clock]			
✓		⊟ **lidar2pntcloud_0**	lidar2pntcloud					
		altIntr	Conduit	**lidar2pntcloud_0_altintr**	[clock]			
		avmm_0_rw	Avalon Memory Mapped Master	**lidar2pntcloud_0_avmm_0_rw**	[clock]			
		azimIntr	Conduit	**lidar2pntcloud_0_azimintr**	[clock]			
		call	Conduit	**lidar2pntcloud_0_call**	[clock]			
		clock	Clock Input	*Double-click to export*	**clk_0**			
		lidarRangeData	Conduit	**lidar2pntcloud_0_lidarrang...**	[clock]			
		origin	Conduit	**lidar2pntcloud_0_origin**	[clock]			
		reset	Reset Input	*Double-click to export*	[clock]			
		return	Conduit	**lidar2pntcloud_0_return**	[clock]			
		returndata	Conduit	**lidar2pntcloud_0_returndata**	[clock]			

Figure 9: Screenshot of the Quartus Platform Designer showing the SoPC schema interconnected using the Avalon bus. The Avalon interface generated by the HLS can be seen on the lidar2pntcloud component (bottom).

pipeline capable of meeting the power-constrained robotic needs. We also show unconventional way of estimating LiDAR odometry by utilizing the raw depth images, instead of the projected point clouds.

4.2.1 Data Preprocessing

Even though we aim to utilize only the raw, unprojected 2D LiDAR data scans, we still needed to compensate for the Ouster's induced scan rows skew. Because of the LiDAR's internal structure, the columns arriving in each data packet aren't consecutive and has to be shifted by utilizing the OS0's intrinsic parameters, otherwise all the vertical lines become spread out and unrecognizable for any feature detector utilizing the data later in the pipeline. Interestingly, compensation of the Lidar intrinsics as seen in Fig. 10 still results in artifacts, especially visible on close vertical edges, that represent a hardware limitation of the given LiDAR type. Note that the offical Ouster driver provides a similar result.

Nevertheless the preprocessing can be approached in multiple ways and we propose two of those and implementing and discussing only one as a proof of concept. The most simple way is to pre-compute the shift table for each row based on the LiDAR's intrinsic parameters and use those as the UDP data packets are parsed to have the data corrected by the time they are written into the u-dma-buf buffer. The other, more involved but more efficient way would be to utilize more FPGA on-chip memory FIFOs to buffer 132 columns of the image (as the skew ranges from -66 to +66) and have the incoming data written into the correct place during the DMA reading phase. We opted for the first option, as the second is deemed as a mere optimization of the first variant and isn't really providing much of a value to our proposed proof of concept pipeline.

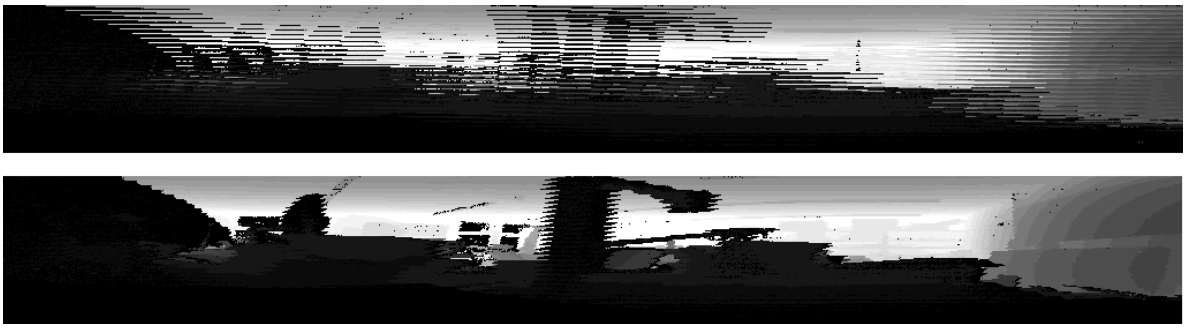


Figure 10: Depth image of our lab before (top) vs. after (bottom) data skew correction. The vertical edge inconsistency on a close object is still visible.

4.2.2 FAST feature detector

After the data is corrected and transferred onto the FPGA fabric, FAST computing component based on the implementation [19] immediately starts to buffer 7 columns of the depth image to produce feature detections. Once a feature is located, its location is immediately transferred via the DMA back to the Linux-running HPS, where it's later utilized for the rest of the odometry pipeline. The feature detector features quick eight steps detection pipeline ended with a non-maxima suppresion module to filter out false-positive feature clusters. The simplicity of the FAST algorithm allows us to detect features in real-time without introducing any noticeable lag or pipeline stalls. It's also very lightweight on the FPGA resource usage and allows a fair bit of parallelization, which makes it perfect fit for the FPGA architecture.

We had to change the original design described in [19] in a few ways to fit our needs. First by changing the expected image width and height, then exchange the original Cyclone IV FIFO memory elements to use the Cyclone V provided ones and also change the image data from 8 bits to 16 bits as the depth measured by the LiDAR ranges from 0.3 m up to approximately 50 m and is expressed in millimeters as specified by [4]. This also meant changing the dynamically chosen detection threshold to 16 bits and increase the steps in which it changes the threshold to increase the convergence speed. The adaptive thresholding is used to keep the number of detected features between 128 and 256 to keep the pipeline running in real-time as bigger number of features would result in longer computation times of the rest of the odometry pipeline.

4.2.3 Binary Robust Independent Elementary Feature description

Once we have the raw LiDAR image paired with coordinates of FAST-found features, we can proceed to the feature description step. We chose the BRIEF description algorithm, descibed in Section 2.2.3. We generate the compared pair coordinates randomly at the beginning of our program based on the window size parameters and the number of pairs required. We then iteratively calculate the descriptor vector for each of the found features. This algorithm could greatly benefit from FPGA implementation, but as a proof of concept it's still reasonably fast even when running on the CPU(s). Also, with the regions usually being rather large (even 128x128 or more), the memory requirements are often too big for the usually small memory available within the FPGA fabric, thus leaving us no choice when it comes to the HPS vs. FPGA implementation.

4.2.4 Feature matching & tracking

Once we identified and described all the features in the current & previous LiDAR scan, we can move onto the feature matching and tracking. Feature descriptors are compared and the number of differences is counted. The comparison is done using eXclusive-OR (XOR) [38] operation, which yields a binary vector with 1s in places where the two vector differed. The number of 1s is summed and is considered as the distance between the two descriptors (it can actually be viewed as the L2-squared distance between the descriptors). If this distance is small enough, the features are considered to be the same and matched together, with an exception to features which are too far away from each other in the depth image. The maximum distance allowed is controlled by a parameter which we evaluated at different values in the Results Chapter 5. This process is done for each pair of features with an algorithmic complexity of $\mathcal{O}(n^2 \cdot k)$, where n is the number of features and k is the length of the descriptors. Such process could benefit from architectures such as FPGA because of it's ability to compute things in parallel and the efficiency of operations such as XOR and counting the number of 1s in the difference vector. On the other hand, in our case we would have to implement another two DMAs to transfer the descriptors onto the FPGA fabric and then the results back to the HPS, given the previous step of feature description would remain implemented in HPS for its high memory requirements. These transfers would then impose additional static overhead. Even with that in mind, it might be worth pursuing this implementation in a future work as it has been shown to be the single slowest part of the entire pipeline, as further described in the Results Section 5.1.5.

4.2.5 Feature Registration

Once we identify feature matches in the subsequent LiDAR scans, we can proceed to the registration phase in which we try to estimate the 6 degrees of freedom (DoF) pose change based on the change in positions of the tracked features. To accomplish this task we first project the features from the raw 2D scans into their 3D coordinates (we use the already projected point cloud to find the coordinates of those features) and then we use the two sets of points from the previous and the last scans to estimate the ego-motion. For this we chose to implement the [17] described in Section 2.2.4.

Once we have the 3D coordinates of the matched features, we can follow the steps described in Section 2.2.4 to estimate the 3D rotation and translation of the sensor between the two consecutive scans. To implement these steps we used the C++ Eigen library [39] to calculate all the necessary matrix multiplications, SVD and matrix determinant.

Chapter 5
Results

In this chapter we show the results of our experiments in which we showcase the functionality of our proposed solution and in the case of point cloud generation a direct comparison with its prior CPU implementation in terms of the data throughput and power consumption. Two sets of experiments were done for this purpose: Latency measurements of various parts of the pipeline as well as benchmarking the reference CPU solution, and benchmarking the odometry pipeline by deploying our solution onto a robotic platform and compare the absolute and relative errors of our estimates to a ground truth. Last but not least, we evaluate the FPGA resources utilized for the individual blocks of the developed architecture.

5.1 Latency Experiments

In this section, we present experimental results of the real-world performance of our architecture by measuring the time it takes to finish various parts of the pipeline and thus giving us a precise idea about its performance. We follow the architecture as illustrated in Fig. 6 and start by presenting the time requirements of receiving the LiDAR data itself, then we move onto measuring the latency of feature detection and point cloud generation after which comes a section about all the odometry blocks running on the HPS - namely the description, matching and registration by the SVD described in Section 4.2.5. All of the results are based on at least 70 measurements unless stated otherwise. Latencies measured on the HPS were done using C++ steady_clock class and were measured with a microsecond precision. All of the statistics from the measured times are presented in Table 1 using various BRIEF window width & height and descriptor length. Boxplots are also presented in Fig. 11a, 11b, 12, 13, and 11c. The detailed description and discussion of the performed experiments follow.

5.1.1 Receiveing the LiDAR Data

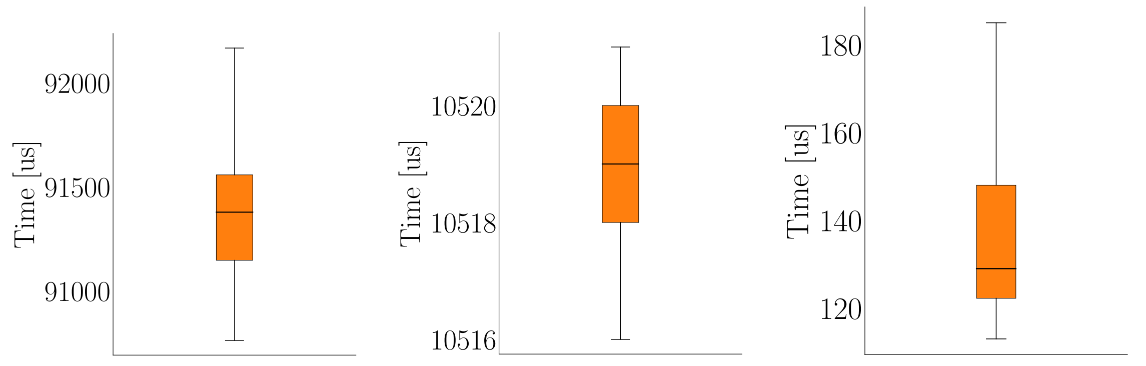
In this subsection we present the timing requirements for receiveing the LiDAR data over the UDP and writing it into the u-dma-buf buffers. The results are based on 80 measurements and are presented in Table 1 and in a boxplot graph in Fig. 11a. Because the LiDAR was running at 10 Hz, we expect the time to receive one complete LiDAR scan to have a mean near tenth of a second, or 100 000 µs, which is almost exactly what we've measured. As we will see, our pipeline is bottlenecked by the HPS and thus runs at a slightly lower rate than 10 Hz depending on various hyperparameters. Because the rate of execution is lower than the LiDAR scan rate, we find the receiveing UDP buffers in Linux to be already filled with the data as we call C recv() function to retreive them, thus resulting in a slightly lower time than expected.

5.1.2 Point Cloud Generation in FPGA

In this subsection, perhaps the most interesting results are presented, describing the time required for the FPGA to produce a point cloud from a single LiDAR scan. The time is measured from the time the HPS starts the DMA transfer, after having prepared the raw data in the buffers, and up to the point where the FPGA finishes the transfer of the resulting XYZ coodrinates. The design produced by our high-level design, written in C++ using HLS (see Section 3.3) and described in Section 5.1.2, after compilation using the Intel's i++ compiler results in a pipeline which computes the points in bursts of 32 points, after which it pauses to buffer intrinsic parameters to later utilize them to compute next points. From ghdl simulations, we measured the pause be 22 cycles long on Cyclone V. Because of this, we should see latency of 2048×128 points being slowed by the 22 cycle buffering pauses, thus yielding approximately $2048 \times 128 \times ([32+22]/32) = 442368$ cycles. We confirm this estimate

Table 1: Latency statistics

Length	Width	Height	*Mean* [ms]	*Median* [ms]	*Std.Dev.* [ms]	*Min* [ms]	*Max* [ms]
			Data receive times				
–	–	–	91.50	91.38	0.88	90.76	98.49
			DMA Start to finish times				
–	–	–	10.54	10.52	0.07	10.52	10.89
			Description times				
16	64	32	1.00	0.93	0.21	0.82	1.61
32	64	32	1.87	1.62	0.47	1.34	3.01
64	64	32	3.12	3.03	0.42	2.60	5.59
128	64	32	6.32	5.87	1.48	4.54	10.92
256	64	32	11.30	10.27	2.75	8.94	20.44
512	64	32	24.85	23.16	5.49	19.20	43.13
1024	64	32	49.03	43.73	13.08	37.21	83.77
16	128	32	1.01	0.96	0.19	0.84	1.63
32	128	32	1.92	1.71	0.44	1.53	3.15
64	128	32	3.63	3.16	0.96	2.73	5.90
128	128	32	7.43	6.21	2.40	5.26	20.84
256	128	32	13.24	12.07	3.25	9.96	21.93
512	128	32	28.14	23.94	7.82	21.04	46.63
1024	128	32	58.20	48.91	16.59	41.55	93.19
16	256	32	1.13	1.00	0.28	0.85	2.02
32	256	32	2.10	1.77	0.56	1.54	3.31
64	256	32	3.71	3.31	0.94	2.79	6.17
128	256	32	6.98	6.36	1.85	5.54	15.45
256	256	32	14.58	12.70	3.89	10.88	23.01
512	256	32	26.55	24.39	6.34	21.56	44.13
1024	256	32	53.29	48.96	13.02	43.67	92.62
			Matching times				
16	–	–	2.14	2.12	0.19	1.79	2.75
32	–	–	5.27	4.62	1.44	3.70	12.26
64	–	–	9.05	8.39	2.03	6.96	15.85
128	–	–	16.19	15.47	2.99	12.03	27.16
256	–	–	36.06	31.69	10.24	27.03	68.86
512	–	–	72.55	66.55	18.13	57.94	136.61
1024	–	–	166.88	144.11	46.16	120.56	274.52
			SVD solver times				
–	–	–	0.15	0.13	0.08	0.11	0.50
			CPU point cloud generation times				
–	–	–	10.85	13.18	4.05	3.27	17.28



(a) Boxplot of data receiveing times.

(b) Boxplot of DMA start to finish times.

(c) Boxplot of measured SVD solver times.

Figure 11: Boxplots of the latency measurements.

by measuring it to be precisely **524254 cycles**, which at the frequency of 50 MHz that the FPGA is running at equals to 10 485 µs and that is almost exactly the latency we measured from the HPS as seen in Table 1.

Table 1 and boxplot in Fig. 11b show a statistics of 167 measurements of the latency as seen from the HPS between starting the DMA transfers to them being finished. Be aware of the latency including *both* point cloud generation and feature detection. The point cloud generation is slightly slower, but takes similar time as the feature detection as seen in Section 5.1.3.

5.1.3 Feature Detection in FPGA

In this subsection we present results of time taken to produce coordinates of FAST features found in the LiDAR's depth image as described in Section 4.2.2. The measurement was carried out similarly to Section 5.1.2 and is only slightly faster, measured at **520192 cycles**, which is about twice as much as we would've expected and is likely due to the two parallel pipelines of feature detection and point cloud generation taking turns in the SDRAM accesses as there is only one physical DDR interface in the SDRAM controller subsystem [40]. We would expect it to take around $128 \times 2048 = 262144$ cycles with a small but deterministic pipeline delay. From our measurement it takes approx. 10 403 µs.

5.1.4 Feature Description

In this subsection we show measured times taken by the description block of our pipeline running in the HPS (see Fig. 6). We see that it is one of the biggest bottlenecks of the pipeline. As the speed of the description is influenced by the parameter choice, we present it under various settings. The BRIEF (described in Section 2.2.3) feature description algorithm is parametrized by the window size (centered at the found feature and in which the pairwise comparisons are computed) and the number of pairs determining the descriptor vector length. In Table 1 and a boxplot graph in Fig. 12 we show statistics of time measurements with various hyperparameter tunings and see the resulting difference. The BRIEF algorithm's algorithmic complexity is $\mathcal{O}(n)$ where n is the descriptor length; i.e., the number of pairwise comparisons. Because of this, we expect the time to grow linearly with the increase of n, with slight and usually constant overhead added to it and being more prominent in smaller choices of n. This was empirically confirmed by the results in Table 1 and boxplot graph of the measurements in Fig. 12. The choice of the window size has no algorithmic complexity impact, but in real-world, bigger windows will be marginally slower because of worse cache utilization and non-sequential memory access patterns. Our choice of the window size was always non-square as the image dimensions 2048 by 128 were highly non-symmetrical. Thus, we tended to choose higher values in width of the window as opposed to its height.

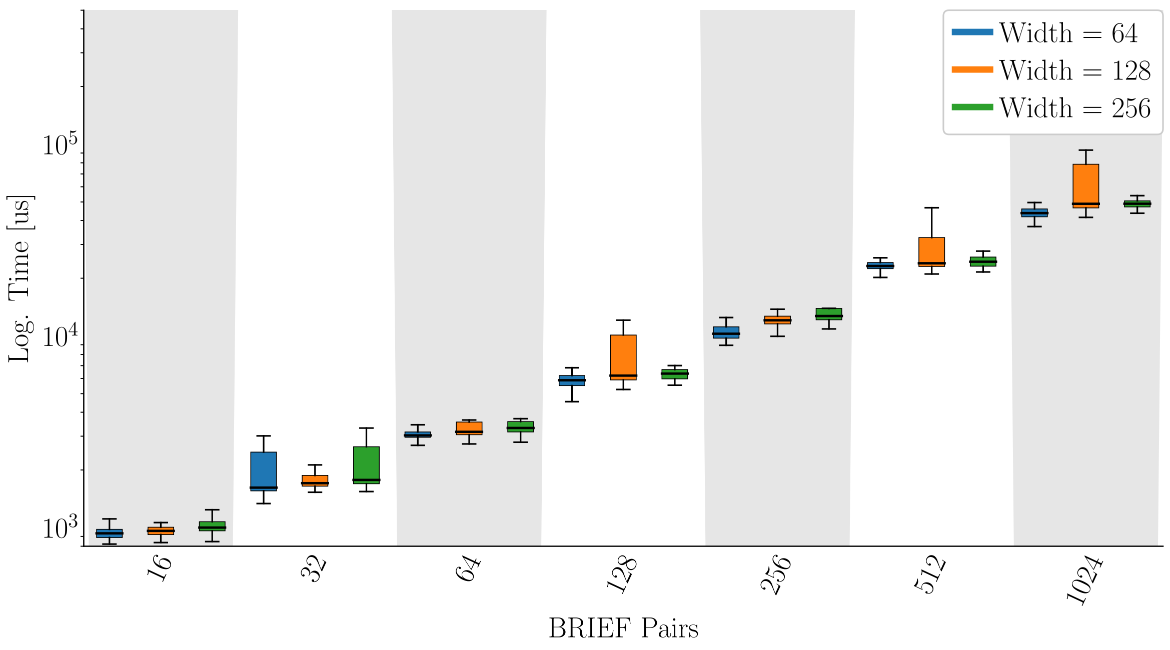


Figure 12: Boxplot of the description times using various hyperparameters.

5.1.5 Feature Matching

In this subsection, we present the measured times required to sucessfully match features found and described by previous steps as per Fig. 6. We found this step to be the most computationally demanding and one which would benefit the most from FPGA implementation, as the comparison of vectors and counting the differences can be implemented very efficiently using combinational logic. The matching's complexity is $\mathcal{O}(n)$ where n is the length of the descriptor vectors to be compared. Because of this, we expect to see linear growth with the increase in the n parameter and that is exactly what can be seen in the statistics Table 1 and a boxplot graphed version in Fig. 13.

5.1.6 SVD Solver

In this subsection, we present the measured times of the rigid transform solver utilizing singular value decomposition as described in Section 2.2.4. We used the C++ Eigen library [39] to calculate all the necessary vector & matrix calculations. Because the FAST component in the FPGA fabric as described in Section 4.2.2 changes its detection threshold to keep the feature count between 128 and 255, the times presented in Table 1 are representative of the average performance as the feature count is going to be similar at all times. Also, because the first matrix multiplication $\mathbf{AB}^\intercal$ yields a 3 by 3 matrix upon which the rest of the calculations is done, the times aren't much affected by the feature count found in the depth image. The SVD is also calculated from 3 by 3 matrix and so is the determinant necessary to check the rotation / flip condition. We thus didn't expect much time spent on those calculations and our expectations are verified by the measurements in Table 1, also visualized using boxplot in Fig. 11c.

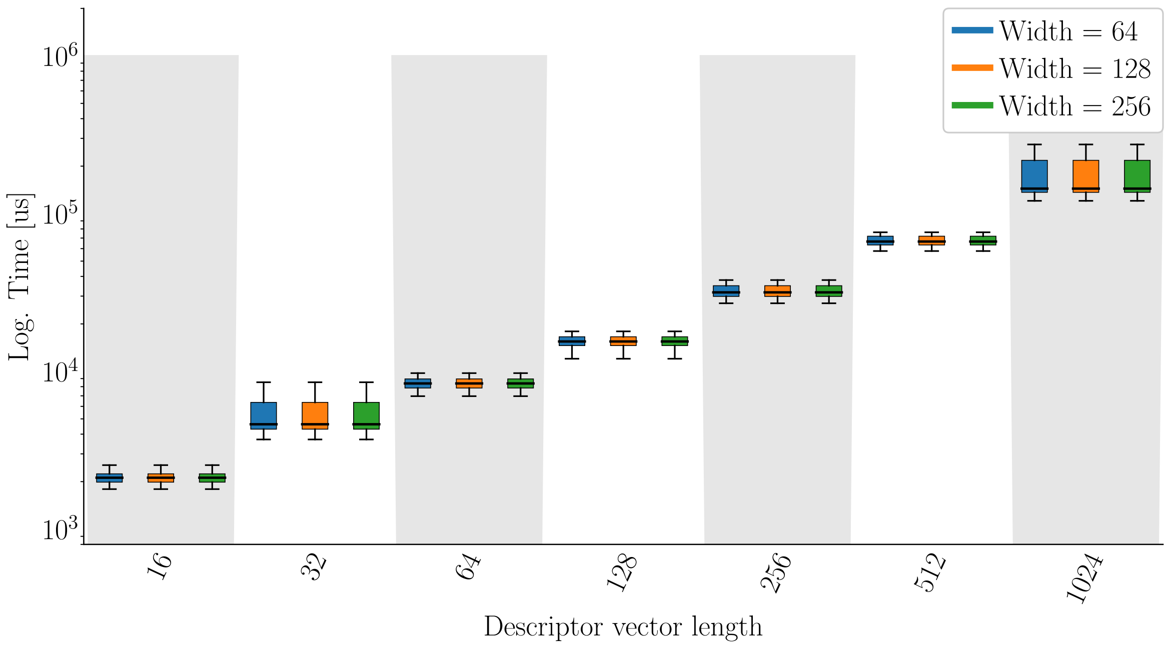


Figure 13: Boxplot of the matching times using various hyperparameters.

5.1.7 Comparison to the CPU Reference Implementation

In this subsection, we present perhaps the most interesting results - the comparison with the currently used CPU implementation of the point cloud generation unit. Currently, all the robots in our laboratory with attached LiDAR unit use the Robot Operating System (ROS) [41] and the official Ouster driver to compute the point cloud and publish into a ROS topic. The ROS is running on the Intel NUC 10i7FNK [42]. We measured both the power draw and times taken while calculating the point cloud. The power draw fluctuated alot but the mean was found at around 300 mA, which at 15 V amounts to 4.5 W. On the FPGA side we did the same procedure and found the increase in power draw to be 65 mA, which at 5 V amounts to 0.35 W, so around **13 times less power** than the reference CPU implementation. In total, the CPU was drawing around 12 W whereas the FPGA drew only around 3.5 W.

The time taken was also measured on the CPU (for the FPGA timing see Section 5.1.2) and the resulting statistics are presented in Table 1. During the experiment, the CPU was running at 2 GHz to 3 GHz with mean around 2.3 GHz. We can see that the time necessary on the CPU has mean of 10.85 ms, which is approx. 0.31 ms slower than the 50 MHz running FPGA as described in Section 5.1.2. One thing to note here is the fact that the FPGA implementation has also deterministic and constant delay, with fluctuations happening only in the HPS subsystem, whereas the CPU reference implementation fluctuated significantly more, which can be seen by comparing the standard deviations or minimas and maximas. Also, even the relatively cheap Cyclone V FPGA we used in this work can house multiple point cloud generating units and thus can in theory process many more LiDAR units with very similar speed as it does with only one unit attached, as all of the components would run in parallel and the only bottleneck would be the DMA accesses. The reason we didn't manage to process multiple LiDAR units was the fact that the UDP packets have to go through the HPS running Linux, in which the receiving buffers were found to be overflowing and thus didn't allow us to receive multiple LiDAR units' data. FPGAs with more network endpoints and direct connection of the endpoint to the FPGA fabric wouldn't suffer from this limitation at all.

5.2 Odometry Experiments

In this section we present the results of the odometry estimation pipeline. To benchmark our solution we conducted multiple experiments with our development board attached to a Husky A200™ robotic platform [43] depicted in Fig. 14a, which we drove through various environments and measured the ground truth using the Leica TS16 [44] total station depicted in Fig. 14b for the translation offsets. Then we conducted a second set of experiments in which we used the XSense MTi-30 [45] attitude heading reference sensor (AHRS) to measure the rotational offsets. We then computed the absolute trajectory error (ATE), introduced in [46] and the relative pose error (RPE), introduced in [47] to draw conclusions on the performance of our design, with translational and rotational ATE for the time synchronized ground truth trajectory $\boldsymbol{P} = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \dots, \boldsymbol{p_n}; \boldsymbol{p_i} \in SE(3)\}$ and estimated trajectory $\boldsymbol{Q} = \{\boldsymbol{q_1}, \boldsymbol{q_2}, \dots, \boldsymbol{q_n}; \boldsymbol{q_i} \in SE(3)\}$ calculated as:

$$\begin{aligned} ATE_{trans} &= \sum_{n=1}^{n} \|trans(\boldsymbol{q}_i^{-1}\boldsymbol{p}_i)\|, \\ ATE_{rot} &= \sum_{n=1}^{n} \|rot(\boldsymbol{q}_i^{-1}\boldsymbol{p}_i)\|, \end{aligned} \tag{5}$$

where *trans()* is the translational part of the resulting matrix and *rot()* is the rotational part.

Similarly for the RPE with a predefined interval Δ:

$$\begin{aligned} RPE_{trans} &= \sum_{n=1}^{n} \|trans((\boldsymbol{q}_i^{-1}\boldsymbol{q}_{i+\Delta})^{-1}(\boldsymbol{p}_i^{-1}\boldsymbol{p}_{i+\Delta}))\|, \\ RPE_{rot} &= \sum_{n=1}^{n} \|rot((\boldsymbol{q}_i^{-1}\boldsymbol{q}_{i+\Delta})^{-1}(\boldsymbol{p}_i^{-1}\boldsymbol{p}_{i+\Delta}))\|, \end{aligned} \tag{6}$$

where SE(3) is a special Euclidean group $SE(3) = \{\boldsymbol{A} \mid \boldsymbol{A} = \left(\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix}\right), \boldsymbol{R} \in \mathbb{R}^{3\times3}, \boldsymbol{t} \in \mathbb{R}^3, det(\boldsymbol{R}) = 1\}$, where $\boldsymbol{R}$ is a 3D rotation matrix and $\boldsymbol{t}$ is a 3D translation vector.

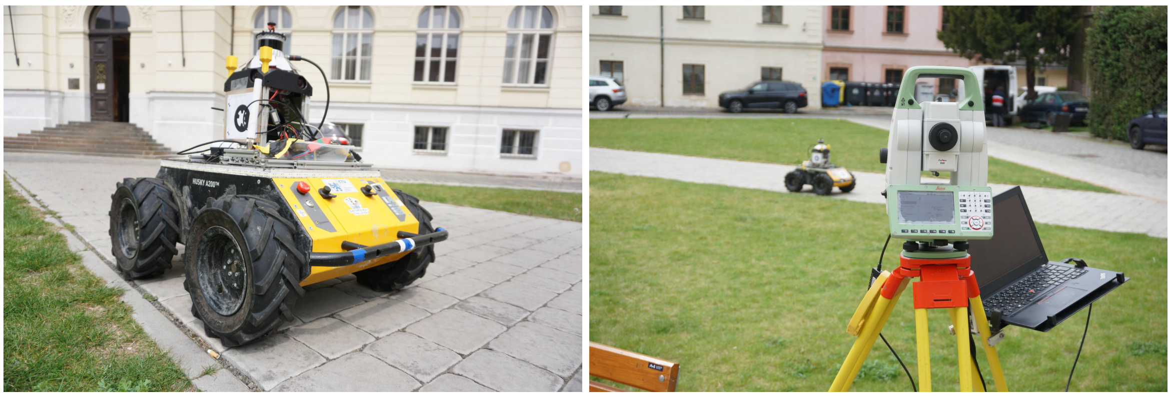
The results of the estimated trajectories are presented in Subsection 5.2.1 and the results of the rotational estimation are presented in Subsection 5.2.2. All of the experiments were conducted inside or outside the Faculty of Electrical Engineering building E of Czech Technical University in Prague.

5.2.1 Translational error evaluation

For the trajectory measurements, we drove the Husky around a parallelogram-shaped courtyard in front of our building in two successive runs. Our pipeline ran at about 5 Hz due to the feature matching time requirements being too high for the pipeline to run at 10 Hz of the LiDAR, and we logged the relative translational changes both using our odometry estimating pipeline and the Leica TS16 Total Station, which we then integrated and plotted in Fig. 15, depicting the ground truth trajectories as $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$, and estimated trajectories as $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$. It's clearly visible that our results diverged significantly from the ground truth trajectory, thus yielding high translational ATE. The trajectories are not similar, closed, symmetrical, or "correct" in any other notion, apart from perhaps few points at the start of the trajectories.

However, upon closer inspection of the data, we suspected the errors to be caused by a relatively few big outlier errors throughout the measurement which then made the rest of the data incorrect. We thus chose to calculate the RPE_{trans} which is not affected by the outliers in this way and the results are promising as depicted in Fig. 16, showing the RPE_{trans} values throughout the second experiment and having a mean and median of 0.624 m, and 0.077 m, respectively. The first experiment lost timestamps due to technical errors and thus failed to be correctly aligned with the ground truth measurements, so only an upper bound on the RPE_{trans} is produced with mean of 0.463 m and median of 0.049 m.

In the second, correctly carried experiment the RPE analysis show that the majority of the odometry estimates were less than 7.7 cm off of the ground truth and that during the first experiment, majority of the odometry estimates were less than 4.9 cm off. These results show that the proposed pipeline is working and that it could greatly benefit from RANSAC [48] algorithm to filter out the outliers ruining the results. This is deemed to be a possible direction of the future work, as well as optimizing the various parameters of the algorithms used in the pipeline.



(a) The Husky A200™ robotic platform. (b) The Leica TS16 Total Station aimed at the Husky A200™ robotic platform.

Figure 14: Photos of the Husky A200™ robotic platform and the Leica TS16 Total Station.

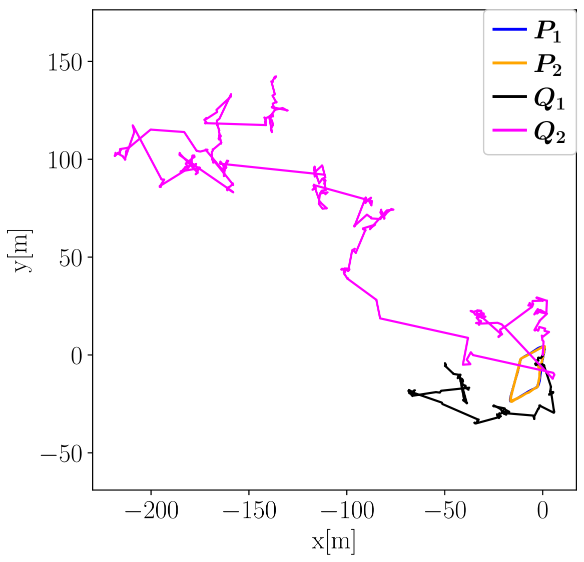


Figure 15: Trajectories from the two runs around the courtyard from the Leica TS16 Total Station ($\boldsymbol{P}_1$ and $\boldsymbol{P}_2$, respectively) and as estimated by our odometry pipeline ($\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$, respectively).

5.2.2 Rotational error evaluation

In the second set of experiments we logged the estimated temporal changes of the yaw, pitch and roll angles and compared them to the ground truth provided by the precise XSense MTi-30 AHRS. We ran four experiments in total, two of which inside a hallway with the surroundings close to the robot as seen in Fig. 17 and two near the open spaced yard in front of our building, similar to the trajectory

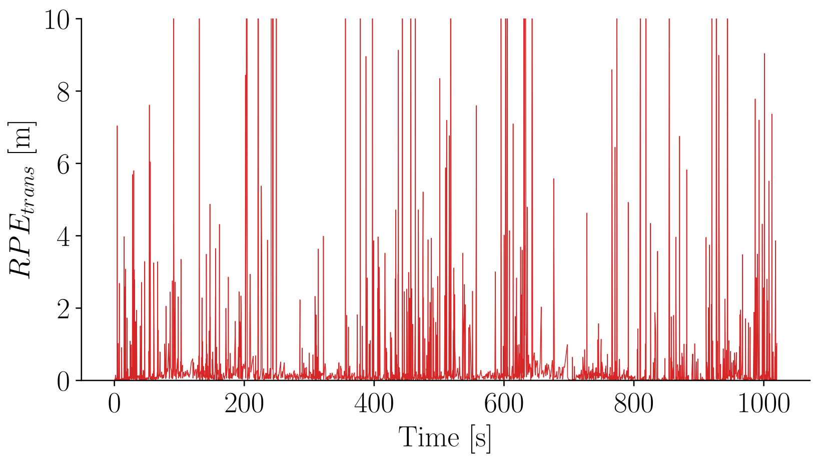


Figure 16: The RPE_{trans} of the second trajectory experiment, limited to 10 m at the RPE_{rot} axis, with few large outlier peaks visible.

experiment shown in Fig. 14b. In both of the locations, we ran two experiments with different values for the maximum allowed pixel distance to the matched featuers in the depth image, described in Section 4.2.4. In both locations, we set the parameter to 16 pixels and 64 pixels and evaluated the ATE_{rot} and RPE_{rot}. The four experiments - hallway with 16px maximum distance, then with 64px max. distance and similarly for the outside experiments - are presented and the graphs of the yaw, pitch and roll angles are visualized in Fig. 18 for individual experiments, respectively. During the experiments, we remotely controlled the robot's yaw angle.

We can see that the trajectories are similar in the case of the hallway experiments but vastly different in the case of the outside experiments. We suspect the latter is due to the fact of bad feature matching that can ruin the absolute error. Also, people movement around the robot might be causing the issues. The non-compensable vertical edge unalignment in the data provided by the Ouster OS0 which is more prominent on the nearby data is also suspected as a source of this issue, as the "fake corners" it generates on vertical edges might actually help the odometry in the close quarters.

However, comparing trajectories directly is prone to outliers in the data as a single outlier can ruin the rest of the data. To counter this effect and understand the results better, we calculated the RPE_{rot} and the plots of the RPE_{rot} for the respective experiments can be seen in Fig. 18. The mean RPEs for the respective experiments is 0.54°, 0.69°, 7.39°, and 13.35°. The medians of the RPEs of the experiments are 0.43°, 0.46°, 2.68°, and 7.20°. In the hallway experiments, this translates to less than $\pm 3px$ offset in the image, in terms of the LiDAR's resolution, as the depth image spans the 360° in 2048px, equaling to approx. 0.18° per pixel. In other words, a noise shifting the tracked feature by less than 3 pixels is enough to cause error of this magnitude. Also, the difference between means and medians in the outside experiments hints that the majority of the error is caused by the outliers. We consider addresing the outlier rejection; e.g., by utilization of RANSAC [48], to be the future work. Nevertheless, the trend in the results shows that with a smaller tracking distance (16 pixels vs. 64 pixels) the resulting RPE is lower. That indicates that the performance of the pipeline can be further improved by changing the parametrization of individual blocks in the processing pipeline.

5.3 FPGA Resource Usage, Precision & Frequency Ratings

In this section we bring an overview of the resulting resource utilization of the used FPGA chip Cyclone V 5CSEBA6U23I7NDK in terms of adaptive logic elements (ALMs), memory elements and digital signal processors (DSPs). Number of available ALMs, memory elements, and DSPs were

Figure 17: The Clearpath Husky A200™ robotic platform inside a hallway.

presented in Section 3.2.1. In case of the point cloud generation unit, we also present its precision as compared to the ground truth calculated using double precision reference implementation on the CPU as it's using fixed point arithmetic and trigonometric functions, both of which are a source of rounding errors.

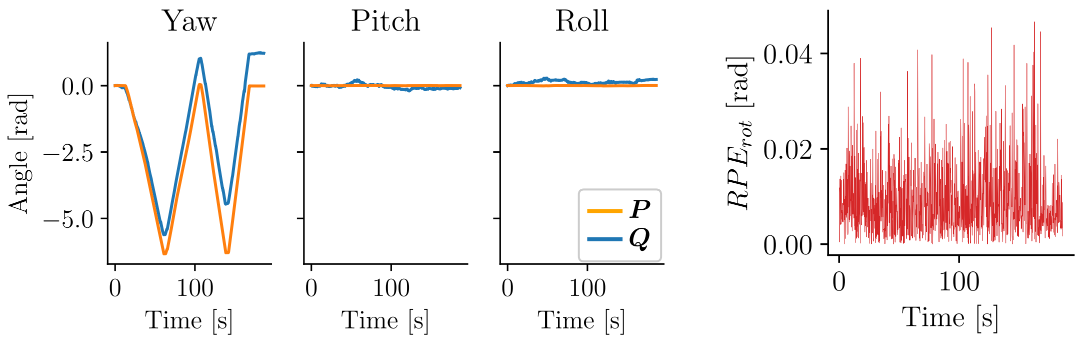
Without any specific pipeline optimizations or synthesis settings, the lower bound on maximum frequency of the entire pipeline within the FPGA fabric is 66 MHz.

The point cloud generator component was tested on random distance data generated in the range of 0 to 2^{20}mm and its absolute difference from the ground truth was on average 1.5 mm in the X axis, 2.68 mm in the Y axis and 3.49 mm in the Z axis. The worst absolute differences were measured to be 2.03 mm, 3.4 mm and 4.13 mm in the X, Y and Z axis, respectively.

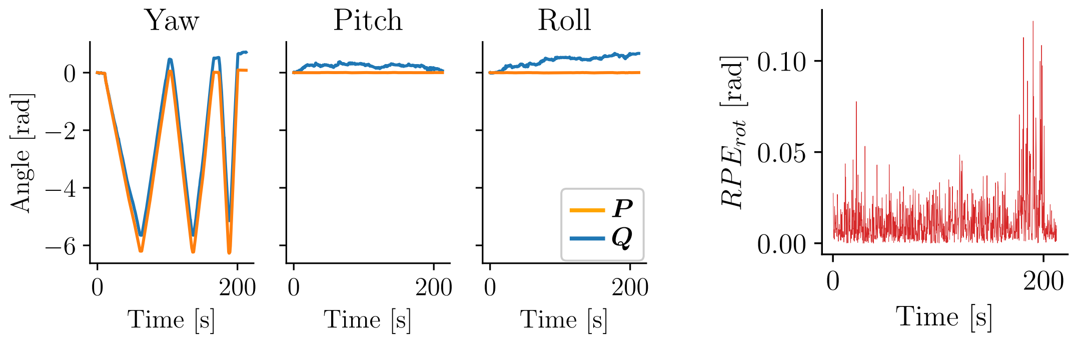
The resulting logic element usage of the "lidar2pntcloud" component is found at 5887 adaptive logic modules (ALMs), 21 DSPs (integer multiplication accelerators implementing the fixed point arithmetic) out of the total 112 of the Cyclone V. To buffer the intrinsics and implement various FIFOs throughout the pipeline, it requires 11 M10K memory blocks out of the total 553.

The FAST detector described in Section 4.2.2 requires only 830 ALMs, 0 DSPs and 11 M10K memory blocks. As the component doesn't do any calculations apart from simple comparisons and additions, there are no DSPs required.

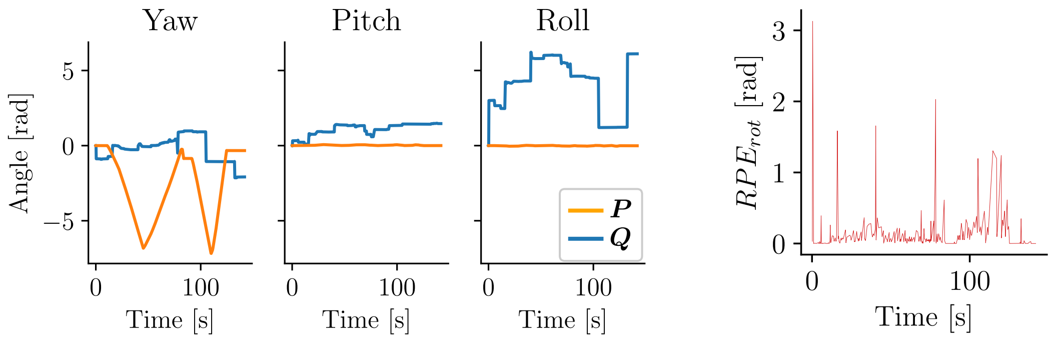
Overall, the architecture utilizes 13131 ALMs, 21 DSPs and 242 M10K blocks equaling to approx. 1880 Mbit.



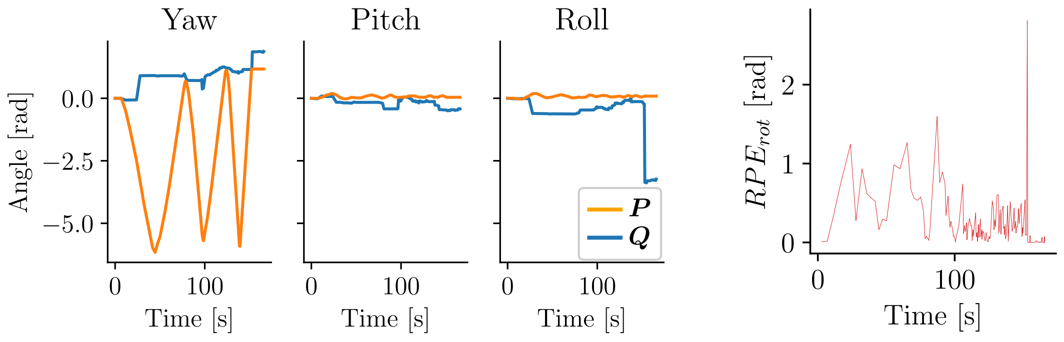
The first experiment in the hallway, with max. feature dist. of 16px.



The second experiment in the hallway, with max. feature dist. of 64px.



The first experiment outside, with max. feature dist. of 16px.



The second experiment outside, with max. feature dist. of 64px.

Figure 18: Measured and estimated orientations of the robot throughout the four conducted experiments along with their respective RPEs.

Chapter 6
Conclusion

In this work we study the utilization of the FPGA architecture for a LiDAR data processing acceleration. We used two tasks to demonstrate the feasability of using the FPGA for this purpose - point cloud generation and ego-motion estimation from the depth image data provided by a LiDAR sensor. We approached this task by developing a custom and general purpose SoPC pipeline running in both the FPGA and the HPS part of the DE10-Nano development board. We provide results based on the proof of concept architecture as described in Chapter 4, yet we establish that the specific choice of algorithms is not important for the PoC as the individual blocks in the pipeline can be swapped for another blocks, hosting a different implementation or different algorithm altogether, and the pipeline would remain working correctly. This, combined with the promising results presented in Chapter 5, is considered by us to be the most important output of this thesis.

We also showed promising results of the unconventional way of estimating odometry, although outliers in the data and non-optimal parametrization makes it one step away from being deployable in production. However, upon inspecting the relative pose error we established that the approach is working well in certain scenarios and that a future work might overcome the limitations discussed in the results Chapter 5.

For a future work we deem the most important steps to be the FPGA implementation of the entire odometry pipeline, including feature description and most importantly the feature matching, as opposed to only a part of it as we present in this work. Another step would be to use FPGA only, from end-to-end, without relying on the Linux running HPS for the UDP communications. This would bring our results few steps forward and only increase the efficiency of the entire system. After that, many of the higher level algorithms or their parts could be also accelerated using the FPGA fabric. Part of the output of this work is to also demonstrate and prove the efficiency that can be achieved with the FPGA technology and to show that future research in this area is relevant.

References

[1] Colin McManus, Paul Furgale, Braden Stenning, and Timothy D. Barfoot. Lighting-invariant visual teach and repeat using appearance-based lidar. *Journal of Field Robotics*, 30(2):254–287, 2013.

[2] Wikipedia contributors. LIDAR traffic enforcement — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=LIDAR_traffic_enforcement&oldid=1065293118`, 2022. [Online; accessed 15-May-2022].

[3] Matteo Palieri, Benjamin Morrell, Abhishek Thakur, Kamak Ebadi, Jeremy Nash, Arghya Chatterjee, Christoforos Kanellakis, Luca Carlone, Cataldo Guaragnella, and Ali-akbar Agha-mohammadi. LOCUS: A Multi-Sensor Lidar-Centric Solution for High-Precision Odometry and 3D Mapping in Real-Time. *IEEE Robotics and Automation Letters*, 6(2):421–428, 2021.

[4] Ouster OS0 Datasheet, revision 04/20/2022. `https://data.ouster.io/downloads/datasheets/datasheet-rev06-v2p3-os0.pdf`. [Online; accessed 26-April-2022].

[5] Wikipedia contributors. Odometry — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Odometry&oldid=1060816258`, 2021. [Online; accessed 26-April-2022].

[6] Wikipedia contributors. Features from accelerated segment test — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Features_from_accelerated_segment_test&oldid=1059161063`, 2021. [Online; accessed 29-April-2022].

[7] Wikipedia contributors. Speeded up robust features — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Speeded_up_robust_features&oldid=1053072748`, 2021. [Online; accessed 11-May-2022].

[8] Wikipedia contributors. Scale-invariant feature transform — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Scale-invariant_feature_transform&oldid=1084345813`, 2022. [Online; accessed 11-May-2022].

[9] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF: Binary robust independent elementary features. In *European Conference on Computer Vision (ECCV)*, volume 6314, pages 778–792, 09 2010.

[10] Ji Zhang and Sanjiv Singh. LOAM : Lidar odometry and mapping in real-time. *Robotics: Science and Systems Conference (RSS)*, 2:109–111, 2014.

[11] Michael Bosse and Robert Zlot. Continuous 3D scan-matching with a spinning 2D laser. In *2009 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4312–4319, 2009.

[12] Tixiao Shan and Brendan Englot. LeGO-LOAM: Lightweight and ground-optimized lidar odometry and mapping on variable terrain. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4758–4765, 2018.

[13] Kamak Ebadi, Yun Chang, Matteo Palieri, Alex Stephens, Alex Hatteland, Eric Heiden, Abhishek Thakur, Nobuhiro Funabiki, Benjamin Morrell, Sally Wood, Luca Carlone, and Ali-akbar Agha-mohammadi. LAMP: Large-scale autonomous mapping and positioning for exploration of perceptually-degraded subterranean environments. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 80–86, 2020.

[14] Petr Čížek, Jan Faigl, and Diar Masri. Low-latency image processing for vision-based navigation systems. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 781–786, 2016.

[15] Peter Biber and Wolfgang Straßer. The normal distributions transform: A new approach to laser scan matching. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2743 – 2748 vol.3, 2003.

[16] Aleksandr Segal, Dirk Hähnel, and Sebastian Thrun. Generalized-ICP. In *Proceedings of Robotics: Science and Systems*, 2009.

[17] K. S. Arun, T. S. Huang, and Steven D. Blostein. Least-squares fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (PAMI)(5):698–700, 1987.

[18] Wikipedia contributors. Singular value decomposition — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Singular_value_decomposition&oldid=1087254983`, 2022. [Online; accessed 15-May-2022].

[19] Petr Čížek. Embedded module for image processing. Master's thesis, Czech Technical University, May 2015.

[20] Wikipedia contributors. Midpoint circle algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Midpoint_circle_algorithm&oldid=1073593456`, 2022. [Online; accessed 29-April-2022].

[21] Jiří Matas, Ondřej Chum, Martin Urban, and Tomáš Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761–767, 2004.

[22] Wikimedia Commons. File:FAST Corner Detector.jpg — Wikimedia Commons, the free media repository, 2021. [Online; accessed 29-April-2022].

[23] Tomáš Krajník, Pablo De Cristóforis, Matias Nitsche, Keerthy Kusumam, and Tom Duckett. Image features and seasons revisited. In *European Conference on Mobile Robots*, 09 2015.

[24] TS Huang, SD Blostein, and EA Margerum. Least-squares estimation of motion parameters from 3-D point correspondences. In *Proceedings IEEE Conference Computer Vision and Pattern Recognition*, volume 10, pages 112–115, 1986.

[25] Officeo. File:netgear-gs308e-8-port-gigabit-plus-managed-switch-default.jpg, 2022. [Online; accessed 16-May-2022].

[26] Bechtle. File:5f4ddefd4c2f853e71dbe7e4-900Wx900H-820Wx820H.jpeg, 2022. [Online; accessed 16-May-2022].

[27] Atyges. File:ouster-os0-atyges.png, 2022. [Online; accessed 16-May-2022].

[28] Mouser Electronics. Mouser DE10-Nano. `https://cz.mouser.com/images/marketingid/2017/img/112951626_TerasicTechnolgies_DE10-NanoDevelopmentKit.png`. [Online; accessed 12-May-2022].

[29] OS0 ultra-wide field-of-view LIDAR sensor for Autonomous Vehicles and Robotics. https://ouster.com/products/scanning-lidar/os0-sensor/. [Online; accessed 26-April-2022].

[30] Wikipedia contributors. Field-programmable gate array — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=1080736735, 2022. [Online; accessed 26-April-2022].

[31] Mahmoud Khaled. *Enhancing the Performance of Digital Controllers using Distributed Multicore/Heterogeneous Embedded Systems*. PhD thesis, 01 2014.

[32] Terasic Technologies. SoC platform - cyclone - DE10-Nano kit. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&amp;No=1046. [Online; accessed 26-April-2022].

[33] Intel Corporation. Intel Cyclone V. https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v.html. [Online; accessed 26-April-2022].

[34] Intel Corporation. DE10-Nano computer system with ARM* Cortex* A9. https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/Computer_Systems/DE10-Nano/DE10-Nano_Computer_ARM.pdf. [Online; accessed 26-April-2022].

[35] Wikipedia contributors. Register-transfer level — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Register-transfer_level&oldid=1041916723, 2021. [Online; accessed 26-April-2022].

[36] Intel Corporation. Intel® High Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. [Online; accessed 26-April-2022].

[37] u-dma-buf. https://github.com/ikwzm/udmabuf. [Online; accessed 26-April-2022].

[38] Wikipedia contributors. Exclusive OR — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Exclusive_or&oldid=1080943142, 2022. [Online; accessed 26-April-2022].

[39] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[40] Intel Corporation. Cyclone V Hard Processor System Technical Reference Manual. https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/hb/cyclone-v/cv-54001.pdf. [Online; accessed 11-May-2022].

[41] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source robot operating system. In *Proceedings of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, May 2009.

[42] Intel Corporation. Intel® NUC products. https://www.intel.com/content/www/us/en/products/details/nuc.html. [Online; accessed 26-April-2022].

[43] Clearpath Robotics. Husky UGV. https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/. [Online; accessed 17-May-2022].

[44] Leica Geosystems. Leica TS16 Total Station. `https://leica-geosystems.com/products/total-stations/robotic-total-stations/leica-ts16`. [Online; accessed 17-May-2022].

[45] Xsense. Xsense MTi 10-series. `https://www.xsens.com/products/mti-10-series`. [Online; accessed 17-May-2022].

[46] Oliver Wulf, Andreas Nüchter, Joachim Hertzberg, and Bernardo Wagner. Benchmarking urban six-degree-of-freedom simultaneous localization and mapping. *Journal of Field Robotics*, 25(3):148–163, 2008.

[47] Rainer Kümmerle, Bastian Steder, Christian Dornhege, Michael Ruhnke, Giorgio Grisetti, Cyrill Stachniss, and Alexander Kleiner. On measuring the accuracy of SLAM algorithms. *Autonomous Robots*, 27(4):387–407, 2009.

[48] Martin A. Fischler and Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24(6):381–395, jun 1981.

Appendix A
Content of the Attachment

```
/
├── orchestrator.c
├── lidar2pntcloud.cpp
├── System.vhd
└── thesis.pdf
```