Agradecimentos

Agradecimentos.....
Acknowledgments.....

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

ALFA-Pd: Aceleração de métodos de redução de ruído em sistemas LiDAR

Palavras-chave: Diversidade de Desenho, Point Cloud, Ruído, Remoção de ruído.

Abstract

ALFA-Pd:Hardware-assisted LiDAR Point Cloud Denoising.

Contents

List of Figures

List of Tables

Glossary

ABS Anti-lock Braking System

ADAS Advanced Driver-Assistance Systems

ALFA Advanced LiDAR Framework for Automotive

APD Avalanche Photodiode

BRAM Block RAM

CW-iToF Continuous-Wave Modulation Indirect Time of Flight

DIOR Dynamic low-Intensity Outlier Removal

DoS Denial-Of-Service

DROR Dynamic Radius Outlier Removal

DSP Digital Signal Processors

dToF Direct Time of Flight

ESC Electronic Stability Control

FCSOR Fast Cluster Statistical Outlier Removal

FF Flip-Flop

FLANN Fast Library for Approximate Nearest Neighbors

FoL Field of Light

FoV Field of View

FPGA Field-Programmable Gate Array

GNSS Global Navigation Satellite System

GUI Graphical User Interface

iToF Indirect Time of Flight

LiDAR Light Detection And Ranging

LIOR Low-Intensity Outlier Removal

LRR Long-Range Radar

LUT Lookup Table

NIR Near-Infrared

ODOA Obstacle Detection and Avoidance

PCL Point Cloud Library

PL Programmable Logic

PM-iToF Pulse Modulation Indirect Time of Flight

PS Processing System

ROI Regions of Interest

ROR Radius Outlier Removal

ROS Robot Operation System

SAE Society of Automotive Engineers

SNR Signal-to-Noise Ratio

SOR Statistical Outlier Removal

SRR Short-Range Radar

TCS Traction Control System

TIA Trans-impedance Amplifiers

ToF Time of Flight

VG Voxel Grid

VTK Visualization Toolkit

1. Introduction

1.1 Motivation

1.2 Goals

1.2.1 Research goals

1.3 Document Structure

2. Background and State of the Art

This chapter starts by addressing the main key sensors that enable autonomous driving in the automotive world, in section 2.1. Since this dissertation focuses on one of those key sensors, a further analysis of LiDAR is done in 2.1.3. Then the inner workings of this sensor are studied alongside its applications. Due to this technology being relatively recent in the automotive world, later is surveyed current challenges of this technology in 2.2.3. This dissertation focuses on one of the current challenges, weather-generated noise, and the current state-of-art solutions are analyzed in 2.3. Finally, a new weather denoising algorithm is presented in 2.3.8, created by fusing mature state-of-art algorithms.

2.1 Automotive perception sensors

Nowadays, more than a decade after the first self-driving car that won the DARPA Challenge, the interest in developing and deploying fully autonomous vehicles has come to a full swing. An autonomous vehicle requires reliable solutions to provide an accurate mapping of the surroundings, which is only possible with multi-sensor perception systems relying on a combination of Radar, Cameras and LiDAR sensors [1, 2, 3, 4]. Working together, they provide the ability to detect the distance and speed of nearby obstacles as well as their 3D shape to safely navigate the environment, contributing to different Society of Automotive Engineers (SAE) Levels of driving automation. While levels 0, 1, and 2 require the driver to monitor the surroundings, with higher levels the automated system monitors the entire driving environment.

2.1.1 Radar

Automotive radars are used to detect the speed and range of objects in the vicinity of the car. An automotive radar consists of a transmitter and a receiver. The transmitter sends out radio waves that hit an object and bounce back to the receiver, determining the object's distance, speed and direction.

Automotive radar sensors can be classified into two categories:Short-Range Radar (SRR), and Long-Range Radar (LRR).

SRR systems use the 24 GHz frequency and are used for short range applications like blind-spot detection [5, 6], parking aid or obstacle detection and collision avoidance [7, 8]. These radars need a steerable antenna with a large scanning angle, creating a wide field of view.

LRR systems using the 77 GHz band (from 76-81GHz) provide better accuracy and better resolution in a smaller package. They are used for measuring the distance and speed of other vehicles while detecting objects within a wider field of view e.g. for cross traffic alert systems. Long range applications need directive antennas that provide a higher resolution within a more limited scanning range. Long-range radar systems provide ranges of 80 m to 200 m or greater [9, 10].

2.1.2 Camera

From photos to video, cameras are the most accurate way to create a visual representation of the world, especially when it comes to self-driving cars.

Autonomous vehicles rely on cameras mounted on all four sides, front, back, left, and right,to create a 360-degree view of their surroundings. Some have a wide field of view, up to 120 degrees, but a limited range. Others concentrate on a narrower field of view to provide long-range visuals[11].

Some vehicles even include fish-eye cameras[12], which have super-wide lenses and provide a panoramic view, allowing a complete view of what's behind the vehicle so that it can park itself.

Although they provide accurate visuals, cameras have their limitations. They can distinguish details of the surrounding environment, however, the distances of those objects needs to be calculated to know exactly where they are. It's also more difficult for camera-based sensors to detect objects in low visibility conditions, like fog, rain or nighttime [13].

2.1.3 Light Detection And Ranging

LiDAR is a method for measuring distances by illuminating the target with laser light and measuring the reflection with a sensor. LiDAR is commonly used to make high-resolution maps with diverse applications like surveying, geodesy, geomatics, archaeology, geography, geology, geomorphology, seismology, forestry and atmospheric physics.

Obtaining distances through measuring travel-time and intensity of light beams date back to the pre-laser decade in 1930 ([14],[15]) but only in 1953 that the concept of LiDAR appeared, which was initially a portmanteau of light and RADAR [16] and introduced in [17]. The LiDAR principle carried out through time unchanged, it can measure distances

by using round-trip time of a flight pulse traveled between the sensor and a target. Between the first appearance of a LiDAR system and now the evolution was focused on the improvement of components, without the appearance of new disrupting techniques.

LiDAR can also provide more information than the three dimensions(x,y,z), it can provide the physical time(t) from repeat LiDAR acquisition and laser return intensity data based on the brightness of single- or multi-wavelength laser returns. An application based on this information is studied by Eitel et. al [18], where the author aprouch topics related to the subjects: earth and ecological sciences (EES) with a special focus in geology, glaciology, hydrology, biogeochemistry, and terrestrial ecology.

2.2 Automotive LiDAR

LiDAR is an active sensor that illuminate the surroundings by emitting lasers. Ranges are measured precisely by processing the received laser returns from the reflecting surfaces. One of LiDAR's competitors is camera-based approaches that despite much progress in processing methods still estimate distances. This approach encounters difficulties when estimating distances for cross-traffic entities, particularly for monocular solutions [19].

Currently, many high-level autonomous vehicles use LiDAR systems as part of their perception systems despite their high cost and moving parts. The output of a perception system comprises the following three levels of information: "Physical description" which consists of the pose, velocity, and shape of objects, "semantic description" that categorizes objects and "intention prediction" that predicts the likelihood of an object's behavior. With this in mind, LiDAR systems outputs are commonly used for object detection, classification, tracking, and intention prediction. But there is a catch, the semantic information carried by LiDAR is more complex than the one acquired from a camera, a contextual sensor is good at object recognition. In practice, LiDARs are combined with cameras to complement each other. A camera is weak in distance estimation, while LiDAR is inadequate for object recognition. Precise physical and semantic information, together with map information, will improve intention prediction without any doubts.

The commonly used LiDAR operate by scanning its Field of View (FoV) with one or several laser beams. The laser beam is generated by an amplitude-modulated laser diode that emits at Near-Infrared (NIR) wavelength. The laser beam is reflected by the environment back to the scanner. The signal received is filtered and the difference between the transmitted and collected signals is measured, being proportional to the distance. The difference in the variation of reflected energy, due to different surface materials, is compensated through signal processing.

A LiDAR system can be partitioned into two main components, the laser rangefinder system and the scanning system. The laser rangefinder comprises the laser transmitter which illuminates the target via a modulated wave and the scanning system that includes the photodetector, which generates an electronic signal from the reflected photon after optical processing and photoelectric conversion. The optics collimate the emitted laser and focus the reflected signal onto the photodetector, and for last, there are the signal processing electronics that estimate the distance between the laser source and the reflecting surface.

2.2.1 LiDAR working principle

The working principle of LiDAR systems stayed untouched through time, consisting of measuring distances by calculating the round-trip travel time of an emitted light signal by a laser and its return, also known by Time of Flight (ToF). To read the ToF, LiDAR systems are composed of an emitter, that transmits the laser signal, and a receiver that receives the back-scattered signal.

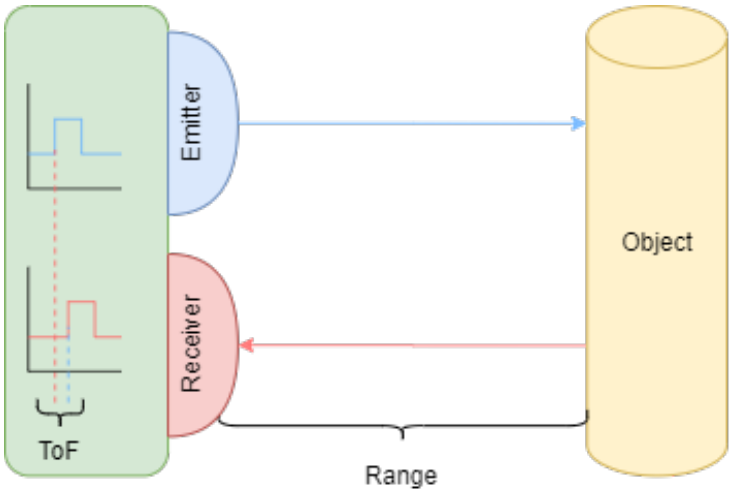


Figure 2.1: LiDAR Working principle.

In order to calculate the range, LiDAR systems use the following mathematical equation:

$$R = \frac{1}{2} c\tau \tag{2.1}$$

In the equation, c is the speed of light in the medium, and τ is the time of flight between the sensor and the target. This calculation is used for obtaining a single point, but nowadays LiDAR systems are capable of 3D representations, containing up to 480,000 points per frame. This is achieved by a predetermined FoV and then the ToF is calculated

for each point. To compute the ToF there are two methods, the Direct Time of Flight (dToF), described in 2.1, and Indirect Time of Flight (iToF).

Imaging Techniques

dToF calculates the ToF based on the signal's pulses and detected reflection. This comes with several advantages, one being simpler to implement, as it does not consider the signal's phase. Another reason is that technology is much cheaper and smaller when compared to other solutions. However, due to the simplicity of the emitter and receiver, measuring ToF comes with low accuracy, suggestible to external factors like external light sources and weather conditions, resulting in a lower Signal-to-Noise Ratio (SNR).

iToF is a more complex technique, but despite that is surely the most mature silicon-based ToF technology with many commercial implementations on the field. Indirect Time of Flight is divided into two technics, continuous-wave modulation (CW-iToF) and pulse modulation (PM-iToF).

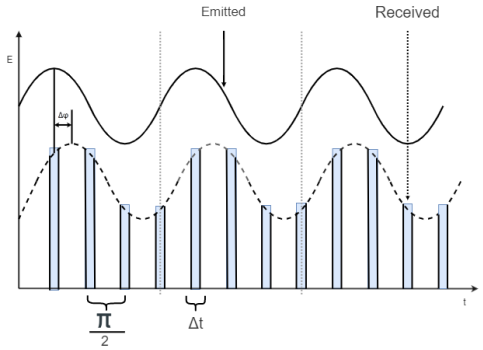


Figure 2.2: CW-iToF scheme.

CW-iToF [20] is typically measured by multiple short-time integrators of duration δt which are mutually displaced by $\pi/2$ to yield four orthogonal samples. These four samples hold the data of depth, reflectivity, and ambient level, which can all be obtained independently.

PM-iToF [20] technique is based on the reflected signal ratio in a series of integrating windows where the signal is modulated in a form of a pulse. Because of the integrated window, the reflected pulse often comes in different integrated windows as shown in figure 2.3 enabling the possibility to calculate the ToF. This technique is often implemented with a CCD/CMOS sensor as they provide charge accumulation and storage capability enabling high-resolution depth imaging and fixed optics. PM-iToF technique comes with other advantages such as reduced laser requirements, as it integrates optical energy, disregarding wavelength.

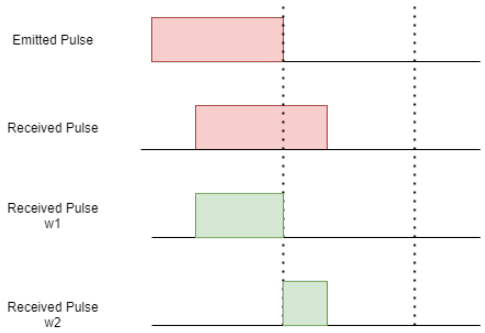


Figure 2.3: PM-iToF scheme.

iToF techniques, like everything, come with a cost associated. The range is significantly lower because longer ranges require longer pulses. Longer pulses are bad for SNR or range gates that influence frame rate. There is also a Multi-object Disambiguation problem caused by the computed distance being an average of all sources of optical reflection. SNR can also be further deteriorated because CCD/CMOS detectors don't typically provide gain, can have a low fill factor, and has lower responsivity than the standard photodiodes used in dToF.

Scanning Techniques

In order to recreate 3D point clouds using LiDAR systems, several techniques were developed. These techniques can be categorized into two main groups, the ones that use beam-steering to scatter a light signal across the environment, and the ones without any mechanical part at all, emitting a light signal for all environments simultaneously.

Figure 2.4: Velodyne VLP-16(left), Velodyne HDL32(Right).

LiDAR systems with beam-steering mechanisms are often categorized as rotor-based mechanical LiDARs [21]. This type of sensor is the most mature scanning technique and its present on a lot of LiDAR systems on the market. Fig 2.4 depicts two rotor-based LiDAR sensors. To achieve 360º horizontal detection, it employs a mechanical rotor to spin the scanning part of the system. Vertically, the field of view is only limited by the number of emitter/receiver pairs. Note that scaling these pairs increases the cost

significantly. Because it has mechanical parts, the frame rate is limited, slowing down the performance of these systems. Most rotor-based LiDAR output information at 10Hz to 20Hz, meaning each second 10 point clouds are sent from the system.

Figure 2.5: Velodyne Vellaray H800.

Scanning Solid-State LiDAR systems [22] differs in several ways when compared to rotor-based mechanical ones. These systems don't have any mechanical parts, therefore they have a limited FoV, but on the other hand, they are cheaper to build because of the lower complexity. This lower complexity also means that they are faster than the others, thus, having a higher frame rate. Due to their scalability in the automobile world, they are often used in groups of sensors in order to increaseFoV and be able to match the mechanical ones.

Flash LiDAR

Flash LiDAR systems are considered solid-state because they don't have moving parts. These systems typically use a flash to illuminate the environment, similar to standard digital cameras, and to receive the signals, has photodetectors that are used to collect the reflected light (2.6). Because of the way the light is emitted, one can say that the Field of Light (FoL) is roughly equivalent to the FoV, being only limited by the receiver that establishes the resolution of the FoV. These systems are considered to be the cheapest available in the market . They have the fastest frame rate due to the lack of moving parts. These systems are also immune to light distortion because all FoV points share the same light source. To increase the range of Flash LiDARs the laser power needs to be increased. Furthermore, due to power limitations, Flash LiDARs have a smaller range in comparison to the other LiDAR types.

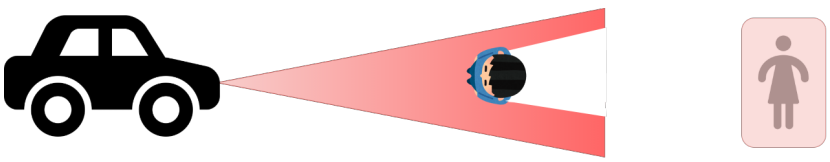


Figure 2.6: Flash LiDAR.

2D Scanning LiDAR

2D Scanning LiDAR systems focus light in a beam to illuminate the target, different from flash LiDAR (2.7). The emitter of these systems has a low divergent laser with some beam steering mechanism. This emitter sets the resolution in one dimension. The receiver no longer is responsible for all the FoV resolution, instead, it is responsible for setting the resolution in the other dimension. These techniques enable longer-range scanning, depending only on the optics of the system, offering also a very flexible architecture. But because the FoV points don't share the same light source anymore, these systems begin to suffer some external interference lowering the SNR.

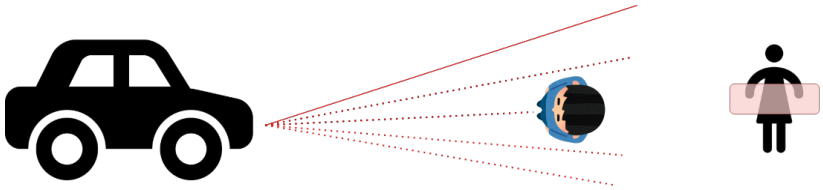


Figure 2.7: 2D Scanning LiDAR.

3D Scanning LiDAR

3D Scanning LiDAR systems are very similar to 2D Scanning LiDAR systems, but with more complex beam steering mechanisms enabling it to scan in both dimensions, x, and y (2.8). These systems are capable of achieving longer scanning ranges, but, they are the most expensive ones because of the beam steering systems' complexity. These beam steering systems also compromise the frame rate, having a very low frame rate compared to flash LiDAR's

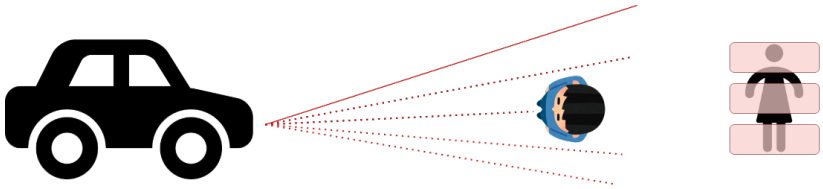


Figure 2.8: 3D Scanning LiDAR.

2.2.2 Applications

LiDAR sensors provide vital information for high-level applications. In this section, a survey of some of the current applications was done.

On-Road measurements

There is a lot of research being done all around the world about LiDARs in the automotive world. Some of the early works around autonomous driving were about road measurements. Rasshofer et al. [23] referenced that automotive radar and LiDAR sensors represent key components for next-generation driver assistance functions.

At the time of writing, some premium vehicles were starting to get features like cruise control, but this article already discussed the need for LiDAR sensors in order to achieve a full autonomous method of transportation.

Not directly connected to autonomous driving, in 2003 LiDAR systems were used to do on-road measurements of automotive particle emissions using ultraviolet LiDAR [24].

Driving Assist

In 2008 an anticollision system named "PROFETA" using LiDAR to do object detection was developed [25]. The main goal was to avoid collisions by breaking the vehicle automatically in dangerous situations. In the year after, there are two more articles published about anti-jamming [26, 27].

Adaptive cruise control using LiDAR was also starting to appear. Pananurak et al. proposed implementation of this system in [28]. Adaptive cruise control is a form of advanced driver-assistance system for road vehicles that automatically adjusts the vehicle speed to maintain a safe distance from vehicles ahead.

Pedestrians Recognition

Since 2011 some work about pedestrian detection and tracking using LiDAR systems start to emerge. Ogawa et al. [29] does the comparison of LiDAR to a camera to see which one is the best at pedestrian recognition, to help reduce traffic accidents involving pedestrians.

Granstrom et al. [30] started improving previous work, [31], to achieve the best performance especially in challenging scenarios using a sampling-based method for data association approximation in multiple extended target tracking, named stochastic optimization.

Patole et al. [32] summarized various aspects of automotive radar signal processing techniques, including waveform design, possible radar architectures, estimation algorithms, implementation complexity-resolution trade-off, and adaptive processing for complex environments, as well as unique problems associated with automotive radars such as pedestrian detection. Even though the article's main focus is radar technology a lot of the concepts discussed can be applied to LiDARs as well. To identify pedestrian

walking, it is possible to check for small changes in range, these changes produce very low Doppler shift. In other words, the micromotion that a target produces is known as a micro-Doppler. Likewise, the frequent motion of limbs creates a periodic pattern in velocity over time, which is also known as the micro-Doppler signature. This signature is greatly investigated by Guo et al. [33].

Pedestrian recognition is a very important subject, and it can be seen in the article [34], where the author writes about safety, inspired by the first death from a self-driving vehicle, Elaine Herzberg on March 18, 2018, that was hit by a self-driving Uber SUV.

2.2.3 LiDAR challenges

LiDAR sensors are promising technologies with a lot of research and development work being done in the field. Despite all the advantages of these systems, the sparse 3D point clouds of a automotive LiDAR sensor can be subject to several sources of noise, such as internal components [35], mutual interference [36, 37], reflectivity issues [38], light [39], adverse weather conditions [4, 40, 41], and others [42], which can corrupt the measurements and the output data.

Most of the current data processing algorithms previously described usually assume ideal weather. However, it is known that adverse conditions can affect the normal operation of the vehicle's perception system. This problem has been thoroughly addressed in the literature [4, 40, 41, 42], where the sensor's response was benchmarked under extremely adverse weather conditions such as fog [43, 44, 45, 46, 47], rain [43, 44, 48, 49, 50, 51], and snow [52, 53]. Despite the broad research that has been devoted to tackling the problems caused by adverse weather, it is crucial to provide solutions that can overcome current limitations related to the performance, accuracy, and overall system complexity.

The downside of LiDAR technology, focused in this dissertation, is the possibility of being influenced by external factors like extreme weather conditions.

Hasirlioglu et al. [54] reference that each year over 1.2 million people die in traffic accidents, to show the need for some kind of drive assistance to prevent further human deaths, but, to have safe driverless vehicles the environmental influences of LiDAR sensors need to be studied. In the year 2016, Kutila et Al. [55] studied the influence of harsh weather conditions, focusing on the effect of fog. This article, concludes that the LiDAR sensor tested degrades his performance by 25% in harsh fog conditions.

First, its essential to understand what fog is, Hasirlioglu et al. describe fog as a complex phenomenon characterized by multiple factors such as droplet microphysics, aerosol chemistry, radiation, turbulence, large/small-scale dynamics, and droplet surface conditions. In other words, fog is a collection of small water droplets in the air, with diameters less than 100 µm. An important note by the author is that foggy conditions differ when

comparing fog at sea, in the atmosphere, and on the ground. The droplet size on the ground is smaller compared to fog droplets in cumulus clouds. One of the conclusions in this article is that stabilizing all possible conditions is too cumbersome for any practical outdoor application where ambient illumination, fog density, snow, etc, are a reality. Furthermore, the author expresses there are two major development steps planned to increase the robustness of the LiDAR system in the automotive world: " Improve the laser scanner hardware by investigating the optimal wavelengths, possible power ranges and optimizing optical and electronic components." and "Develop a software module, which continuously assesses the performance level of the laser scanner according to the obvious changes in detection capability."

In 2017, Hasirlioglu2017 et al. [56] made a fog simulation to test LiDAR performance. As it is known, the fog has a negative influence on wave propagation, especially in the visible range. Hence, surround sensors must be tested under realistic conditions. First, one needs to know the influences of fog on optical communication. The Kruse model ([57]) predicts the specific attenuation using the local visibility and is given by:

$$Attenuation = \frac{3.91}{V}\left(\frac{\lambda}{\lambda_0}\right)^{-q}, \text{(dB/km)}$$

Where V is the visibility in km, Λ the wavelength in nm, Λ0 the visibility reference wavelength (550 nm), and q the coefficient related to particle size distribution in the atmosphere, q depends on visibility and can be calculated as follows:

$$q = 0.585V^{\frac{1}{3}} \text{ for } V < 6\,\text{km}.$$

The results provided by the author show that it is clear that with an increasing number of activated fog layers, the disturbance increases. The influence of small water particles in the air is most visible on a dark background. The fog influence is less visible on a bright background.

The analysis of weather influences on the performance of LiDAR sensors and weather detection are critical steps towards improving safety. In 2019 Heinzler et al.[43] presented an in-depth analysis of automotive LiDAR performance under harsh weather conditions like heavy rain and dense fog. For LiDAR sensors, the most challenging environmental conditions are bright sun, fog, rain, dirt, and spray. The author concludes that the detection quality and range are expected to be impaired as significant laser power is scattered by the atmospheric particles. In dense fog (visibility at 20-40m) the environment perception is highly limited. Nearly all primary echoes are observed at a range of fewer than 5 m and thus caused by the fog. Nevertheless highly reflecting targets like retroreflectors

of the taillights are still correlated with secondary or tertiary echoes. Echo's existence is a drawback regarding Automotive LiDAR systems is that they usually use monochromatic unpolarized light and measure only elastic scattering effects. This is compensated in some systems by providing multi-target detection (multiple echoes), which facilitates the differentiation of detections caused by rain, fog, or snow from those caused by solid objects.

Goodin et al. [48] does a quantitative study on how rain-rate influences Advanced Driver-Assistance Systems (ADAS) performance by developing a mathematical model for the performance degradation of LiDAR as a function of rain-rate. Later this model is incorporated into a simulation of an obstacle-detection system to show how it can be used to quantitatively predict the influence of rain on ADAS that use LiDAR systems. In this article, the author concludes that the resulting point cloud is drastically affected by the increasing rain rate. However, the LiDAR range reduction does not have as strong of an impact on the detection range for the obstacle, meaning that the capability of the sensor to detect obstacles is dependent on factors other than rain.

Byeon et al. [50] address the impact of rain on the LiDAR system by considering the raindrop distributions of different regions. When light encounters raindrops in the air, various physical phenomena occur. These phenomena can be understood by considering the raindrops as particles. In general, particles illuminated by light demonstrate reflection and refraction, absorption, and scattering.

Rivero et al. [41] identified local weather information to recognize if it is working inside its operational design domain and adapt itself accordingly. The author tries to identify weather with LiDAR systems but acknowledges that there are different alternatives to evaluate current weather and road friction in the vicinity of a car by using the information provided by the vehicle's windshield wipers, fog lights, torque, speed of the engine, and tires, Anti-lock Braking System (ABS), Electronic Stability Control (ESC) and Traction Control System (TCS) intervention events, temperature, Global Navigation Satellite System (GNSS) position, steering wheel angle and braking signal. The author concludes that it is possible to use an automotive LiDAR sensor to differentiate between four different weather types: Clear, Rain, Fog, and Snow.

Focusing on LiDAR noise generated by extreme conditions, previously discussed, a further study on the solutions to mitigate this was done. A survey of state-of-art denoising algorithms is presented, in which the goal is to remove all weather-related noise from point clouds. Note that most of the output of LiDAR is represented in point clouds.

2.3 Point cloud weather denoising

Most of the current data processing algorithms usually assume ideal weather. However, it is known that adverse conditions can affect the normal operation of the vehicle's perception system.

Current state-of-the-art solutions include: (i) simulators to analyze the influence of adverse weather under different road conditions and scenarios [46, 48, 49, 50, 51]; (ii) improved background filtering and object clustering methods to better process the roadside LiDAR data [47, 53]; (iii) learning approaches based on convolutional neural networks (CNNs) [44, 46]; and (iv) weather classification systems [41, 43, 47]. The latter use the information provided by the LiDAR to predict weather and adapt the sensor's operation based on atmosphere and asphalt changes, which contributes to a better understanding of the surroundings according to the actual environmental conditions.

Despite the broad research that has been devoted to tackling problems caused by adverse weather, it is crucial to provide solutions that can overcome current limitations related to the performance, accuracy, and overall system complexity.

The LiDAR point cloud can be differently affected due to specific weather conditions. While the water droplets present in rain and fog scatter the emitted light (reducing the operating range and producing inaccurate measurements), the solid particles present in smoke and snow can originate ghost information in the point cloud.

With the most important metrics for a LiDAR weather denoising system in mind, such as accuracy, performance, and algorithm's simplicity, state-of-the-art solutions include voxel grid (VG) filters [58] and algorithms based on outlier removal techniques, e.g., radius outlier removal (ROR) and dynamic radius outlier removal (DROR) [52], statistical outlier removal (SOR) and fast cluster statistical outlier removal (FCSOR) [59], and low-intensity outlier removal (LIOR) [60].

An outlier is an observation that is noticeably different from other (adjacent) observations in a dataset.

Regarding the 3D point cloud data, which can hold millions of generated points per second, there are several points that do not share any relation or characteristics, e.g., distance or intensity, with their neighbors. Such outlier points mostly represent noise in the point cloud, which can compromise the normal operation of the sensor or affect high-level applications such as object detection and classification algorithms. Aside from outlier removal methods, learning-based denoising algorithms also started to emerge [44, 46].

However, they are considered complex since they require real-world datasets and powerful computational resources. Thus, they are out of the scope of this work.

2.3.1 Voxel Grid Filter

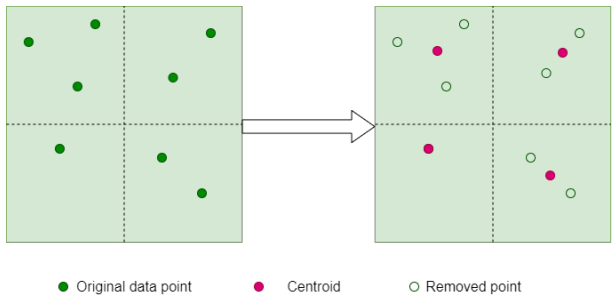


Figure 2.9: VG working principle

Voxel Grid (VG) Filter, [58], differs from other filters in the way that it does not classify points as inliers or outliers, but it downsamples the point cloud. All the points are grouped inside a predefined 3D box in a 3D space and downsampled to an approximated voxel center point, centroid, as it is depicted in Fig. 2.9. Even though the Voxel Grid is a downsample filter, it can be considered a noise removal one. When only noise points are inside the box, they will be removed, and replaced by the centroid, thus reducing the number of noise points from the point cloud. However, not only noise will be removed, because all points are downsampled, so relevant information is lost during execution.

2.3.2 Radius Outlier Removal

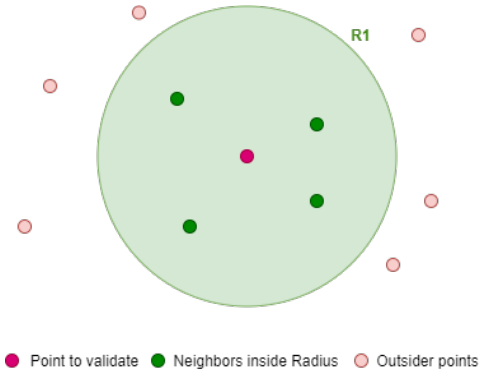


Figure 2.10: ROR working principle

The working principle of the Radius Outlier Removal (ROR) is depicted in Fig. 2.10, for every point it computes the mean distance to its neighbors within a user-defined radius **(R1)** by using a k-d tree data structure. If the number of neighbors within the specified radius is below the user-defined threshold the point will be classified as an outlier, being removed from the point cloud. Thus, the performance of this filter greatly depends on the specified radius and minimum of neighbors threshold. This filter has the advantage

of being easy to implement due to its simplicity, but when applied on 3D LiDARs the fixed filter radius search becomes a problem, because as the detection range increases the space between points will also increase due to the horizontal and vertical resolutions of LiDARS systems. Thus, the points detected by LiDARs at great distances most likely will be removed by this filter.

2.3.3 Statistical Outlier Removal

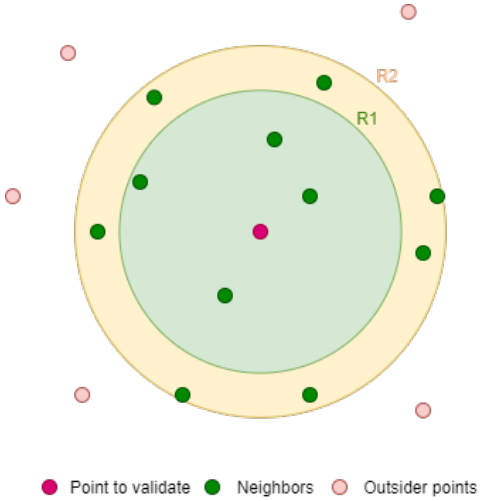


Figure 2.11: SOR working principle

The Statistical Outlier Removal (SOR) filter concept is somewhat similar to the ROR as it classifies points as inliers or outliers based on neighbor information. But instead of having a fixed search radius or a minimum of neighbors threshold as the ROR filter has, it instead computes the mean distance of each point to its neighboring points considering the k-nearest neighbors **(R1)** as depicted in 2.11. When these points are greater than the sum of the mean distance and the standard deviation **(R2)** they are classified as outliers. Similar to the ROR filter, the performance of **SOR** filter depends on the number of nearest points, and the amount of times the stander deviation is calculated. ROR and SOR filters have the same disadvantages, both need to find the number of neighbors which comes with a heavy computational cost. Moreover, because of point cloud sparse distances, the greater the distance of the point to the LiDAR sensor, the higher the possibility of a point being removed as an outlier.

2.3.4 Fast Cluster Statistical Outlier Removal

The Fast Cluster Statistical Outlier Removal (FCSOR) [59] is an improved version of the SOR filter, This method is a somewhat junction of a voxel subsampling method

and statistical outlier removal. By adding the voxel subsampling step the computational complexity reduces improving time requirements. This is done by reducing the number of clusters and performing some computations needed in parallel. However, the improvements shown in this filter are only performance-related, and still cannot achieve real-time capability.

2.3.5 Dynamic Radius Outlier Removal

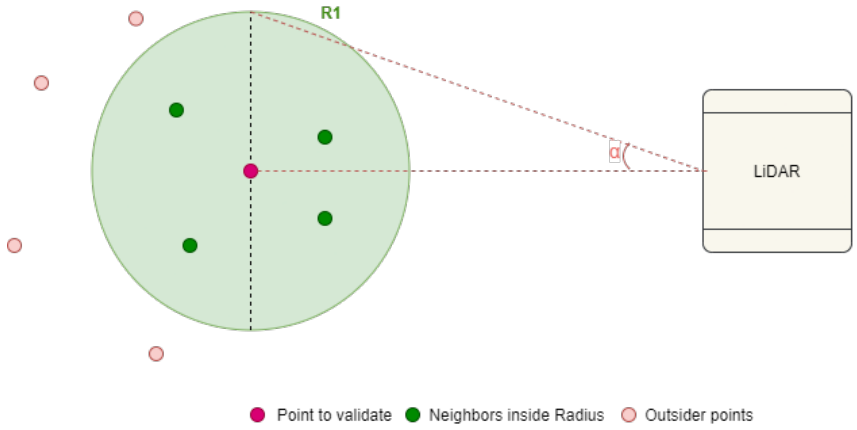


Figure 2.12: DROR working principle

The Dynamic Radius Outlier Removal (DROR) [52] filter was developed to compensate for the accuracy problems shown by the SOR and ROR in 3D LiDAR sparse point clouds. The problem with 3d LiDAR point clouds is that the further the point measured the sparser the point cloud will be. Instead of having a fixed radius like ROR filters, DROR has a dynamic one, reducing distant point losses. This is done by changing the search radius for neighboring points as the distance measured increases **(R1)** multiplied by the sensor angular resolution (α) as depicted in Fig 2.12. In the results presented by the author, the DROR filter outperforms conventional filters, in fact, it improves the accuracy by more than 90% compared to the conventional ROR. However, despite its good accuracy, it still has performance issues, having a high computational cost.

2.3.6 Low-intensity Outlier Removal

Low-Intensity Outlier Removal (LIOR) is a method proposed by Park et al. that aims at improving the speed and accuracy performance limitations of previous methods by removing the noise caused by snow or rain based on the intensity of the reflected light [60]. Noise points usually present a lower intensity value when compared with neighbors at the same distance. Thus, every point below a defined threshold value is classified as an outlier. To reduce the false positive ratio, a second step is applied to each outlier, which

can be turned into an inlier if several neighbors (defined by a threshold) are detected within a specified distance.

The working principle behind LIOR is based on the ROR algorithm (depicted by Fig. 2.10) with the addition of the point intensity information. When comparing LIOR with the previous filtering methods, it can achieve filtering speeds up to 12x faster than SOR, and 8x faster than DROR. However, real-time filtering in high-speed vehicles is only possible if the method is applied only to certain Regions of Interest (ROI) rather than the full point cloud. Regarding the accuracy, the noise points can be filtered with the same efficiency as the DROR method.

In their evaluation, LIOR claims to achieve a false positive ratio of 1%, while DROR reached almost 50% of points wrongly classified as outliers. LIOR was deployed and tested on an Intel Core i9-9900KF CPU (at 3.60 GHz), with 32 GB of RAM.

2.3.7 Discussion

To summarize, all previously discussed state-of-art weather denoising filters offer different advantages. The VG filter offers real-time processing, point cloud compressing and a somewhat denoising filter performance. SOR and ROR are the most mature filters, being used in innumerous LiDAR applications, but in an autonomous application, they lack filter performance due to the point cloud sparsity. FCSOR was developed to improve the time performance of SOR, parallelizing and adding a voxel step to increase performance, but it lacks in true positive ratio. DROR and LIOR can be considered the best state of art algorithms.

Table 2.1: Weather denoising algorithms summary

Algorithm	Low complexity	Sparsity ready	Dynamic radius	Intensity Based	Good TP ratio	Real Time
VG	X					X
SOR	X					
FCSOR						X
ROR	X					
DROR	X	X	X		X	
LIOR	X			X		X

In Tab 2.1 is depicted the denoising methods focused on this dissertation, indicating the advantages and disadvantages of the state-of-art methods. By analyzing the table, one can identify that the most advantageous algorithms are DROR and LIOR. DROR is designed to address the sparsity effect in outlier removal based filters. DROR states that can achieve a very high true positive rate but at a cost of heavy computational resources. On other hand, LIOR states that the filter can achieve real-time, by classifying points based on intensity values.

By analyzing the Tab 2.1, one can identify that there is a lack of an algorithm that can achieve the good true positive ratio of DROR, while still complying with time constraints as LIOR provides. It is possible to combine the best features of DROR and LIOR to create a better algorithm capable of providing a good true positive ratio while complying with time constraints.

2.3.8 Dynamic low-Intensity Outlier Removal

Dynamic low-Intensity Outlier Removal (DIOR) is a novel approach, proposed in this work [61], for weather denoising based on existing outlier removal techniques. DIOR follows the working principle of LIOR, which classifies outlier points based on their intensity values within a constant search radius***R1*** (as used by ROR). However, instead of using the ROR principle, DIOR follows the DROR strategy, which classifies outlier points based on a search radius ***R1*** that dynamically increments as the distance to the objects increases.

DIOR differs from DROR in the way that before calculating the search radius, it's added an extra step where the filter deletes points with intensity values below a specified intensity threshold. DIOR sets aside of LIOR, because instead of not accounting for the space between points, the sparsity, DIOR uses the DROR approach, calculating the search radius using the sensor angular resolution and distance to the sensor.

By combining DROR and LIOR, DIOR checks all the boxes in Tab 2.1. DIOR was developed in this dissertation, and can be used in software-only execution, or resorting to dedicated hardware accelerators.

3. Platform and tools

This chapter identifies the platform and the tools that will be used throughout the development of this work. Because the main goal of this dissertation is to have an embedded system running all algorithms in software and hardware, the platform chosen will be described in 3.1, then all the tools that will be used will be described in 3.2 and 3.3

3.1 Reconfigurable technology

The main goal of this dissertation is to implement a framework capable of using hardware-accelerated denoising filters, which implies the usage of a CPU and a deployable hardware space, all in the same platform.Field-Programmable Gate Array (FPGA) technology gives the flexibility and functionalities needed for this work. FPGAs contain an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" allowing blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple Flip-Flop (FF) or more complete blocks of memory Block RAM (BRAM).

3.1.1 Zynq UltraScale+ MPSoC ZCU104

The platform selected to run all the functionalities developed in this dissertation was the Zynq UltraScale+ MPSoC ZCU104, featuring the Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC for hardware development enabling the design of embedded applications such as ADAS with support of video codecs and the most common peripherals and interfaces for embedded vision solutions. The MPSoC features a Processing System (PS) that includes a quad-core Arm Cortex-A53 running at a clock speed of 1.2 GHz application processor, a dual-core Cortex-R5 real-time processor, a Mali-400 MP2 graphic processing unit, a 4KP60 capable H.264/H.265 video codec, Programmable Logic (PL) with FPGA technology, and 2GB of DDR4 memory running at 525 MHz. Regarding the

FPGA, it holds 23040 Lookup Table (LUT)s, 46080 FFs, 312 BRAMs, and 1728 Digital Signal Processors (DSP)s. Thus, with this platform, it is possible to deploy the work developed in this dissertation without exhausting the available resources.

3.2 Robot Operating System

Robot Operation System (ROS) is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It provides services designed for a heterogeneous computer cluster, and implementation of messaging-passing between processes and devices in the same network.

ROS processes are represented as nodes in a graph structure, connected by edges called topics. ROS nodes can pass messages to one another through topics, make service calls to other nodes, provide a service for other nodes, or set or retrieve shared data from a communal database called the parameter server. A process called the ROS Master makes all of this possible by registering nodes to itself, setting up node-to-node communication for topics, and controlling parameter server updates. Messages and service calls do not pass through the master, rather the master sets up peer-to-peer communication between all node processes after they register themselves with the master. This decentralized architecture lends itself well to robots, which often consist of a subset of networked computer hardware, and may communicate with off-board computers for heavy computation or commands.

A node is a single process running the ROS graph. Every node has a name or is anonymous having a randomly generated identifier, which is registered with the ROS master before it can take any other actions. Nodes are at the center of ROS programming, as most ROS client code is in the form of a ROS node that takes actions based on information received from other nodes, sends information to other nodes, or sends and receives requests for actions to and from other nodes.

Topics are named buses over which nodes send and receive messages. In order to send messages to a topic, a node must publish to said topic, while to receive messages it must subscribe. The publish/subscribe model is anonymous: no node knows which nodes are sending or receiving on a topic, only what it's sending/receiving on that topic being the types of messages passed can be user-defined. In this dissertation, the content of these messages will be point clouds and configuration settings.

A node may also advertise services. A service represents an action that a node can take which will have a single result. As such, services are often used for actions that have a defined beginning and end, such as capturing a single-frame image, rather than

processing velocity commands to a wheel motor or odometer data from a wheel encoder. Nodes advertise services and call services from one another.

ROS runs on top of a Linux distribution, because of its dependence on a large collection of open-source software. In this dissertation, the ROS environment will allow communication between the embedded system and a desktop to configure and visualize the results from the embedded system, by publishing and subscribing to custom ROS topics.

3.3 Point Cloud Library

The Point Cloud Library (PCL) [62] is an open-source library of algorithms designed to process 2D and 3D point clouds containing algorithms for filtering, feature estimation, surface reconstruction, 3D registration, model fitting, object recognition and segmentation. Each module is implemented as a smaller library that can be compiled separately and has its own data format for storing point clouds, Point Cloud Data (.pcd), but also allows datasets to be loaded and saved in many other formats like Polygon File Format (.ply).

The algorithms have been used, for example, for perception in robotics to filter outliers from noisy data by implementing filters like ROR and SOR, fuse two 3D point clouds, segment relevant parts of a scene, extract keypoints and compute descriptors to recognize objects in the world based on their geometric appearance, and create surfaces from point clouds and visualize them.

PCL is a cross-platform software that runs on the most commonly used operating systems written in c++ and fully integrated with ROS enabling it to run on an embedded system like the platform chosen to deploy the work of this dissertation. Important to note that this library requires several third-party libraries to function like the Eigen library mostly used for mathematical operations, Visualization Toolkit (VTK) for enabling point cloud visualization, Boost for shared pointers, and Fast Library for Approximate Nearest Neighbors (FLANN) for quick k-nearest neighbor search.

3.4 QT

Qt is a software toolkit for developing multi-platform Graphical User Interface (GUI) that run on desktop, mobile, and embedded platforms supporting various compilers including the GCC c++ compiler. In this dissertation, Qt is used to create ALFA-DVC,

the debugger, configurator and visualizer tool, that allows real-time point cloud visualization debugging and configure all the parameters of the software/hardware running in the embedded system.

3.5 Open Embedded

OpenEmbedded is a build automation framework and cross-compile environment used to create Linux distributions for embedded devices. OpenEmbedded is the recommended build system of the Yocto Project, which is a Linux Foundation workgroup that assists commercial companies in the development of Linux-based systems for embedded products.

The build system is based on BitBake "recipes", which specify how a particular package is built, but also includes lists of dependencies and source code locations, as well as for instructions on how to install and remove a compiled package. OpenEmbedded tools use these recipes to fetch and patch source code, compile and link binaries, produce binary packages (ipk, deb, rpm), and create bootable images.

OpenEmbedded collections of recipes are structured in multiple layer configurations in which the lowest layer including platform-independent and distribution-independent metadata is called "OpenEmbedded-Core". Architecture-specific, application-specific, and distribution-dependent instructions are applied in appropriate target support layers that can override or complement the instructions from lower layers. In this dissertation, OpenEmbedded is used to generate the Linux image with ROS, PCL, and all the packages required to run this work, furthermore, OpenEmbedded is also used to cross-compile all the software by executing a costume recipe to fetch the code from GitHub and installing it into the ROS environment.

3.6 Advanced LiDAR Framework for Automotive

Advanced LiDAR Framework for Automotive (ALFA) is an open-source Framework for Automotive that aims to offer a multitude of helpful features for the validation and development of automotive LiDAR systems. These features include: (i) Generic and multi-sensor interface; (ii) Pre-processing algorithms for data compression, noise filtering, ground segmentation, along with others; (iii) Configurable output for High-level applications; (iv) Reconfigurable point-cloud representation architecture.

ALFA can be divided into several Cores. These Cores are independently configurable, removable, and expandable, boosting ALFA's functionality and flexibility. The individual cores offer basic communication interfaces and resources. From here, an algorithm can be built and added to its specific Core, eliminating the hassle of developing an entire system

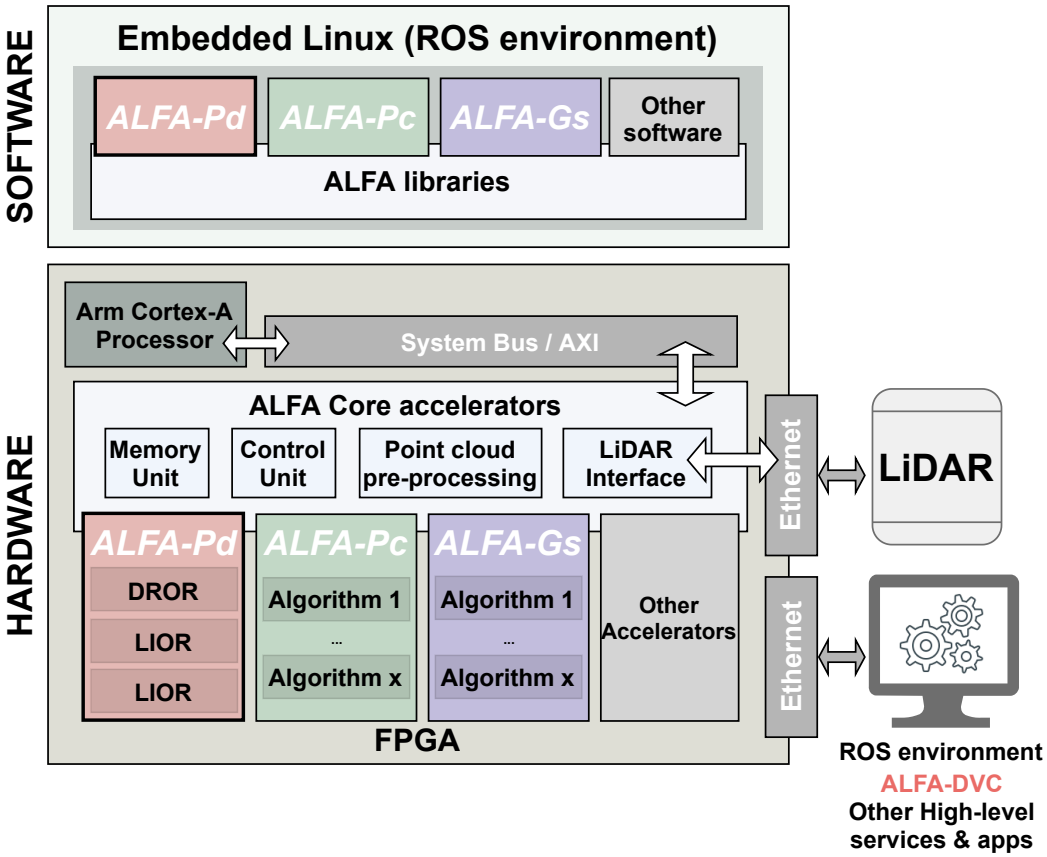


Figure 3.1: ALFA architecture block diagram

to validate it or part of it. As seen in Figure 3.1, ALFA is also divided into hardware and software capabilities. The ALFA Core accelerators are the main components of the system, providing a Control Unit, a Memory Unit, LiDAR interface, and point cloud pre-processing modules. Being the latter where this dissertation can be included. Although the framework proposed in this dissertation can as executed standalone. By integrating with ALFA, it is possible to access point cloud data without the overhead of PS-PL communication, increasing even further the performance of the proposed work.

4. ALFA-Pd Framework

The ALFA-Pd framework enables point cloud visualization in real-time, filtering processing, filter evaluation, and system configuration, using the ALFA-DVC tool. ALFA-Pd also offers point cloud weather denoising on an embedded system, with an option to use built-in hardware accelerators, providing real-time filtering. ALFA-Pd framework gives access to the state-of-art denoising method and a novel approach to weather denoising developed in this dissertation named DIOR.

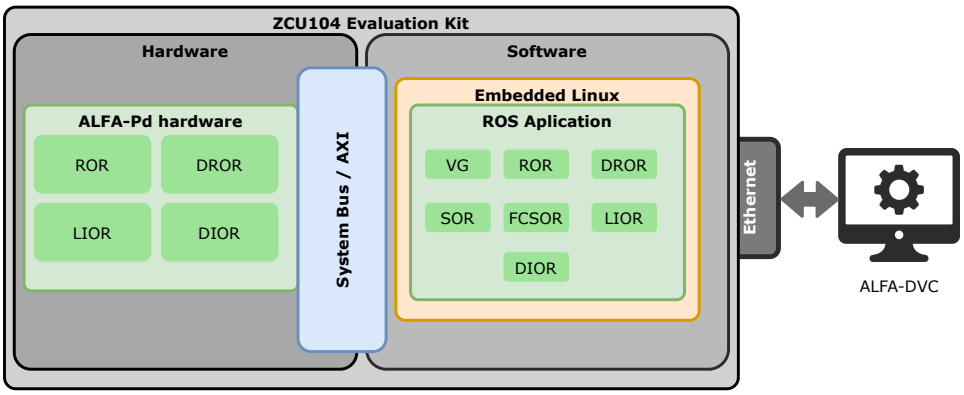


Figure 4.1: ALFA-Pd Framework architecture.

Fig. 5.1 depicts the ALFA framework that, following a hardware-software co-design approach, enables the fast deployment and evaluation of state-of-the-art weather denoising methods both in hardware and software. Regarding the software layer, ALFA features the Robot Operating System (ROS) environment on top of a minimalist embedded Linux, providing different levels of abstraction for high-level applications, e.g., the ALFA-DVC tool. This tool, developed in this work, provides several features such as debug, a real-time point cloud visualizer, platform setup, and algorithm configurations.

The framework supports a collection of software-based denoising methods (assisted by the PCL library) and core-libraries that abstract and interfaces the weather denoising accelerators.

The communication is handled by the advanced extensible interface (AXI)-4 system bus, where the AXI4-full is used for high-performance memory accesses and the AXI4-stream for high-speed point cloud data streaming.

The hardware features encompass several core blocks such as a hardware interface for the software libraries; a memory management module; and distance calculator units to be used by the hardware weather denoising accelerator. The hardware platform also provides two Ethernet ports to respectively connect the LiDAR sensor and to make the denoised point cloud available to high-level applications, but in this dissertation, only the point clouds obtained by ALFA-DVC will be used.

The weather denoising framework and the ALFA-DVC tool currently support the deployment and evaluation of the following state-of-the-art denoising algorithms: VG, SOR, FCSOR, ROR, DROR, LIOR, and the new method developed in this dissertation, DIOR.

These software-only implementations are supported by the PCL [58] software, which provides a set of modules for point cloud processing, e.g., filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation.

4.1 ALFA-DVC

The ALFA-DVC tool was created to offer a way to quickly deploy state-of-art 3D point cloud algorithms. To fulfill this goal, the tool offers real-time point cloud visualization, resorting to a ROS environment. Because the algorithms need to be evaluated and evaluating up to 600,000 points per frame by hand is imaginable, ALFA-DVC offers a way to evaluate point clouds using a box system. To make an all-in-one tool, ALFA-DVC supports the ALFA-Pd framework, communicating through ROS topics using ROS messages.

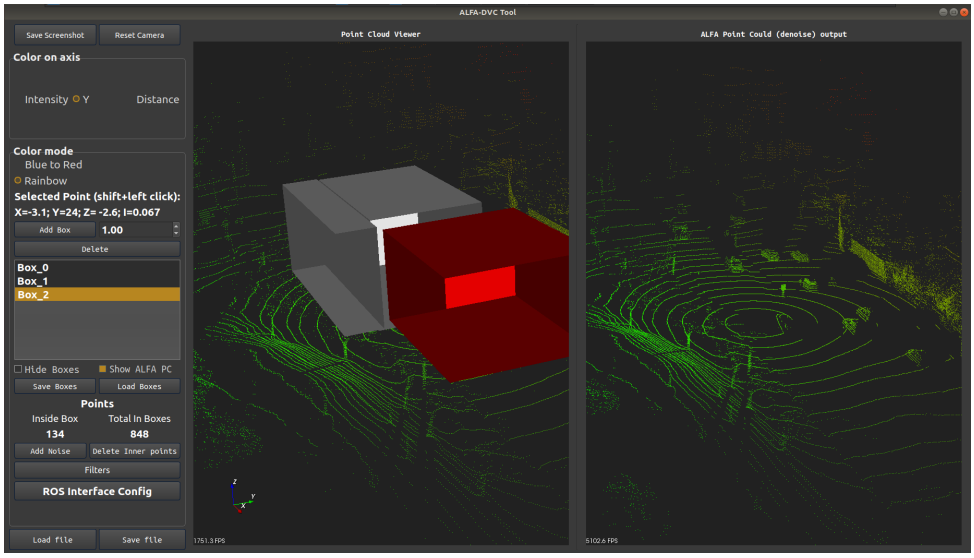


Figure 4.2: ALFA-Pd system architecture.

The ALFA-DVC tool, depicted in Fig. **??**, is a high-level and cross-platform QT application that runs on a desktop system. It enables the real-time visualization of up to two point clouds, allows the deployment and evaluation of software-based algorithms directly on top of a point cloud, and provides the debug and configuration of weather denoising methods.

Point cloud data can also be loaded into the ALFA-DVC tool using Point Cloud Data (.pcd) and Polygon File Format (.ply) files stored in the file system or through the available ROS interface. This interface can provide point clouds directly received from a LiDAR sensor connected to the ALFA hardware platform or from a ROS topic running in a local ROS node. The tool also enables screenshots saving of the point cloud currently being played, as well as to publish the noised/denoised point cloud on a new ROS topic.

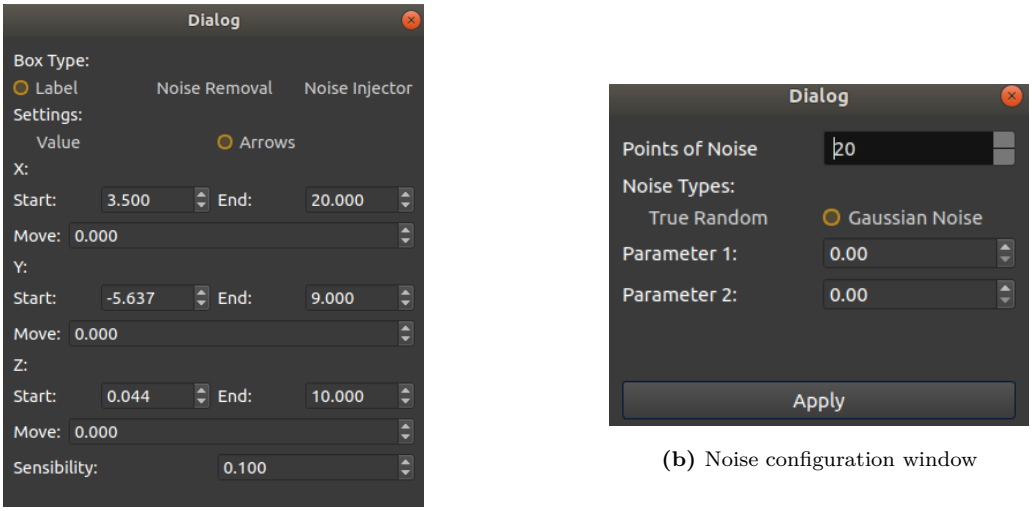
To help point cloud visualization and analysis, ALFA-DVC offers three types of coloring. The point cloud can be colored by the intensity of a given point, ranging from blue when intensity is low, to red when the intensity is high. The user can choose to color points based on the y axis. The point cloud can be colored using the point distance to the sensor, to give a better look at what radius can be used by DROR in the calculation of the search radius.

To obtain the point parameters from the interface, the user must shift plus the left mouse click on the point. By presenting the point parameter, it is easier to validate what LIOR claims, that the noise points present lower intensity than their neighbors.

ALFA-DVC provides a user-friendly way to interact and analyze the point cloud through a box system. The user can create and delete a box. These boxes are placed in the 3D space as depicted in Fig **??** and displayed in a list, where the user can see and interact with all the placed boxes. Because the placed boxes can take a lot of visual space, the user can hide the boxes by clicking in the check box designated for that. The user can also save/load box configuration from the file system, allowing for later use of the session configurations.

In ALFA-DVC there are three types of boxes, label boxes, noise removal boxes, and noise injection boxes. The label boxes are used by the algorithm evaluation system to classify the performance, the noise removal boxes give the option to delete points inside them, and the noise injection boxes are used by the noise emulator module that injects noise points in the point cloud.

When the user double clicks in a box item, inside the box list, a new window pops up. In this window, the user can identify which type of box it is, and move it in the 3D space. As depicted in fig. 4.3a the user can adjust all the coordinates of the box, increasing/decreasing the box size. The user can also move the box, on one axis each time. The user can also edit the sensibility to which the arrows respond.



(a) Box configuration window

(b) Noise configuration window

Figure 4.3: ALFA-DVC subwindows

When the data points are collected in ideal weather conditions, the ALFA-DVC can also emulate and generate different weather noise applied to specific regions of the point cloud through a box system. Such a box system allows to label and adds/remove noise points, as well as to analyze the algorithm through several performance and accuracy metrics. This can be done by clicking on the dedicated button. By clicking on the button a small window appears, depicted in 4.3 offering all the configurations related to the emulation of point cloud noise.

To use, or watch filter performance, the user must click on the specified button in the main window. The filter window allows quick use of denoising algorithms, algorithm evaluation, and filter configuration. It also allows to hide/show the filtered point cloud, to better analyze the differences between them.

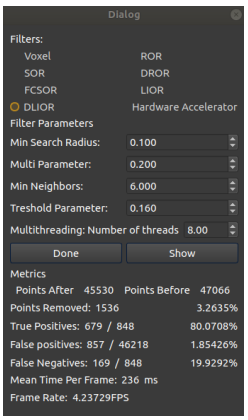


Figure 4.4: ALFA-DVC filter window.

In fig 4.4 is depicted the filter window, where it's possible to interact with everything related to weather denoising filters. The user can select the filter to be applied within

the panoply of filters supported, including DIOR, the filter developed in this work. After selecting which filter to use, the user can configure the filter selected, including the number of threads in the supported algorithms. The user can apply the filter to the point cloud, and if there are boxes that are labeling the points, all the metrics will be displayed in this window.

4.2 ALFA-Pd

The ALFA-Pd is the module running on an embedded target. It offers the ability to run the state-of-art denoising methods in software, and DROR, DLIOR, and LIOR using dedicated hardware accelerators. It runs on top of an embedded Linux version, with a custom device tree, to enable faster communication with hardware accelerators. In this dissertation, two versions of ALFA-Pd are presented, both using two distinct memory management approaches, and aiming at solving different purposes.

The ALFA-Pd embedded module is responsible for applying denoising filters to a point cloud and, provides the resultant product through a ROS message in a dedicated ROS topic. As depicted in fig **??**, this module interfaces with ALFA-DVC using ROS, through an ethernet connection between a desktop, where ALFA-DVC runs, and the embedded system.

Depending on the version in use, the user can define the default filter parameters that will be by the hardware modules before compilation. At run time, the user can configure what denoising algorithm to use, with the option to select the hardware-accelerated ones. The user can also configure the parameters of the executing filters in run time, with the number of threads for the multithreading configuration included **??**. All data, inputs, and outputs are done through ROS messages, with ALFA-DVC offering a user-friendly way to interface with these ROS topics.

BRAM version

The BRAM version provides faster execution times, at cost of BRAM modules. Because the memory used in this version is very scarce, the version is target-specific, being difficult to port to other platforms. To execute the filtering of a point cloud is required that all points from the point cloud are previously stored before execution, due to one point depending on all the others to be validated as an inlier or outlier.

The ALFA-Pd BRAM version uses four block memory generators. The block memory generator, depicted in Fig. **??**, is an advanced memory constructor IP provided by Xilinx that generates area and performance-optimized memories using embedded block RAM resources in Xilinx FPGAs. This IP core has two independent ports that access the shared

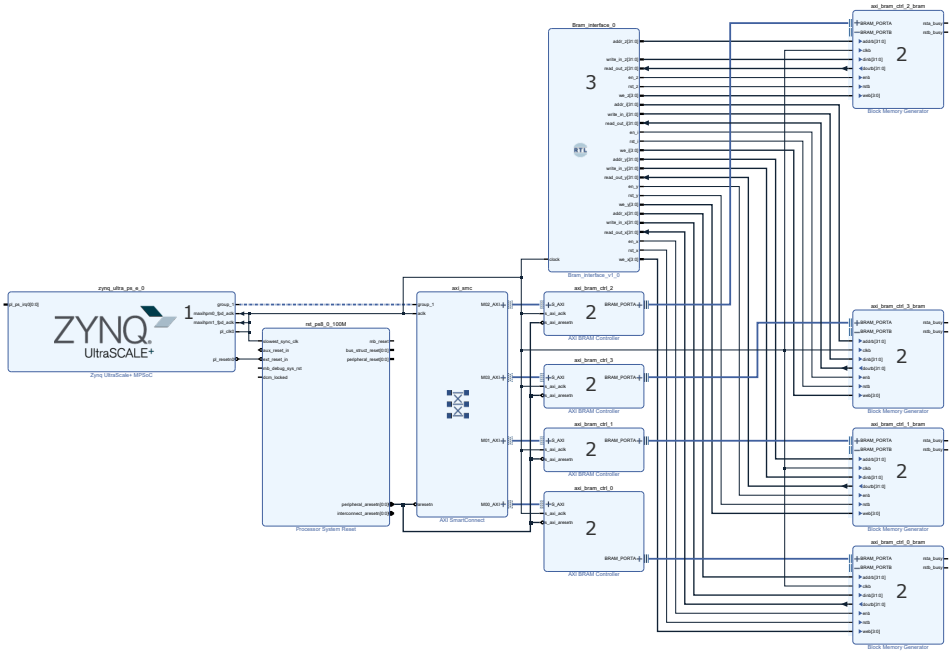


Figure 4.5: ALFA-Pd BRAM version.

memory space, port A and port B. In this version, port A is connected to an AXI BRAM controller, and port B to the BRAM memory interface developed in this dissertation. The AXI BRAM controller is an IP core designed as an AXI endpoint slave for integration with the AXI interconnect and the block memory generator. The core supports both single and burst transactions to the block RAM and is optimized for performance.

As described in **??**, ALFA-Pd is composed of three main modules, software, memory, and hardware modules. In Fig. **??**, the number one is the modules that correspond to the software modules **??**, in this case, it represents the processing system, where the embedded Linux runs. The number two are all the modules required to implement the memory module **??**. The number three are the hardware-accelerated weather denoising algorithms.

The memory modules (2) are composed of four block memory generators, one for each point parameter. By dividing the information in this scheme is possible to fetch two points from the point cloud per clock cycle. Each memory module has is independent address bus, but the BRAM interface links them all together. When the point cloud data is written to the BRAM memory in the software modules (1), the data is passed through AXI4 full to the AXI BRAM controller, which is responsible to write the points using port A of each BRAM.

The hardware weather denoising methods (3) are encapsulated by the memory interface, being the module that interfaces with all the external modules, like the BRAM modules. The memory interface is responsible for controlling all the B ports of the block

memory generators, having dedicated buses for each one. Because all the data required for processing a point cloud frame is in the block memories, the memory interface only connects to them.

Table 4.1

Paremeter	Default value
BRAM Bus size	32
BRAM 0 base address	0xA000_0000
BRAM 1 base address	0xA100_0000
BRAM 2 base address	0xA200_0000
BRAM 3 base address	0xA300_0000
Point Clusters number	16
Neighbor Finders number	2

The DDR memory interface has its parameters editable for the user. Because ALFA-Pd hardware accelerators are built in a modular way **??**, it's possible to edit all the crucial parameters and the hardware will be generated accordingly.

As shown in the tab. **??**, the user can edit the bus size of the BRAMS. This allows for different block memory generators configuration compatibility. In this version, the BRAM bus used by default is 32 bits, but in other platforms, it's possible to use block memory generators with buses greater than 32-bits. Using the chosen platform to deploy this project, it is possible to use a 16-bit configuration, enabling one point per clock. Increasing the bus size means that more points can be obtained per cycle, for example using a 64-bit wide bus, it is possible to fetch four points per cycle, increasing even further the performance. The block memory generators are accessible from the PS by the AXI communication. The block memory is mapped into the memory space. To increase the compatibility of the memory, there is the possibility to change the base address of each AXI BRAM controller. The base addresses are depicted in the tab. **??**.

The default configuration for the number of point clusters and neighbor finders is depicted in the tab. **??**. The number of neighbor finders is set to two by default, since by using a BRAM bus of 32-bits, the module fetches two points per cycle, enabling the possibility to do two comparisons for the clock cycle, one from each neighbor finder.

DDR version

The DDR version provides a more generic approach, at a cost of time performance. Because the memory used in this version is somewhat abundant, when compared with the BRAM, it is easier to port the version to other platforms. This implantation fetches the point cloud date from the DDR4 present in the platform, commonly used by the PS. To configure the hardware denoising filters, it was used AXI Lite to handle the communication between the PS and the PL.

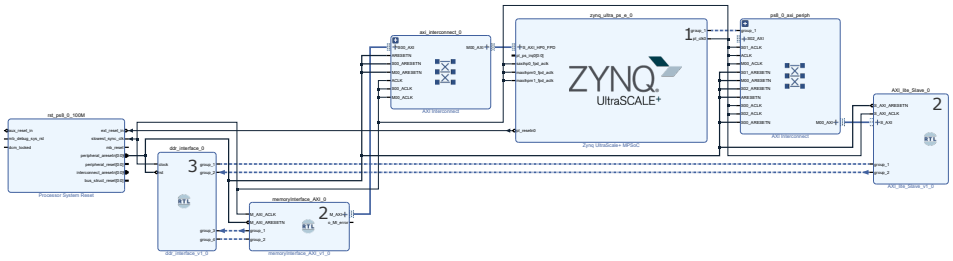


Figure 4.6: ALFA-Pd DDR version.

The DDR version uses AXI full to get all the related data to the point cloud. The AXI lite is used for signaling and configurations related to the processing of a point cloud frame. The AXI Full and Lite transactions are handled by two modules different modules developed for this version.

Similar to the BRAM version **??**, the DDR version also has is composed of three main modules, software, memory, and hardware modules **??**. The number 1 in Fig.**??** is where the PS is located, and where the software runs. In this version, the MPSoC is used as an AXI slave to give access to the DDR.

The memory modules (2), depicted in Fig.**??**, are composed of two elements, the "memoryInterface_AXI" and "AXI_lite_slave". The first one is used to handle the AXI Full transactions, as a master, to read and write in the DDR memory space. The module also offers a read burst capacity of 32, meaning that for each burst transaction the module outputs thirty-two points. The configuration of the memory interface is done through "AXI_lite_slave", as described in **??**. The AXI lite module handles communication in both directions, from the PS to the PL, and the PL to the PS.

The DDR interface (3), encapsulates all the hardware denoising accelerated methods. It receives the starting signal from the AXI Lite module and sends back the result through AXI Lite. The point clouds points are obtained in bursts of thirty-two through the DDR from the AXI Full module. The deleted points are removed from the DDR using the same module, but, because the outliers indexes are not sequential in the memory, the writing in the DDR is not done in bursts.

Table 4.2

Paremeter	Default value
AXI Burst size	32
AXI cluster cache multiplier	1
AXI feeder cache multiplier	1
Point Clusters number	30
Neighbor Finders number	1
DDR Base address	0x0F000000

The parameters of the DDR memory interface module are depicted in Tab. **??**. The interface allows changing the burst size, enabling compatibility for different burst sizes from the AXI module. If the number of point clusters is superior to the burst size, then the AXI cluster cache multiplier needs to be adjusted, to guarantee that the AXI cluster cache is always greater than the number of clusters. The interface also allows the same operation to the feeder cache, but, in this version, the number of neighbors finders is always set to zero.

Because the number of cycles needed to do an AXI burst transaction is always greater than the number of points obtained in the transaction, the number of neighbor finders modules is set to one. As explained in **??**, by setting the number of modules to one, the number of resources needed decreases, optimizing the version.

Filter configurations

The filter parameters are chosen by the user, but if no configuration is done, the hardware module will use the default values for each filter. The default configurations are set to a Velodyne Puck LiDAR sensor. This is the sensor is capable of outputting an average of around 17098 points per frame with 0.3 degrees of angular resolution. This was defined as default because the point clouds gathered in this dissertation using this sensor contain real weather noise.

The default filter parameters depend on the filter chosen to execute. The default filter is DIOR, being the one developed in this dissertation.

DROR only has tree parameters: neighbor threshold, defining the number of neighbors a point needs to have to be classified as an inlier, the multiplication parameter β that is

Table 4.3

Filter	Parameters	Default value
DROR	Neighbor treshold	6
	Multiplication parameter	16
	Minimun Search Radius	1
LIOR	Neighbor treshold	5
	Multiplication parameter	8
	Search radius	50
	Intensity threshold	5
DLIOR	Neighbor treshold	6
	Multiplication parameter	16
	Minimun Search Radius	1
	Intensity threshold	5

used to calculate the search radius using the eq. **??**, and the minimal search radius, to safeguard that the execution goes smoothly for points closer to the center.

LIOR has four parameters due to the use of intensity values to validate points. Because LIOR is based on a ROR filter, it uses a fixed radius, the parameter search radius. Because all points are multiplied by one hundred by the PS, to give a resolution of 1 centimeter, the base value of the search radius, fifty, Is equivalent to fifty centimeters. The intensity threshold is used to validate points if they have an intensity value superior to the threshold and is set to 5.

DLIOR is a combination of DROR and LIOR, so it uses parameters from both, the neighbor threshold, multiplication parameter, and the minimum search radius that are the same as the DROR ones. The intensity threshold used is equal to the one used in the LIOR filter.

4.3 ALFA-DVC integration

ALFA-DVC serves as an interface for ALFA-Pd to configure, visualize, and data transfers. Using ALFA-DVC the user can select the weather denoising filter in use by ALFA-Pd, visualize the input/output of ALFA-PD and it can be used to send/retrieve point clouds.

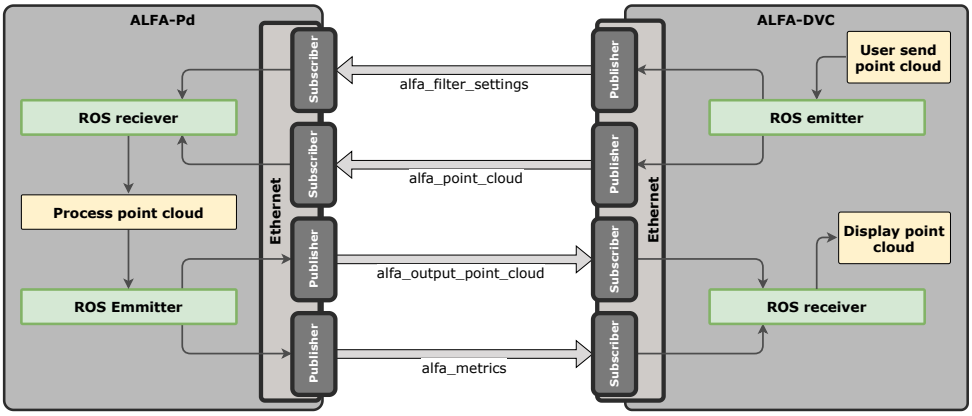


Figure 4.7: ALFA-Pd system architecture.

The ALFA-Pd module interfaces with the ALFA-DVC through a ROS environment. Each module is composed of 2 publishers and 2 subscribers. For each publisher in the ALFA-DVC tool, there is a subscriber attached. As depicted in Fig.**??**. The user decides when a point cloud is emitted. A point cloud can be sent frame by frame by manually clicking in a button, or when the tool receives and pre-processes one frame received by ROS. The ALFA-DVC tool serves as an intermediary for external ROS topics. The user can also configure the embedded node through the ALFA-DVC.

The point cloud is sent to the ALFA-Pd node, through a ROS topic named alfa_point_coud. When the embedded system receives the point cloud, an event is triggered starting the filtering process **??**. ALFA-Pd has a default configuration that is used if no configurations were sent.

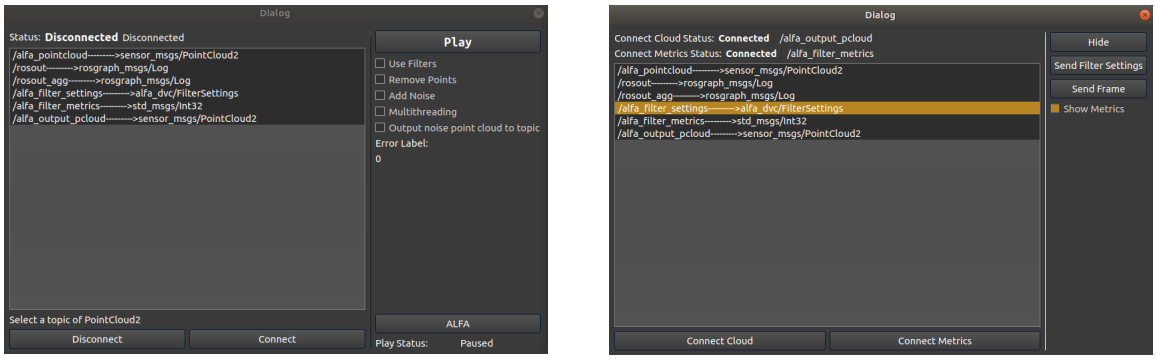
The filter configurations are sent through a costume ROS message type, with all parameters required, through a ROS topic named alfa_filter_settings. The tool is only an interface to interact with this topic, further application can be done to configure ALFA-Pd if they meet the message format. The configuration is only sent when the user clicks on a dedicated button for that action, and upon receiving the configurations, ALFA-Pd stores them to use on the next filtering operation, meaning when it receives the next point cloud.

When ALFA-Pd ends processing a point cloud frame the data is sent to two ROS topics. The point cloud data is sent to topic alfa_output_point_cloud in the format pointcloud2. The pointcloud2 format is the newly revised ROS point cloud message, that is used to represent arbitrary n-D (n-dimensional) data. Point values are of any primitive data types (int, float, double, etc), and the message can be specified as 'dense', with height and width values, giving the data a 2D structure.

ALFA-Pd also measures the time needed to process the frame, and at the end of processing, this metric is also sent to a ROS topic. The topic alfa_metrics holds the

processing time data, being read and interpreted by the ALFA-DVC tool.

The ALFA-DVC tool has two dedicated subscribers to read the topics where ALFA-Pd posts the data. These topics are selected and connected to by the user interface. When receiving a point cloud from the topic, an event is triggered that will display the point cloud. Upon receiving the metrics info, an independent event is triggered, where it will evaluate the difference between the point cloud sent and the one received, calculating all the metrics needed to evaluate filter performance, using the box system **??**. The processing frame time that is received by the topic, sent by ALFA-Pd, is displayed in the filter window **??**, alongside the new calculated metrics.



(a) A subfigure **(b)** A subfigure

Figure 4.8: A figure with two subfigures

ALFA-DVC serves as an interface for ALFA-Pd to configure, visualize, and data transfers. Using ALFA-DVC the user can select the weather denoising filter in use by ALFA-Pd, visualize the input/output of ALFA-PD and it can be used to send/retrieve point clouds.

ALFA -DVC can interface with the ROS environment through two different windows, depicted in Fig **??**. The ROS interface, depicted in **??**, allows the user to choose the workflow of the ROS environment. The user can connect/disconnect from a topic with messages of pointcloud2 format. After receiving a point cloud frame, the user can configure the next execution, is possible to filter the point cloud, using the ALFA-DVC built-in weather denoising filter, remove all points inside the noise removal boxes, inject noise to the point cloud using the noise boxes, use multithreading, and at last, the user can output the point cloud to a topic at which the ALFA-Pd is connected.

Two interface ALFA-DVC and ALFA-Pd, user can use the ALFA window, depicted in Fig. **??**. The user connects to the topic, in which ALFA-Pd is publishing the point cloud, to enable second point cloud visualization in ALFA-DVC. The user can also collect the filter metrics performance of ALFA-Pd, by connecting to the right topic. To configure ALFA-Pd, the user must click on the dedicated button, and the configuration in the filter window will be sent. If the user chooses to load a point cloud frame from memory, then it's possible to forward that frame to ALFA-Pd through the button in this window.

5. ALFA-Pd Implementation

ALFA-Pd is part of ALFA and is composed of two main modules, ALFA-DVC and ALFA-Pd node, ALFA-DVC stands for ALFA debugger visualizer and configurator, and it is a high-level cross-platform QT application that runs on a desktop system. It enables the real-time visualization of point clouds (up to two), allows the deployment and evaluation of software-based algorithms directly on top of a point cloud, provides debug and configuration of weather denoising methods, and configures the ALFA-Pd node that is running on an embedded system.

5.1 ALFA-Pd Architecture overview

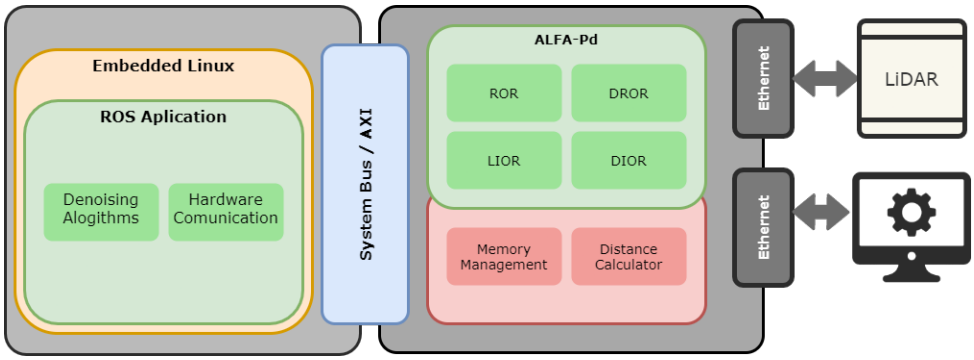


Figure 5.1: ALFA-Pd system architecture.

Fig. 5.1 depicts the ALFA-Pd framework that, following a hardware-software co-design approach, enables the fast deployment and evaluation of state-of-the-art weather denoising methods both in hardware and software. In the software layer, ALFA-Pd features a ROS environment running on top of a minimalist embedded Linux generated by yocto using an open embedded project. It provides different levels of abstraction for (other) high-level applications, such as the ALFA debug, real-time point cloud visualizer, system configurator (ALFA-DVC) tool, and ALFA-Pd ROS node. ALFA-Pd supports a collection of software-based denoising methods (assisted by the PCL library), core-libraries which

abstract and interfaces the weather denoising accelerators developed on hardware capable of achieving real-time constraints.

5.1.1 ALFA-Pd module overview

ALFA-Pd has two main modules, running in hardware and software. Regarding the hardware platform, the hardware is built on a Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC present in the ZCU104 Evaluation Kit (3.1.1).

On the PS side, ALFA-Pd is built on top of an embedded Linux operating system, being a ROS application, It receives both configurations and points cloud data through ROS messages, from a desktop running ALFA-DVC. The application also posts on a ROS topic the denoised point cloud. The application is cross-compiled by yocto and auto appended to the linux image at generation time. When a point cloud frame is received by the application, it applies the configuration previously received by the ALFA-DVC, and after processing, by hardware accelerators or software, it publishes on a ROS Topic enabling real-time visualization.

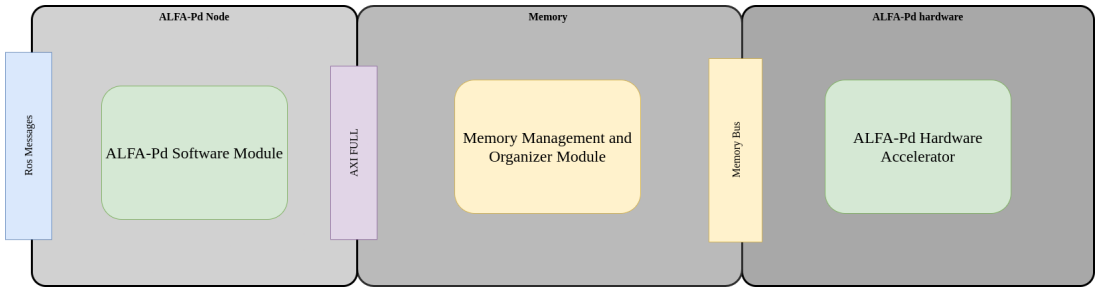


Figure 5.2: ALFA-Pd modules architecture.

Depicted in Fig.5.2 is the main module division overview that compose ALFA-Pd node. It is dived in three modules, software, memory and hardware.

The software module group is the first to execute, it receives information via ROS messages posted into a ROS topic, and if the configuration is set to use the hardware accelerators, it encodes the point cloud information and stores it in a pre-defined memory region accessible from both hardware and software modules.

The memory module is the one interfacing software modules and hardware modules. It communicates with the software by AXI4 full, by linking a specific address range to the mapped memory. Depending in the implementation, and every time something is written into that space, it is copied and stored in a fast and versatile memory named Block RAM (BRAM).BRAM is a type of random access memory embedded throughout an FPGA for data storage. The memory modules communicate with the hardware accelerators by a specific bus composed of the address, data, and write enable flags.

In this dissertation, two approaches were developed for the memory modules. One implementation is based on BRAM, and the other is based on DDR. Both implementations use AXI Full to communicate with the software layer. The interface between hardware modules and memory modules will vary depending on the chosen implementation.

If the BRAM memory implementation was chosen, the hardware modules communicate with the memory modules by a specific memory bus. The starting and ending of the hardware accelerator are signaled by the memory. The point cloud data is also stored in these memories and quickly accessed and edited by the hardware modules. Once the hardware modules finish processing a frame, is saved the result on a specified position in the memory, and this result is further read by the software modules to continue process flow.

If the DDR memory implementation was chosen, the hardware modules communicate with the memory modules by a specific memory bus. The signaling and configurations of the hardware accelerator are done through AXI Lite **??**. The point cloud data is stored in DDR4 memory and is accessed using transactions of AXI Full. Once the hardware modules finish processing a frame, the outliers are removed from the memory, and the result is further read by the software modules to continue process flow.

5.1.2 ALFA-Pd Software

The software modules are responsible for communication with outsider systems, processing, and denoising point clouds both in software and hardware. This modules also provides an abstraction layer for high-level applications that use point cloud data

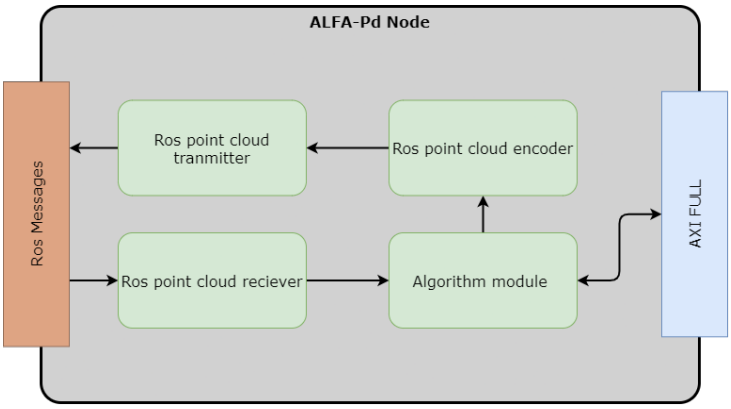


Figure 5.3: ALFA-Pd software modules architecture.

In Fig.5.19 is depicted a more detailed view of the software module architecture. The first module to execute is the ROS point cloud receiver. This module is running in an independent thread, waiting for a new point cloud to process. When a new point clouds is received, it's converted to the PCL format and a sginal is sent to the next module to

start the filtering process. This module is also responsible for receiving the configuration settings from the ALFA-DVC tool.

The Algorithm module is responsible for denoising the point cloud. It receives the data from the previous module, and depending on the configuration it executes one filter algorithm. Most algorithms are implemented using the PCL library, assisting with functionality like neighbor search. The algorithms implemented in this module are listed in the tab. 5.1 where is depicted all the PCL functions used in order to implement all the denoising filters. ROR, SOR, and VG are already implemented in the PCL library and, therefore, not implemented in this work. FCSOR [] LIOR[], DROR[] were implemented by analysis and interpretation of the algorithm description provided by the authors of those algorithms. DIOR is an algorithm developed in this work, fusing the advantages of LIOR and DROR, being almost as fast as LIOR, and providing a good true positive ratio like DROR.

Table 5.1: PCL software modules used by the ALFA framework and the ALFA-DVC tool.

Algorithm	***PCL module(s)***	***Description***
Voxel Grid	***VoxelGrid()***	Assembles a local 3D grid over a given PointCloud, and downsamples + filters the data
ROR	***RadiusOutlierRemoval()***	Filters points in a cloud based on the number of neighbors they have
SOR	***StatisticalOutlierRemoval()***	Uses point neighborhood statistics to filter outlier data
DROR	***kdtree.radiusSearch()***	Obtains the number of neighbors inside a dynamic radius *R1*
FCSOR	***StatisticalOutlierRemoval()***, ***VoxelGrid()***	The two PCL modules were combined to implement the FCSOR algorithm
LIOR	***kdtree.radiusSearch()***	Obtains the number of neighbors inside a fixed radius *R1*
DIOR	***kdtree.radiusSearch()***	Obtains the number of neighbors inside a dynamic radius *R1*

5.1.2.1 Multi-threading configuration.

On the software module, LIOR, DROR and DIOR use a multi-threading configuration in order to improve performance relative to the base algorithm description, giving it a fair comparison when evaluating the software algorithms versus the hardware-accelerator ones. The fig. **??** depicts how the multi-threading was deployed. The goal of this configuration is to divide the workload between different threads, running different sections of the point cloud in parallel.

As depicted in fig **??** the workload is divided in N threads, being N the number of cores present in the platform CPU. In this cause, it was deployed in a Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit that has a quad-core ARM® Cortex™-A53, 3.1.1. The

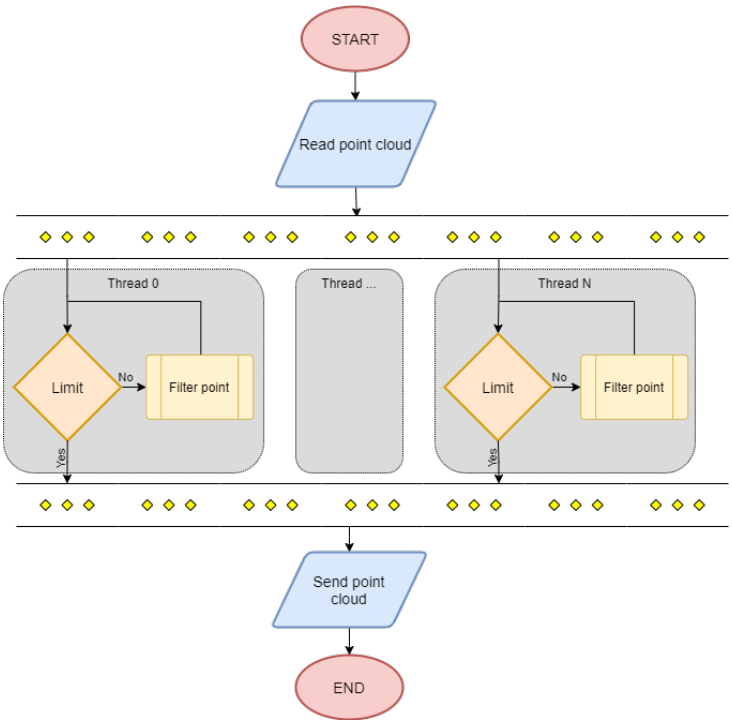


Figure 5.4: ALFA-Pd Multi-threading configuration.

thread will execute inside a for loop, in which the start point is given by the Eq.6.1, and the end point is given by the Eq. 6.2.

$$StartPoint = \frac{PointCloudSize}{NumberOfThreads} * ThreadNumber \tag{5.1}$$

$$EndPoint = \frac{PointCloudSize}{NumberOfThreads} * (ThreadNumber + 1) \tag{5.2}$$

The main thread launches all the threads simulations and then stays in a waiting state until all the worker threads finish. Once all the threads finish, it will wrap all the results in the same point cloud and send them to the ROS point cloud encoder.

The filter point method varies depending on the previously chosen algorithm. Depending on the user-selected algorithm, the worker thread will execute the point cloud filtering in blocks. Each multithread supported filter has its worker, implemented based on the filter description. DIOR being the novel approach proposed in this work, is also supported by the multithreading configuration.

The working principle of DROR is to filter points based on their distance to the sensor. This concept is suitable for LiDARs in autonomous driving applications because of sparsity of point clouds. A sparse point cloud is the result of how LiDAR sensors do the scan process, Because most sensors are rotor based, the far the object is, less dense it will be , due to the sensor angular resolution. By increasing the sensors range, the distance between shooting angles will also increase. Due to this, traditional outlier removal-based

Algorithm 1 DROR filter point Pseudocode

Require:
P : Point cloud cluster
n_{min} : Minimum number of neighbors
SR_{min} : Minimum search radius.
$NeighborSearch$: PCL module that calculates the nearest neighbors

```
1: for p ∈ P do
2:     if r_p < SR_min then
3:         R1 ← SR_min
4:     else
5:         R1 ← SR_p
6:     end if
7:     n ← NeighborSearch(p, R1)
8:     if n > n_min then
9:         Inliers ← p
10:    else
11:        Outliers ← p
12:    end if
13: end for
```

filters don't perform well, and DROR came to combat these downsides, introducing a dynamic search radius on a ROR based approach. This filter was deployed using the Alg. **??**.

The DROR filter implementation requires some parameters previously defined and the point cloud that will be filtered. It requires a minimum search radius, because when points are very close to the sensor, they will have a very small search radius, which implies that will be very hard to find neighbors in such a small search radius. In order to prevent this situation, if the calculated distance between the point and the sensor is bellow this threshold, the search radius will be set the SRmin. The second constant is the minimum of neighbors, this sets the minimum of close points that a certain point requires to be classified as an inlier. To simplify this implementation, it was used a PCL library module, that does the neighbor search in the point cloud.

$$SR_p = \beta \times (r_p \times \alpha) \tag{5.3}$$

$$r_p = \sqrt{x_p^2 + y_p^2} \tag{5.4}$$

The workflow of DROR is depicted in Alg. **??**. For each point present in the point cloud DROR checks if the distance to the sensor is bellowed the search radius threshold,

if it is then the minimum search radius is set to R1, the search radius, if not, R1 is set by using Eq. **??**. The next step, is to obtain the number of neighbors that the point has, using the PCL module, if the number of neighbors is above the threshold, the point is classified as an inlier, if not, it's classified as an outlier.

Algorithm 2 LIOR filter point Pseudocode

Require:
P : Point cloud cluster
n_{min} : Minimum number of neighbors
I_{thr} : Minimum generic intensity threshold
SR : Search radius.
$NeighborSearch$: PCL module that calculates the nearest neighbors

```
1: for p ∈ P do
2:     if I_p > I_thr then
3:         Inliers ← p
4:     else
5:         n ← NeighborSearch(p, SR)
6:         if n > n_min then
7:             Inliers ← p
8:         else
9:             Outliers ← p
10:        end if
11:    end if
12: end for
```

LIOR filter was designed to attack some cons of the DROR filter, trying to achieve the best performance time possible. Instead of having a dynamic search radius like DROR, LIOR has a fixed one. Because the calculation of the radius impact significantly the performance, LIOR has the innovative concept of using the reflected intensity, to avoid heavy computing processing at every point of the point cloud.

The working principle of LIOR is based on the theory that noise points, outliers, have less intensity reflected than inlier points. With this concept, LIOR is able to classify most of the point cloud as an inlier with one simple comparison, because every point that has an intensity above a pre-defined threshold will automatically be classified as an inlier.

The filter point implementation requirements, on the software modules, is depicted in Alg **??**. It requires the point cluster P, the point cloud block corresponding to the thread it is executing, the minimum of points required to classify a point as an inlier n_{min}, and the intensity threshold I_{thr}, upon will classify point as an inlier. This threshold will be defined based on the used sensor specifications, due to different standards between manufacturers. LIOR uses a fixed search radius, so this needs to be set prior to execution,

in this algorithm the value is set using SR. Similar to DROR, the software implementation of this algorithm uses the NeighborSearch module of the PCL library (5.1).

For every point in the cluster, LIOR first checks if the point is an inlier by analyzing the point intensity, if it is above the defined threshold, then the point is classified as an inlier, but if not, the point will go through a second step, in which, it's applied a ROR type filter, checking the number of points inside a fixed radius. If the number of points inside is above the threshold then the point is an inlier, if not, it's an outlier.

Algorithm 3 DIOR filter point Pseudocode

Require:
P : Original point cluster
n_{min} : Minimum number of neighbors
I_{thr} : Minimum generic intensity threshold
SR_{min} : Minimum search radius.
$NeighborSearch$: PCL module that calculates the nearest neighbors

```
1: for p ∈ P do
2:     if I_p > I_thr then
3:         Inliers ← p
4:     else
5:         if r_p < SR_min then
6:             R1 ← SR_min
7:         else
8:             R1 ← SR_p
9:         end if
10:        n ← NeighborSearch(p, R1)
11:        if n > n_min then
12:            Inliers ← p
13:        else
14:            Outliers ← p
15:        end if
16:    end if
17: end for
```

Being DIOR a combination of DROR and LIOR, it uses the same execution requirements as its ancestors. It requires the point cluster P set to the execution thread that it will run on, the minimum of neighbors n_{min} to validate points as an inlier, the generic intensity threshold I_{thr} equal to the one used in LIOR configuration, the minimum search radius SR_{min} used previously on DROR configuration. DIOR also uses the same PCL modules as its ancestors.

According to the author, DROR can achieve a higher accuracy number because of its dynamic search radius, adapting to combat the sparsity in automotive point clouds,

but, the additional computational power required to calculating the radius slows down the filter, furthermore, when filtering distance point to the sensor, the PCL module, need a lot more computation power for navigating K-D tree in order to find the neighbors, slowing even more the filter. By adding a previous step, verifying the point intensity, helps the filter performance, thus, avoiding unnecessary processing.

The workflow of the implemented software is depicted in Alg **??**. For every point present in the point cluster, its checked the point intensity I_p. If the intensity is above the threshold then the point is classified as an inlier and no further processing is required for this point, but if the points fail on this verification, then it's calculated the search radius $R1$ for that point, exactly the same way as DROR, then it's done the neighbor search, resorting to the PCL module, to verify if the point has the sufficient number of neighbors n to verify as an inlier.

5.1.2.2 Hardware communication

When processing a point cloud frame, the user can select to use hardware-assisted denoising modules instead of software filters. In this work, it was developed two different versions of the memory interface, using BRAM or DDR, which have different methods of configuration and accessing in the software counterpart.

Algorithm 4 Software communication in BRAM implementation

```
1: Send point cloud size to BRAM X
2: Send filter settings to BRAM I
3: for p ∈ P do
4:     convert and compress X,Y,Z,I
5:     if P_index%4 == 0 then                ▷ 64 bits of information ready
6:         Send points data to BRAM
7:     end if
8: end for
9: Send start signal to BRAM Y
10: Wait hardware to finish on BRAM Z
11: for p ∈ BRAM do
12:     Convert and decompress points
13:     Inliers ← p
14: end for
```

Wen ALFA-pd is configured to use the BRAM implementation, the communication with hardware is depicted in Alg. **??**. The memory design will be analyzed in the next section, being out of the scope of this topic, but in short, the memory is divided into four BRAM blocks, one for each point parameter. The first position of every BRAM block is used for configurations or for signaling between the PS and the PL.

When a frame is received for processing, and the configuration dictates that will be used the Hardware-accelerated modules, the PS saves in the first position of the BRAM 0, containing all X points the point cloud size, the number of points the PL needs to process. In the next step, the PS sends the filter settings, selecting between DROR, LIOR, and DIOR what filter to use. After the first configuration step it's needed to convert every point p in the point cloud$P$, this means concatenating for points together and storing it in a 64-bit variable.

The architecture of the PS allows transfers of 64-bits. Each time four points are concatenated, the PS sends them to their respective position in the BRAM block, this means sending 64-bits of X coordinates, 64-bits of Y coordinates, 64-bits of Z coordinates and 64-bits of intensities values, due to the separation of the information in the BRAM.

After converting and concatenating all the points in the point cloud, the PS sends the start signal for the PL to start processing the frame. After signaling the PS stays waiting for the PL to signal back its end of processing. Wen the PL finishes processing, the PS reads the filtered point cloud in the BRAMs. After reading the entire point cloud, the data is sent to the next software module, similar to the execution of the software modules.

Algorithm 5 Software communication in DDR implementation

```
1: Compress and convert filter configuration
2: for p ∈ P do
3:     convert and compress X,Y,Z,I
4:     Send points data to DDR memory region
5: end for
6: Send point cloud size
7: Send frame id
8: Send configurations
9: Send start signal
10: Wait hardware to finish
11: for p ∈ DDR do
12:     Convert and decompress point
13:     Inliers ← p
14: end for
```

The DDR implementation was made to offer a more generic approach, because not all systems have the amount of BRAM available to the PL as the platform that was used. This implementation implies slowers memory access, but it offers versatility to porting for other platforms.

When ALFA-Pd receives a frame to process, and the configuration is set to execute using hardware accelerators, the implemented software compress and concatenate all the configurations needed to execute the filter. This includes the following parameters: filter

selector, intensity threshold, multiplication parameter, neighbor threshold, and search radius. If LIOR is the filter chosen, the search radius is used in the neighbor search, if not, the parameter search radius is used for the minimum search radius. After processing the configurations, the PS will compress and concatenate an entire point into a 64-bit variable and store it on a specified memory region. In order to have a shared memory space between the PS and the PL, some configuration on the device tree where required, in which it was allocated enough space to fit the output of a top-of-the-shelf LiDAR sensor. After storing all the point cloud points, the PS sends the execution parameters through AXI Lite to the PL, being these parameters the point cloud size, the frame id, to prevent over the execution of frames and to identify the execution frame, the filter configuration, and the start signal.

When the PL signals the finish, the PS module will then read the point cloud stored in the reserved zone of the DDR and decompress the points to the PCL format enabling them to be sent through a ROS topic for real-time visualization on the ALFA-DVC tool.

5.1.3 ALFA-Pd Memory

The ALFA-Pd memory module is the one interfacing the software module running on the PS and the hardware module running on the PL. Its also responsible for storing all the information to be accessed both from the PL and the PS. This information includes point cloud data, configuration data, and signaling.

Two distinct implementations of this module where maid, one using BRAM, and another using DDR4, the BRAM one is a more specific implementation, targeting the platform chosen for this project, taking advantage of all the resources available, however, a more generic implementation was also made, to enabling easier ports for other platforms with fewer resources. These two implementation only impact the memory module of ALFA-Pd, being the core of software and hardware modules barely modified.

BRAM memory module

The BRAM module implementation aims to increase performance. BRAM can provide the output with one clock delay. Thus, the PL can get data from memory, enabling point process at each clock. This implementation uses 32 data buses, meaning that each memory position holds 32 bits of data. Each point parameter requires 16 bits of data, so it's possible to store two points in each BRAM position. Storing the point cloud in BRAM requires an extra step before filtering to transfer the data, once each point depends on the remaining ones.

The first position of the BRAMs is reserved for configurations as depicted in fig. **??** . The X BRAM, the one containing the x coordinates of all points, holds the point cloud

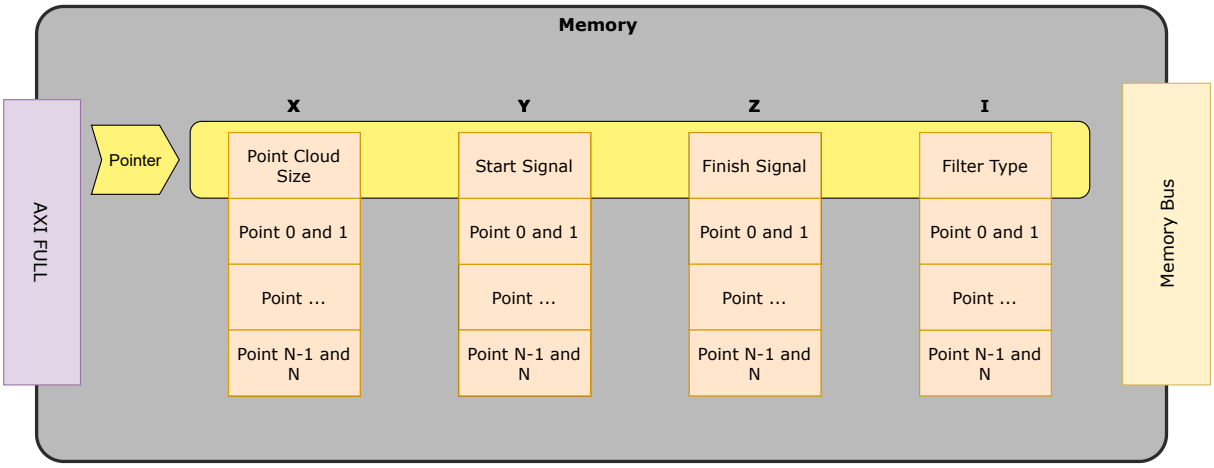


Figure 5.5: ALFA-Pd software modules architecture.

size. The point cloud size is necessary for the PL to know the number of points it needs to process. In the Y BRAM, stored the start signal. The PL is constantly reading this position to know when to process a frame. The Z BRAM contains the finish signal. When the hardware completes the filtering, it sends the finish signal to the software through the Z BRAM. The I BRAM stores the filter selector. ALFA-Pd supports DROR, DIOR, and LIOR, being possible to change the filter running upon execution time, the selector of the filter running is the one stored in the I BRAM.

The BRAM module supports true dual-port, thus, enabling read and write operation at the same time. In this implementation, one port is connected to the PS through AXI full, enabling the ALFA-Pd software modules to write and read on the BRAM. The other port is connected to the memory interface, an ALFA-Pd hardware module, through a custom memory bus, as depicted in fig. **??**. Using this feature enables for when the PS writes in the BRAM and at the same time, the PL reading what the PS is writing, and the opposite also applies. The dual-port feature also simplifies signal handling due to the fact of both PS and PL have read and write permissions, allowing for example, when the PS receives the finish signal clear it, and for the PL to clear the start signal received by the PS.

The ALFA-Pd BRAM memory module has a common pointer through all four memory modules, maintaining synchronism between them. Furthermore, having a common pointer and 128 bits worth of data each clock means that this memory module is capable of feeding 2 points each cycle.

The BRAM module stores 128 bits of data, providing storage capacity to hold two points per address position. Because the memory is divided into 4 BRAM modules, each module holds information of two points inside the parameter that it's holding. As depicted in fig **??**, the first 16 bits store the information of the first point, and the next 16 bits

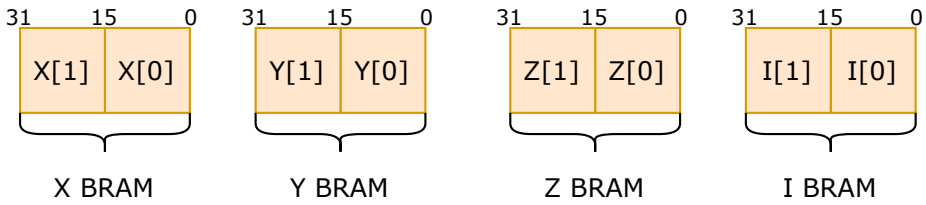


Figure 5.6: ALFA-Pd software modules architecture.

have the information of the next point. Every point stored in the less significative bits has a pair index, for example, when the pointer is 1, the first 16 bits of memory holds the information of point 0 and the most significative bits holds point 1, but when the pointer is 2, the first 16 bits stores the point 2, and the less significative the point 3. This information is important, to the memory interface of the ALFA-Pd hardware module to know in which address the points are stored.

DDR memory module

The DDR implementation aims to increase generality and compatibility with other platforms. Contrasting with the BRAM implementation, which uses a very scarce resource, and in other to provide compatibility with all LiDAR sensors it was required to allocate almost every BRAM in the platform, the DDR uses a more abundant memory, but at a cost of time performance. Because DDR is closer to the CPU than the PL fabric, the time per memory access is higher. Using DDR also withdraws the need of storing all the points in a specific memory, because the PL will access the memory domain of the PS. The DDR memory uses 64-bit wide data buses, so each position of the memory will old a point, 16 bits for each parameter.

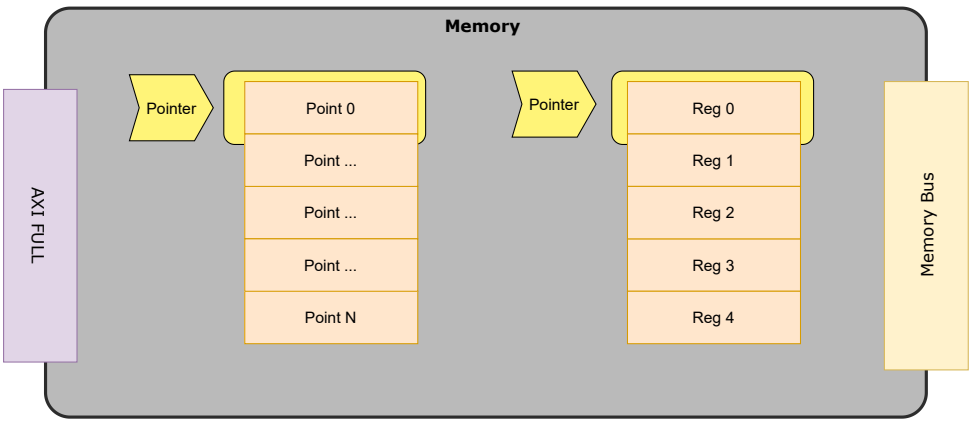


Figure 5.7: ALFA-Pd software modules architecture.

The DDR olds only the point information, each memory position holds on the entire point, having a single pointer controlling the read and write position. The PS writes

directly to this memory, but the memory module is implemented on the PL side, so in order to access the DDR the PL uses the PS as an AXI full slave, communicating through AXI full. To signaling and configure the ALFA-Pd hardware module used an AXI lite module with four registers as depicted in **??**. This module interfaces with the ALFA-Pd hardware module through a custom bus, giving access to each register individually and the interface to communicate through AXI full.

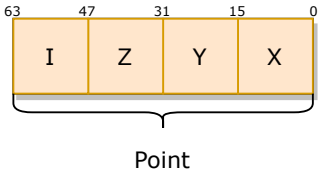


Figure 5.8: ALFA-Pd software modules architecture.

A point is composed of four parameters, the 3D coordinates x,y,z, and i the reflectivity intensity of the point. Each parameter requires 16 bits of memory, making a total of 64 bits of memory for the entire point. The AXI was configurated to use a 64 bits barrament size, meaning that each transaction outputs 64-bit data. The point constitution is depicted in fig **??**, going from the less significative 16 bits that represent the parameter X of a point, to the intensity value on the most significative 16 bits. Using 16 bits to represent x means that the max distance on the x axle that the ALFA-Pd can process is approximately 327 meters. This information is also true for the y and z axles. The max range of modern LiDAR sensors varies between 100 to 200 meters, so with this configuration, ALFA-Pd is capable of processing all the modern sensors.

The configurations and signaling are made through AXI lite. All the registers are connected to the hardware module. By using AXI lite, the complexity of the system is reduced. The AXI lite module has 4 registers, containing all the data required for the execution of a frame, each register holds 32 bits of data, furthermore, to efficiently use this register, the configurations are concatenated to fit in this 32-bit registers.

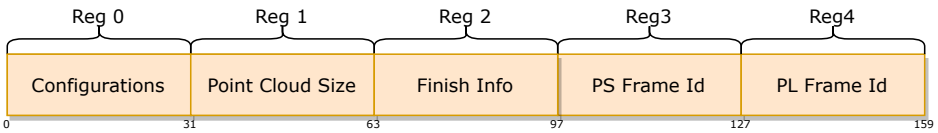


Figure 5.9: ALFA-Pd software modules architecture.

The parameters required for the execution of a frame are depicted in fig. The AXI lite register 0 is the one holding all the configurations and signaling. This register is written by the software modules, and read by the hardware ones.

Register 1 stores the point cloud size of the executing frame written by the software module and read by the hardware module. This site is used by the ALFA-Pd hardware modules to identify the finish of the point cloud.

Register 2 is only written by the hardware modules and read by the software modules. The finish info contains the number of outliers, written by the hardware modules at the finish of the processing.

To maintain synchronism between the PL and the PS registers 3 and 4 store the current frame id of each module. When the PS frame id differs from the PL frame id, the PL starts processing the new frame. When the PS and the PL frame id are the same, the PS knows that can send a new frame to the PL.

Figure 5.10: ALFA-Pd software modules architecture.

The DDR memory module implementations allow for filter parameter change at run time. All the parameters are concatenated into the AXI lite register 0 as depicted in fig. The first two bits are allocated to the start signal. The next 4 bits are reserved for the filter selector, the variable responsible for changing what filter is executing. From bit 6 to bit 9 the intensity threshold is stored, a common filter parameter of DLIOR and LIOR. Between bit 9 and 13, the search radius is allocated, when the filter selector dictates that the filter execution is the LIOR, then this register is used as the search radius for the neighbor search. When the filter selector commands that the filter executing is DROR or DLIOR, the search radius register is used for the minimal search radius parameter of those filters. From bits 13 to 21, the min neighbor threshold is stored, this parameter is used by all the hardware filters implemented. The neighbor threshold parameter is used to classify points as an inlier. If a point has more neighbors than this threshold, the point is classified as an inlier.

The multiplication parameter is fitted between bit 21 and 31, this is a tuning parameter used to compensate for the point spacing increase resulting from surfaces that are not perpendicular to the LiDAR beams. The multiparameter translates to beta in eq. The constants β and α usually are below 1, meaning they are a decimal variable, but ALFA-Pd doesn't support operation with floating points. So instead of applying eq?? the application eq was applied. If the angular resolution of the sensor is 0.1 degrees, instead of multiplying the range of the point for 0.1, the point distance is divided by 10. The concept of converting the multiplication for divisions allows the calculation of the search radius of DROR and DLIOR filters without the need for a floating-point module.

5.1.4 ALFA-Pd Hardware

One of the key points of the ALFA framework is the ability to deploy customized hardware accelerators on the FPGA fabric. This opens the possibility to improve the performance of the supported software version of weather denoising algorithms.

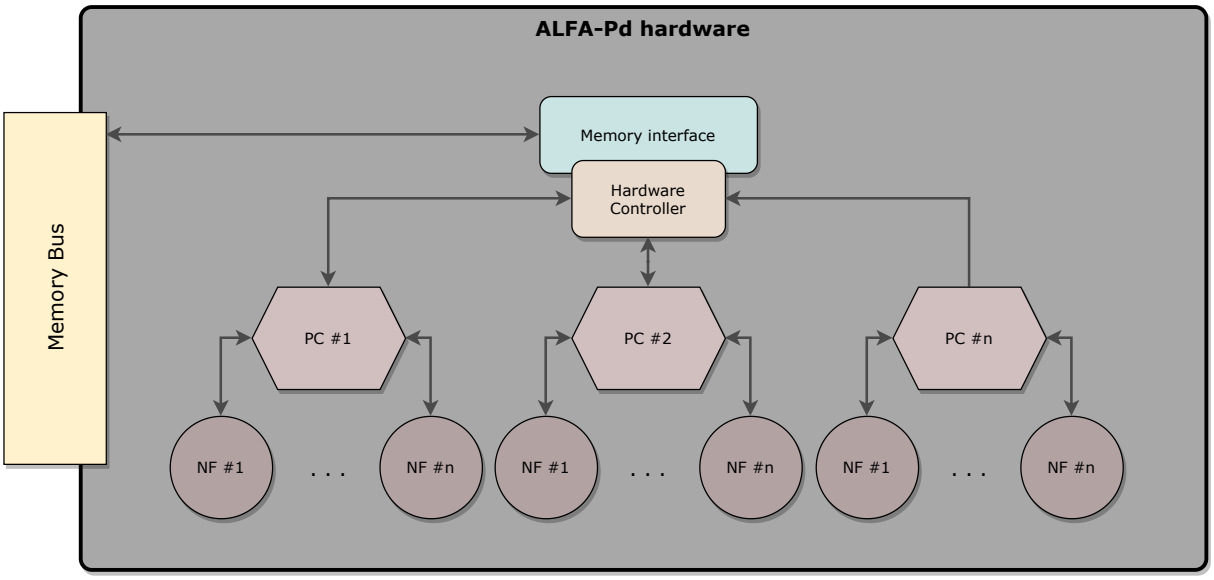


Figure 5.11: ALFA-Pd software modules architecture.

Their generic hardware implementation, depicted in Fig. , is composed of four main components: (1) the hardware controller; (2) a memory interface; (3) several Point Cluster (PC) blocks; and (4) Neighbor Finder (NF) units. Each PC block deployment requires at least 2 NFs, requiring each of them 1 Distance Calculator module available from the ALFA core accelerators.

The PC and NF blocks are used to parallelize the algorithm's execution in finding neighbors to classify a given point as inlier or outlier, decreasing the required time to process a point cloud frame. The number of PC and NF dictates the performance of the filter but at a cost of resources. The limit of modules deployed is defined by the resources on the target platform. In an ideal world, with unlimited resources, the ideal number of PC and NF will be the size of the point cloud, processing the whole point cloud in one clock cycle.

BRAM memory interface

The memory interface module is responsible for interfacing the BRAM memory where the points are stored and the hardware controller. This module controls the execution flow of all the other hardware modules. As described in **??**, the memory interface uses a custom bus to communicate with the BRAM. A transaction with BRAM takes one

cycle, meaning every clock the modules debilitates one output, simplifying the control mechanism of this module.

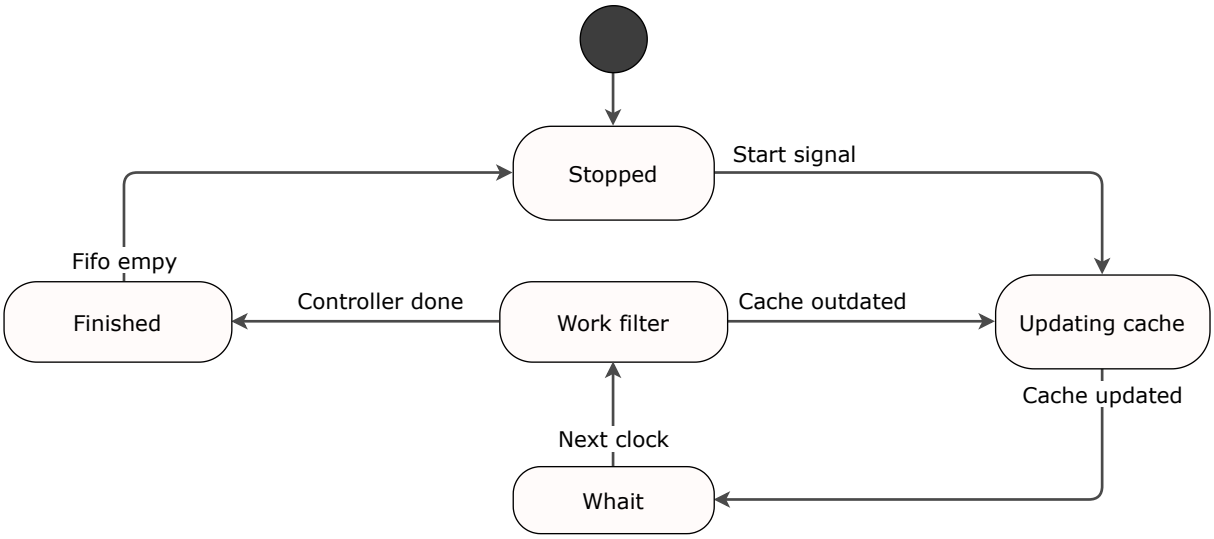


Figure 5.12: ALFA-Pd software modules architecture.

The BRAM memory interface implements a state machine depicted in fig **??**. The state machine has five states. Upon starting, the module is in a stopped state, waiting for the configurations and the start signal. The PS sends the start signal, starting the execution of the module. The first state that executing after the start signal is the update cache.

The memory interface has 2 caches, a point cluster cache, and a neighbor finder cache. Each cache is divided into 4, one for each point parameter, using the same approach described in REF MEMORY MODULE. The cache size is defined by the number of point clusters, and for the neighbor finder cache, the size is set by the number of neighbor finders. The point cluster cache is used by the hardware controller to distribute points when the point cluster finishes. The neighbor finder cache, is used by the neighbor finder cache, to calculate the distance between the point in cache and the one storer by the point cluster, and depending on the result, classify the point as an inlier or an outlier.

As described in REF MEMORY MODULE, each BRAM address holds 2 points, so the number of cycles needed to fill the caches is given by REF FORMULA.

$$Cycles = \frac{CacheSize}{2} \tag{5.5}$$

When the point cluster cache is full, a cycle of delay is required, to prepare the hardware controller. The work filter state starts the hardware controller, and in parallel, fetches the neighbor finder cache. When the point cluster cache is outdated, the state

execution changes to the update cache. This process is repeated until the hardware controller signals its finish.

Once the hardware controller finishes, the executing state changes to the finish one. In the last state, the memory interface reads all the outlier indexes from a FIFO. The hardware modules when it finds an outlier saves its index in a FIFO, to later be used by the interface to remove all the outlier. The finished state removes all the points, using the indexes stored in the FIFO. When the FIFO is empty, the hardware module signals the finish to the PS using the Z BRAM as described in REF MEMORY MODULE. After finishing, the module goes to the stopped state again, waiting for the next frame to process.

DDR memory interface

The Memory Interface module is responsible for interfacing the DDR memory where the points are stored and the hardware controller. This module controls the execution flow of all the other hardware modules.

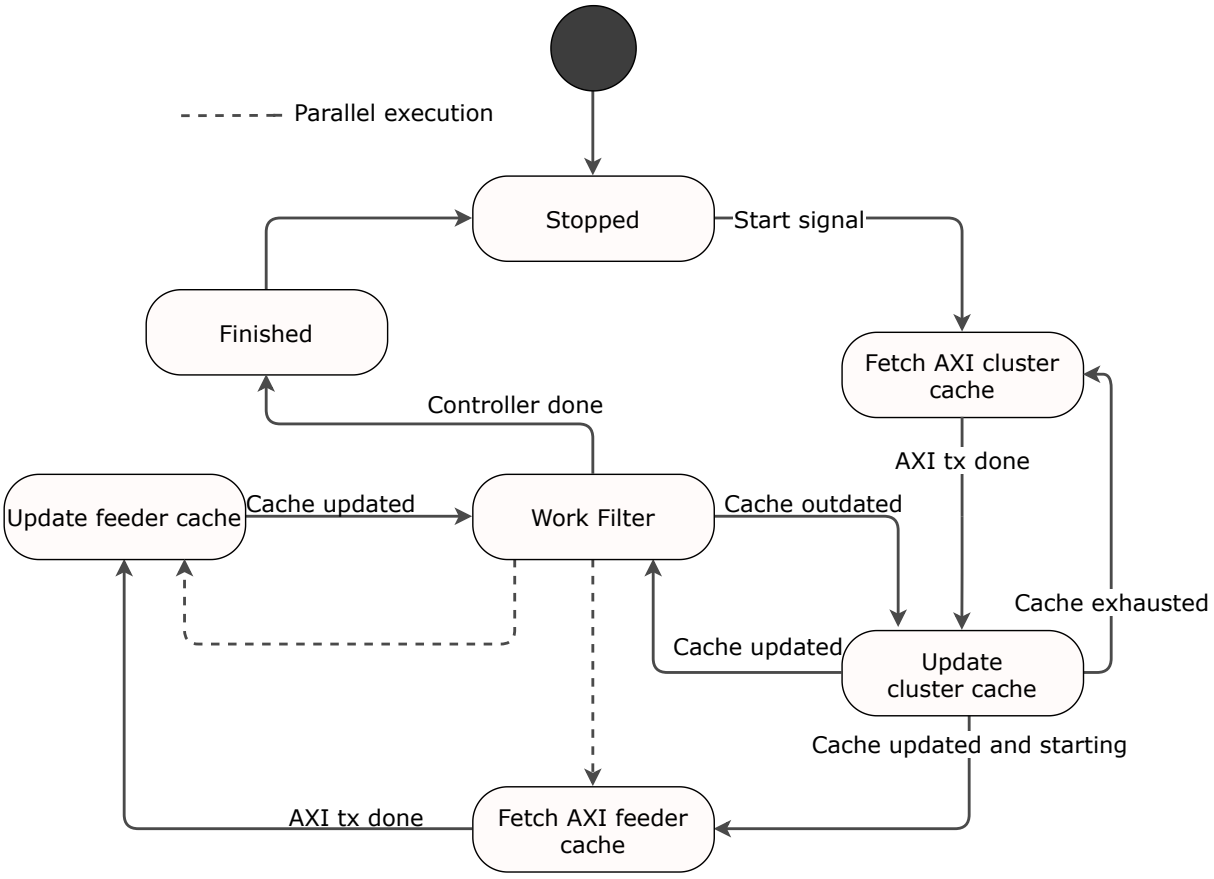


Figure 5.13: ALFA-Pd software modules architecture.

In fig. is depicted the state machine used by the memory interface. In the stoped state, the interface is waiting for a start signal. If it receives a start signal, and the PS

frame id differs from the PL frame id then the interface starts the execution by going to the second state, fetching AXI cluster cache.

DDR memory interface has a total of 4 caches, 2 for the point clusters and 2 to feed the neighbor finder. Equal to the BRAM memory interface, each cache is divided into 4, one for each parameter because the hardware controller requires that configuration.

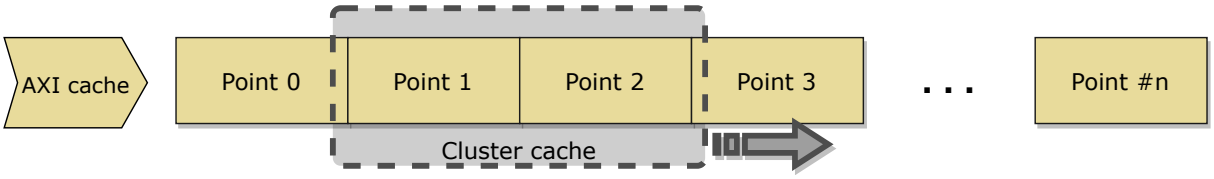


Figure 5.14: ALFA-Pd software modules architecture.

The 2 cache design is depicted in fig .**??**. The AXI cache holds the values obtained through the transaction. The AXI transaction is set to read a burst of 32-bits from the DDR, to store all the points obtained, a cache with the same length of the burst is used. At the start of a new AXI operation, all the values on the output of the modules are reset, so there is a need to store the point in a specific cache. When the number of point clusters is superior to the number of points obtained per AXI burst, the cache size is increased by multiple numbers of the burst size, eg: burst size is 32 bits, and the number of point clusters is 42, so the size of the cache will be 64-bits.i

The cluster cache size is defined by the size of point clusters in use and is used is the same as the cache described in the BRAM implementation. The size of this cache is always smaller than the AXI cache because, as depicted in fig **??**, the cluster cache acts like a sliding window of the bigger cache. The feeder cache design is the same as the cluster, also working as a sliding window of the AXI feeder cache. The operating of allocating the points of the AXI caches to the smaller caches takes one clock cycle, speeding up the execution, until the cache is exhausted, meaning that there are not enough points to score in the cluster cache.

When the cluster cache is updated, the state machine can go to two different states, depending on the condition. If it's a start, meaning the first time each state is executing, the module goes to the fetch AXI feeder cache. If it's not a start, there is no need to fetch the AXI feeder cache, because the one stored can still be relevant and not outdated, so the next state executing is the work filter.

The fetch AXI feeder cache works exactly in the same way as the fetch AXI cluster cache, but instead of using the cluster point index, it uses the feeder point index. Instead of saving the data received through the AXI transaction into the AXI cluster cache, the fetch AXI feeder cache holds the values in the AXI feeder cache, as the name suggests. When the AXI transaction ends, the state machine executes the update feeder cache state.

The update feeder cache state execution flow is the same as the update the cluster cache, using a sliding window to go through all the points inside the AXI feeder cache. The size of the feeder cache in this implementation is one, this also means that, in this implementation, the size of neighbor finders is one per each point cluster module. Upon completing the update procedure, the next state is the work filter.

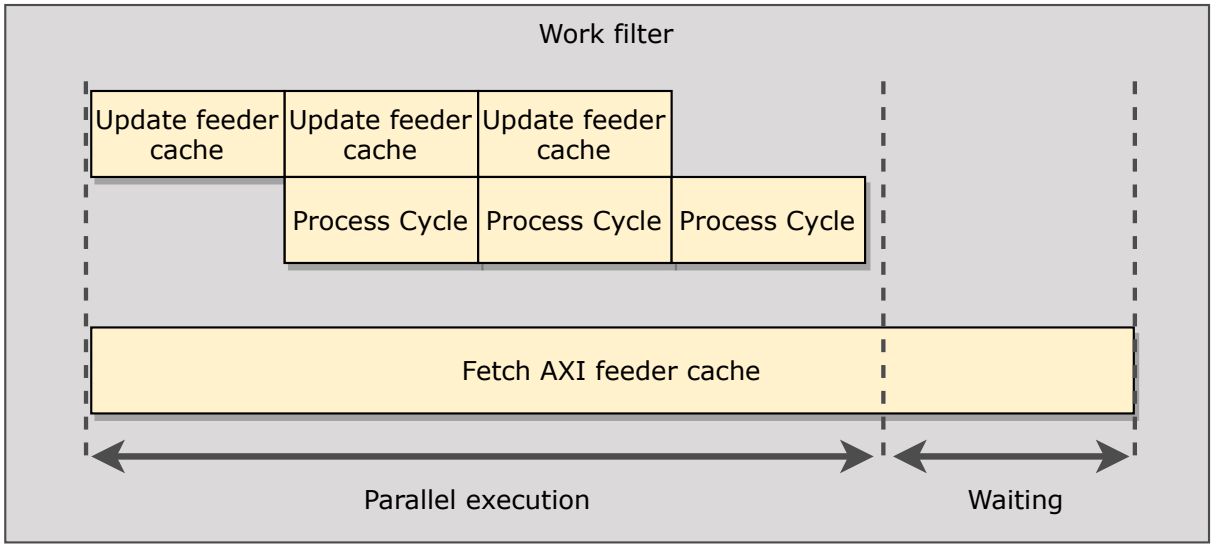


Figure 5.15: ALFA-Pd software modules architecture.

The work filter state controls the hardware controller module. The hardware controller only needs to execute when every cache is updated and ready to use and to do that a pause and start mechanism is implemented. To increase the overall throughput of points validated, a parallel state execution was used.

The parallel state execution is depicted in fig **??**. When the work filter state starts executing, meaning the first clock cycle that executes, the state fetch AXI feeder cache and the state update feeder cache is also executed. Upon the first execution of the update feeder cache, the hardware controller receives permission to execute the validation of points. The process cycle is always one clock behind the update feeder cache one. Together they work in a somewhat of a pipeline, comparing one point per cycle. If the feeder cache gets exhausted before the end of the AXI transaction occurring in the fetch AXI feeder cache state, then, the execution of the update feeder cache stops, staying in a wait state until the transaction ends. When the AXI transaction ends, the AXI feeder cache gets updated, and the execution of the work filter state restarts. Two events can happen while executing the flow depicted in fig **??**, if the AXI transaction ends before the cache is exhausted, the AXI bus is blocked and the fetch AXI feeder cache state stays in hold until all its cache is used, preventing missing point comparisons.

If while executing the work filter state, a signal of cache outdated can come from the hardware controller, and if it does come, the update feeder cache stops, and consequently

the process cycle also stops. Because an AXI transaction is always active, the work filter state waits for the finish of the transaction, and then the state executes changes to the updated cluster one.

When the hardware controller finishes, it signals the memory interface. At the memory interface module, when it receives the finish signal, the state machine executes the finished state. In the last state, every outlier stored in the FIFO of the hardware module is removed from the DDR memory. The points are removed using the AXI full module, setting the point in the point index of the FIFO to zero, deleting it. When the FIFO is empty, the executing state changes to the stopped, waiting for a new frame to process.

Hardware controller

Hardware Controller is the main block of the denoising accelerator and it is responsible for: send points to the PCs for comparison, checking and controlling the output of all PCs, tagging and storing the points classified as outliers from each PC. This module is controlled by the memory interface, requiring the cluster cache and the neighbor finder cache to work. The hardware controller doesn't change depending on the memory implementation, because it gives an abstraction layer for the top modules. It has a start and stop mechanism to maintain synchronism with the caches that are updated in top modules.

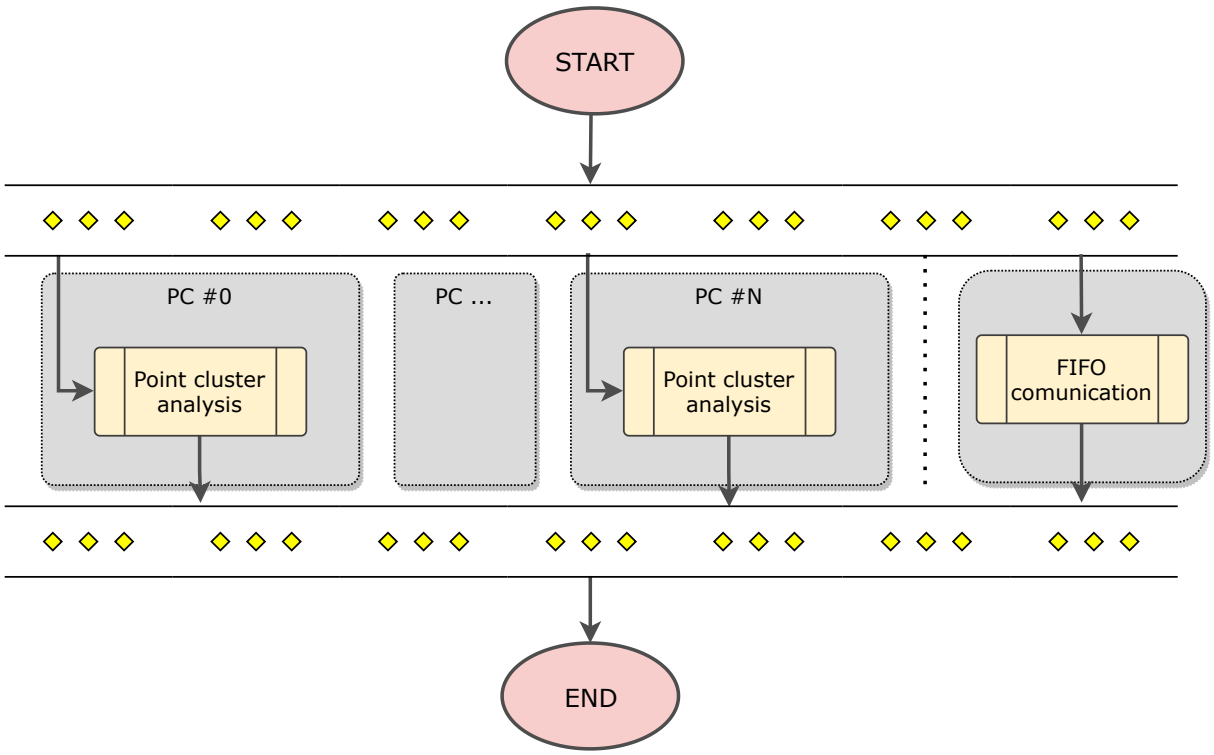


Figure 5.16: ALFA-Pd software modules architecture.

The working flow of the hardware module is depicted in fig **??**. Every clock cycle that the module is on play state, analyzes the results of every point cluster output, which means if a point is an outlier or an inlier. The hardware controller also has a built-in FIFO. The FIFO holds the point index of every outlier. In parallel with the analyzes of the points clusters, the hardware module also stores one point index into the FIFO. Because it's only possible to write on point per clock to the FIFO, the hardware controller also has a buffer where the point indexes are firstly stored, and then moved to the FIFO. The size of the FIFO buffer is set to double the number of point clusters. eg: the number of point clusters is 2, so the buffer size will be 4. By setting the buffer size to double, guarantees that every outlier is saved in the FIFO. Communication with the FIFO is always active. Every cycle that the buffer has more than one outlier, the hardware module store the outlier in the FIFO, not depending on the start and stop system.

Algorithm 6 Point cluster output analyzes

```
1: if inlier or outlier then
2:     if outlier then
3:         fifo_buffer[fifo_size] ← cluster_point_index
4:         fifo_size ← fifo_size + 1
5:     end if
6:     cluster_point_index ← point_index
7:     point_index ← point_index + 1
8:     cluster_point ← cluster_cache[finished_counter]
9:     finished_counter ← finished_counter + 1
10: end if
```

The point cluster analyzes workflow is depicted in the Alg. **??**. For every point cluster in the implementation, there will be a point clutter analyzer logic deployed. The point cluster analysis is responsible for analyzing the output of the point cluster associated, and in case of the result from the point cluster being an outlier, store the cluster point index in the buffer.

If the result from a point cluster defines a point as an inlier or an outlier, then that point cluster requires to be reloaded and restarted. Furthermore, if the validated point is classified as an outlier then, the point cluster point index is saved into the FIFO buffer. To the FIFO communication submodule store in the actual FIFO, it needs to know how many points it has on the buffer, so the fifo_size variable is updated. Because the point cluster ended validating the point, all the information, such as the point itself and the indexes, is outdated. The process of reloading the point cluster information is presented in the Alg. **??**, first the index of the point that the point cluster will validate is passed, then the point index of the hardware controller is updated to point to the next point. The module uses

the cluster cache from the memory interface to update the point cluster validating point. The position of the cache that is used depends on how many point clusters are finished before the one analyzing. After updating the point, the finish_counter is updated to the next point cluster analyzer to know how many of them finished before. Note that all the point cluster analyzers are executed in the same clock cycle.

Point cluster

A point cluster module is used to store the current point under validation, i.e., to be classified as an outlier or inlier. The classification is based on the number of neighbor points that lie inside the search radius ***R1***. If the number of neighbors found is below the defined neighbor threshold, the point under validation is classified as an outlier, otherwise, the point is tagged as an inlier.

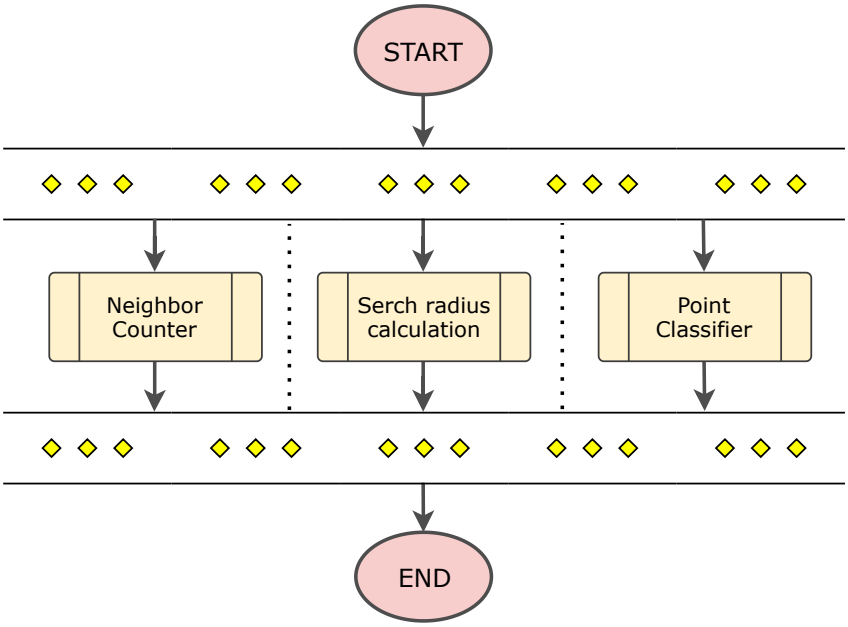


Figure 5.17: ALFA-Pd software modules architecture.

The point cluster module is divided into three submodules as depicted **??**. The module is responsible for counting the number of neighbors of a point, calculating the search radius of the point, and classifying a point as an inlier or outlier based on the neighbors found. The neighbor counter requires the information provided by the search radius calculation, and to classify a point the point classifier submodule requires the number of neighbors provided by the neighbor counter. At the start of execution, the neighbor counter submodule is delayed one clock, thus it requires the information from the search radius calculation submodule. Furthermore, the point classifier is delayed two cycles from the start and one cycle from the first execution of the neighbor counter, because it uses the number of neighbors given by the neighbor counter submodule.

Algorithm 7 Neighbor counter

```
1: for neighbor_index = 1, 2, 3...N do
2:     if distance[neighbor_index] <= R1 then
3:         neighbor_counter ← neighbor_counter + 1
4:     end if
5: end for
```

The neighbor counter submodule calculates the number of neighbors found per cycle as depicted in the Alg.**??**. All the neighbor finder modules have the output connected to an array of the point cluster module. the neighbor counter module goes through the array of distances and compares each distance to check if it's below the search radius. If the distance of the neighbor finder is below the search radius then, it is a neighbor, incrementing the number_counter, if not, the distance calculated is discarded.

Algorithm 8 Search radius calculation

```
1: if F_s != 1 then
2:     if r_p < SR_min then
3:         R1 ← SR_min
4:     else
5:         R1 ← SR_p
6:     end if
7: else
8:     R1 ← SR_LIOR
9: end if
```

The search radius calculation module calculates the radius $R1$ **??** based on the filter selected. F_s is the filter selector, if the filter selected is one, then the point cluster will execute the LIOR algorithm. Because LIOR has a fixed search radius, the search radius $R1$ is defined by a user-selected radius. If the F_s is different than one, the filter selected for execution is either DROR or the DLIOR. Both filters have a dynamic search radius, calculated in the same way. If the point distance to the sensor r_p is below the predefined threshold SR_{min}, then the value of the search radius $R1$ is set to the SR_{min}, similar to the software counterpart **??**. On normal conditions, $R1$ is defined by the eq. **??**.

$$SR_p = \beta/(r_p/\alpha) \tag{5.6}$$

The AFLA-Pd hardware modules don't support mathematical operations using decimal numbers, due to the lack of a floating-point module. The original search radius equation **??**, takes the point distance r_p and multiplies it to numbers between zero and one, which is the same as diving the number, eg if α is 0.1, instead of multiplying r_p by

0.1, the hardware modules divide r_p by 10. The concept of converting the multiplication to division is also applied to α and β, the multiplying parameter, as shown in the eq. **??**. The parameter in eq. **??** are defined before hardware synthesis when using the BRAM implementation. When using the DDR implementation, the parameters are set at run time.

Algorithm 9 Point classifier

```
1: if F_s >= 1 & I_p > I_thr then
2:     inlier ← 1
3:     outlier ← 0
4: else
5:     if neighbor_found >= neighbor_threshold then
6:         inlier ← 1
7:         outlier ← 0
8:     else
9:         inlier ← 0
10:        if p_comp > P_size then
11:            outlier ← 1
12:        else
13:            outlier ← 0
14:        end if
15:    end if
16:    p_comp ← p_comp + NF
17: end if
```

The point classifier submodule is responsible for tagging points as inlier and outliers. It uses the information from the other submodules to analyze and classify a point. As depicted in the Alg. **??**, first the submodule checks if the filter selected is the DLIOR or the LIOR, that requires the intensity of the point, if the intensity is above the threshold I_{thr}, then the point is classified as an inlier. The classification of inliers when the point reflectivity is above the threshold requires one clock cycle to complete, being the fastest validation case of the system, if every point were validated as an inlier through this if statement, the point cloud validation would be the point cloud size, not accounting for the time lost in AXI transactions.

If the point fails to validate by the intensity, either because is below the threshold or because the filter selected doesn't use intensity, then it's required to have at least *neighbor_threshold* neighbors to be classified as an inlier. If the point doesn't have the neighbors required, then it's needed to check if the point is an outlier. If the number of points compared p_{comp} is above the point cloud size P_{size} then the point was already compared with all the other points within the point cloud frame, meaning that the point is an outlier. If the point was not compared to the whole point cloud, then the point cluster

keeps counting the number of comparations done per cycle executing. The number of comparisons per cycle is given by the number of neighbor finder modules, with more modules, the faster will be the classification of outliers. If the number of neighbor finder modules is equal to the size of the point cloud, then an outlier is classified in one clock cycle, in the other hand, if the point cluster only has one neighbor finder module, then, to find an outlier it would require the size of the point cloud in cycles.

Neighbor finder

The neighbor finder module is used by the point cluster to calculate the distance between two given points. The output is then used by the point cluster to classify the point under validation considering the neighbor threshold and the search radius ***R1***.

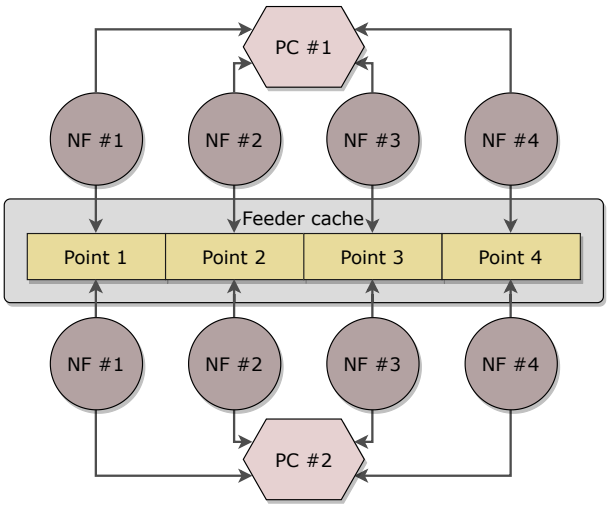


Figure 5.18: ALFA-Pd software modules architecture.

Each point cluster can have multiple neighbor finder modules, accelerating the finding of an outlier process. The modules are grouped for each point cluster and are completely independent of the other neighbor finder modules of other point clusters. The neighbor finder module uses a feeder cache to obtain the points that the point under evaluation will be compared to. As depicted in Fig. **??**, different neighbors finders use the same point for comparison. The feeder cache feeds all the neighbor finders with points each cycle, always moving forward in the point cloud. The cache is independent of all the neighbor finder and point clusters and acts as a circular buffer for the point cloud, ensuring that all points will pass through the neighbors finder.

The neighbor finder module calculates the distance between two points in one clock cycle, being the core calculation module of the ALFA-Pd hardware accelerator. Depicted in the Alg. **??** are the method used to calculate the distance between two points, the point under evaluation stored in the point cluster, and one point given by the feeder cache, in

Algorithm 10 Distance calculation

1: $Vector \leftarrow p_1 - p_2$
2: **if** $Vector[15] == 1$ **then**
3: $\quad Vector = -Vector$
4: **end if**
5: $distance \leftarrow P_{dist}$

the memory interface module. First, the neighbor finder module calculates the 3D vector containing the two points. In order to do the square of the vector, the point needs to be positive, because if not, when the number is negative, the complement of 2 is used in the square root calculation, giving completely wrong results, due to the overflow of the calculation. To cope with the negative vector problem, the neighbor finder module, checks the most significant bit, the one that indicates if a number is negative or not, and depending on the result, the absolute value is used.

$$P_{dist} = \sqrt{x_p^2 + y_p^2 + z_p^2} \tag{5.7}$$

The distance between the two points is obtained through the eq. **??**. The squares are implemented doing multiplication by themselves. In this implementation, the vivado operation '*' was used to do the multiplication, meaning that it will require some scarce resources like DSP modules. The square root was implemented with the resource of a prebuilt arithmetic module, allowing the calculation without floating points.

5.2 ALFA-DVC tool

The ALFA-DVC tool is a high-level and cross-platform QT application that runs on a desktop system. It enables the real-time visualization of point clouds, allows the deployment and evaluation of software-based algorithms directly on top of a point cloud, and provides the debug and configuration of weather denoising methods.

When the data points are collected in ideal weather conditions, the ALFA-DVC can also emulate and generate different weather noise applied to specific regions of the point cloud through a box system. Such a box system allows to label and remove noise points, as well as to analyze the algorithm through several performance and accuracy metrics.

Point cloud data can also be loaded into the ALFA-DVC tool using Point Cloud Data (.pcd) and Polygon File Format (.ply) files stored in the file system or through the available ROS interface. This interface can provide point clouds directly received from a LiDAR sensor connected to the ALFA hardware platform or from a ROS topic running in

a local ROS node. The tool also enables saving screenshots of the point cloud currently being played, as well as publishing the noised/denoised point cloud on a new ROS topic.

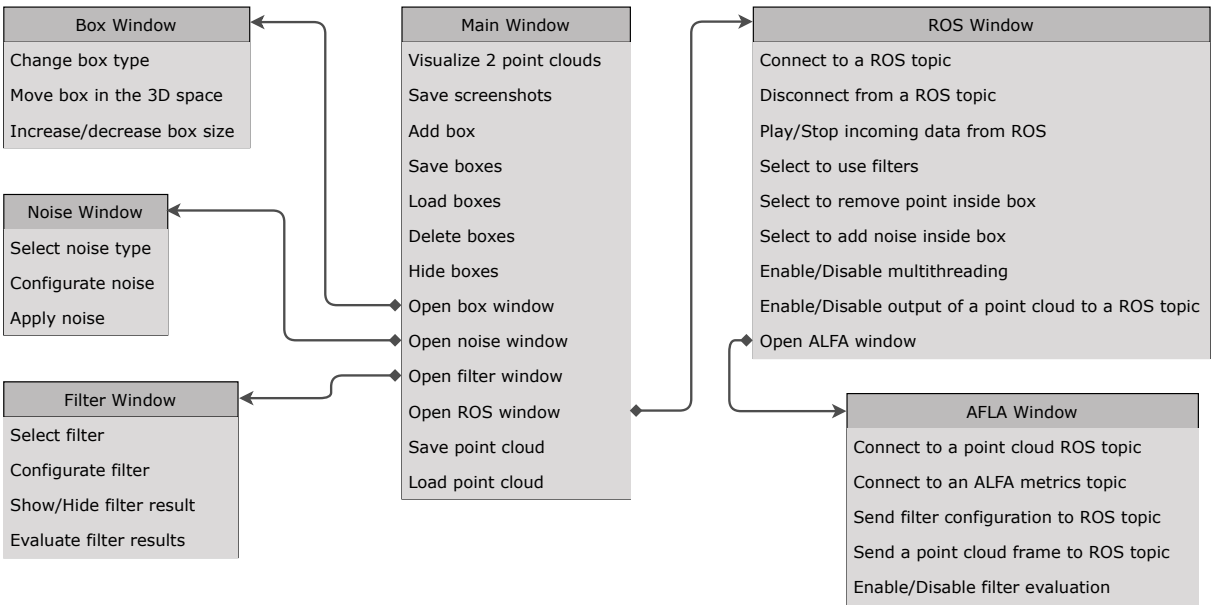


Figure 5.19: ALFA-Pd software modules architecture.

The ALFA- DVC is composed of six windows, each one dedicated to one functionality of the tool, providing support for testing the state-of-art weather denoising algorithms, and interface with the ALFA-Pd. To accelerate the filter configuration processes, first, all the parameters are tested with the tools provided by ALFA-DVC, and then those parameters are passed to the embedded system.

The main window allows the visualization of up to two point clouds in real-time, this way it's possible to see what the embedded system will receive, the input point cloud, and what the AFLA-Pd filters, the output point cloud. The tool allows saving screenshots of the point cloud visualized, enabling a later analysis of the results.

In order to classify and benchmark all the weather denoising filters, a method of obtaining metrics to evaluate the performance was implemented. ALFA-DVC uses a 3D box system that allows reading how many points that lie inside the box, deleting all the points inside a box, and injecting noise to the point cloud by adding random points inside the box. The box system allows to store all the box configurations, it also allows the user to load a previously configurated set of boxes, allows to delete boxes and, to help point cloud visualization, hide the box system. The user can edit all the box settings, like a position in the 3D space and size, through a dedicated window that opens when the user double clicks on it.

Through the box system, the user can also inject noise into the point cloud. On the main window, the user has the option to open the noise window, where all the noise

settings are accessible. Because AFLA-DVC offers the ability to use weather denoising filters within the tool, a special window is also accessible through the main window. The ROS communication is handled in a dedicated window, accessible through a dedicated window in the main window. The ALFA-DVC tool allows to save and load point clouds. These files can be saved in two standard formats, Point Cloud Data and Polygon File Format, enabling the point clouds processed using ALFA to be read by other point cloud visualizers. It also enables the ability to restore a point cloud frame, by giving the user to load a point cloud frame from the filesystem.

The box window is where all the configurations to a given box are. Through this window, it's possible to increase or decrease the box size, by giving control of all four coordinates of the boxes to the user. This window also enables 3D movements, by applying vectors to the 3D shape.

There are three types of boxes in ALFA-DVC, label, noise, and removal boxes. The label boxes are used by the filter evaluating systems, to label points inside them as noise, to later calculate the filter metrics. The noise boxes exist to deploy simulated noise into the point cloud in real-time. The noise is later used to test all the denoising filters when using point clouds that don't have real noise. The removal boxes are used to identify areas to remove points. It's a common technique to evaluate filters to ignore the points that represent the vehicle, being these points removed from the point cloud. The removal boxes can be used by the user upon receiving data from a ROS topic instead of a static frame.

The noise window is responsible for configuring and adding artificial noise to point clouds. The window supports two types of noise, which are user-defined. The true random noise generator adds a user-defined number of random points in the target area, defined through the box system. The Gaussian noise is the other type of noise that can be injected into point clouds by the ALFA-DVC tool. The user can define the standard deviation and the number of points that will be injected into the point cloud.

The filter window empowers ALFA-DVC with the ability to execute weather denoising filters on the fly. The implementation of these filters is the same as the ones running on the embedded system in the ALFA-Pd software node **??**. The filter window also allows the configuration of the filter running in the ALFA-Pd, both the hardware-accelerated ones and the software ones. The user first selects the filter that will be executed, and upon the selection, the default configurations will appear, giving the user the ability to edit them. These configurations can be sent through a ROS message by the ALFA window **??** or can be executed in the point cloud that is currently selected in the tool. After executing the filters, this window gives the possibility to show the results and hide them. Upon a filter execution, the tool also executes the filter metric evaluations. When the ALFA-DVC tool

receives a point cloud from the ALFA-Pd, the execution of the filter evaluation is also done in this window. The window also displays all the metrics obtained by the evaluation.

The ROS windows offer the user the ability to interface with ROS in a user-friendly way. This window displays all the ROS topics that the user can connect to. When a user clicks to connect to a specific topic, the window presents the possibility to disconnect from that topic. The interface has a built-in stop and play system, enabling the user to stop the receiving module from retrieving new point clouds from the ROS topic. The user also decides what happens when the tool receives a new ROS point cloud. The user can choose to use filter the point cloud at arriving time. The filter and the configuration used, are selected in the filter window. The user can select to remove all the points inside the boxes that are tagged as noise boxes upon receiving a frame. It is also possible to select the option to add noise inside the boxes tagged as noise boxes, giving the possibility to generate artificial noise upon receiving a frame.

Because DIOR, LIOR, and DROR can be executed in a multithreading configuration, as explained in the section **??**, the user can enable the use of that feature inside the ROS window. The point cloud generated after all the selected settings can be outputted to a ROS topic, in the pointcloud2 **??** format, to a specific topic, where the ALFA-Pd will get the point clouds to process.

To connect and configure ALFA-Pd, the ROS window can open the ALFA window, where all the configurations, related to the second point cloud that can be visualized in the main window, are displayed.

The ALFA window is the one with all the connections and configuration needed to interface with the ALFA-Pd node. It connects to a ROS topic to receive the point cloud from the embedded system, enabling two point clouds visualization in parallel and in real-time. Because ALFA-Pd has built-in filter evaluators, that measure the time needed to process each frame, the ALFA window allows the connection for the retrieval of those metrics. The metrics received by this window are displayed in the filter window, thus these metrics are only related to the filter performance.

The user can prefer to send a single frame instead of forwarding multiple point clouds frames, so the AFLA window has a button that when pressed sends the point cloud in the visualizer, the main window, to the predefined ROS topic in which the ALFA-Pd is connected and waiting for point clouds. To configure the ALFA-Pd module, the user first needs to configure the filter settings in the filter window, and then in the ALFA window, send it through a custom ROS topic. The user is able to enable or disable the filter evaluation that is done upon receiving a frame from the topic connected in this window.

6. Evaluation and Results

This chapter is presented an extended benchmark and comparative analysis between state-of-art, including DIOR. First, all algorithms will be evaluated running in software, and then the performance using the hardware accelerator will be benchmarked. The algorithms that performed the best in the software analysis were the ones implemented in hardware.

$$PR = Input\ Points - Output\ Points \tag{6.1}$$

$$TP = \frac{Filtered\ Noise\ Points\ In\ Noise\ Areas}{Total\ Labeled\ Noise\ Points} \tag{6.2}$$

$$FP = \frac{Filtered\ Noise\ Points\ In\ Nown\ Noise\ Areas}{Total\ Labeled\ Non\ Noise\ Points} \tag{6.3}$$

$$FN = \frac{Unfiltered\ Noise\ Points\ In\ Noise\ Areas}{Total\ Labeled\ Noise\ Points} \tag{6.4}$$

The denoising algorithms were evaluated using the following metrics: (i) points removed (PR), Equation 6.1; (ii) true positives (TP), Equation 6.2; (iii) false positives (FP), Equation 6.3; (iv) false negatives (FN), Equation 6.4; (v) frame processing time (FPT); and (vi) frames per second (FPS). This metrics ensures that the output of all algorithms is analyzed and compared in all the performance metrics, and not only the time-related ones.

All the evaluation tests were executed using the full point cloud from a public dataset that was retrieved with a Velodyne Puck (VLP-16) in a real-world scenario during an intense snowstorm [63]. In this point cloud, the sensor produces frames containing, on average, around 17098 points per frame. The same frames were used both in hardware and software tests, to maintain a fair comparison between them.

The benchmarking performed in this section was done using ALFA-DVC to read and classify the point cloud. All the performance metrics were obtained through the box system that ALFA-DVC offers. For visualization purposes, the before and after of each

algorithm is also shown, through ALFA-DVC. Because the box system takes a lot of visual space, the box system is hidden in the following screenshots.

6.1 Evaluation of Software-based Denoising Algorithms

The evaluation of the software-based denoising algorithms was made using ALFA-Pd embedded node, to execute the filters, and ALFA-DVC to gather the filter results and calculate the performance metrics. Because the goal is to depict the performance of state-of-art algorithms in an embedded environment, the execution of the filters wasn't done in a desktop environment.

Regarding the DROR, LIOR, and DIOR algorithms, they were both executed in the original format and in a multi-thread configuration (four threads). The execution of these filters in a multi-thread configuration is done to give a fair comparison with the hardware accelerators counterparts. By accelerating the software-based methods, we are ensuring that the best possible version of these algorithms is executed.

6.1.1 System configuration

All software runs on top of an embedded Linux (4.19.0-xilinx-v2019.2) with the PCL library (version 1.8.1-r0) and a ROS environment (Ros1 melodic distribution). The software is supported by the ALFA hardware platform, previously described in Section **??**, with the CPU running at a clock speed of 1.2 GHz and the DDR4 memory running at 535 MHz. Because the CPU, Arm Cortex-A53, has four cores the multithread configurations are set to use four threads, to optimize the throughput of points filtered per thread. The FPGA was not used during the evaluation of the software-based algorithms.

The FTP metric is only the time that a given algorithm requires to provide an output. The time lost in ROS transaction was not accounted, to reduce the number of possible variations during tests.

To enable replication of the evaluation presented in this work, the denoising algorithms were evaluated with the filter parameters depicted in Table **??**. Because it is not shown how to configure the filter parameter in the state-of-art, these parameters were obtained using trial and error. The execution performance depends heavily on the configurations, so all tests were performed using these parameters. The hardware parameters are the most similar to these ones to limit the impact of configuration in this evaluation.

Table 6.1

Algorithm	Parameter	Value
SOR	Number of k neighbors	4
	Standard deviation multiplier	0.9
ROR	Searching radius	0.5
	Min. number of neighbors	5
DROR	Min. searching radius	0.1
	Min. number of neighbors	30
	Angular resolution	0.3
	Radius multiplier	0.9
LIOR	Searching radius	0.5
	Min. number of neighbors	5
	Intensity threshold	4
DIOR	Min. searching radius	0.1
	Angular resolution	0.3
	Radius multiplier	0.9
	Min. number of neighbors	30
	Intensity threshold	4

6.1.2 Voxel Grid Filter

The voxel grid filter removes noise from a point cloud by downsampling the point cloud data. Furthermore, because of the low complexity, VG-based filters are the most time-efficient ones.

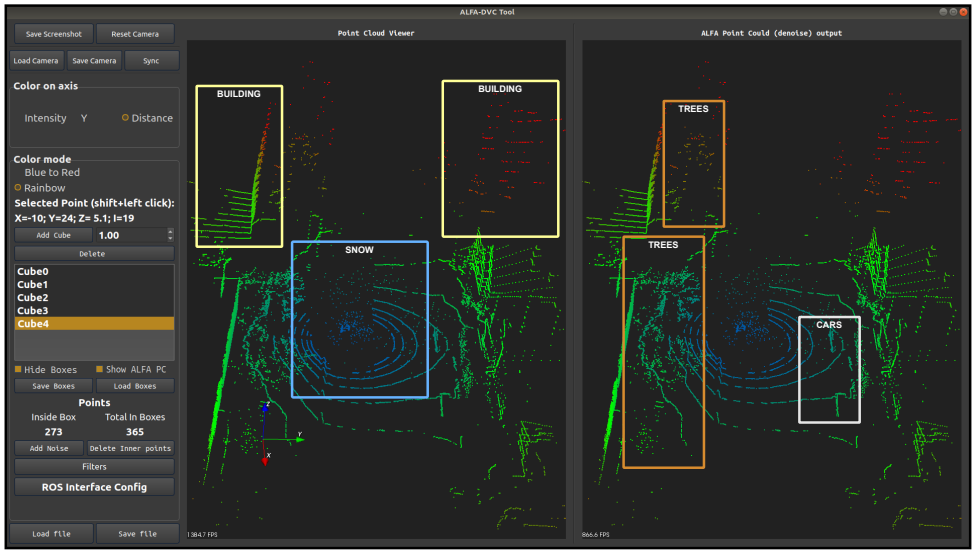


Figure 6.1: ALFA-Pd system architecture.

Depicted in Fig**??**, is the before/after comparison by using the VG filter. At the left point cloud, is shown the original frame, where it is visually possible to see all the noise

caused by the intense snowfall. At the right is illustrated the output point cloud of this filter.

Visually, the output of the VG filter seems very similar to the original point cloud. Voxel grid filters consist in defining 3D boxes (forming a voxel grid) in the 3D space of the point cloud, then, for each voxel, the algorithm selects a point (usually the central point or the centroid of the box) to approximate the remaining points inside the voxel. Thus, the most affected point cloud zones affected by this filter are the most point dense ones. To the naked eye the point dense zones, are so dense that it seems that no change was performed. But when analyzing the metrics provided by ALFA-DVC it is possible to see the difference between the original and filtered point cloud.

Table 6.2

Algorithm	PR	TP	FP	FN	FPT (ms)	FPS
Voxel Grid	28.8%	12%	29%	87%	6	166

The Tab.**??** depicts the performance metrics obtained using the box system in ALFA-DVC. The voxel grid filter only took 6 milliseconds to finish its processing. It can achieve 166 frames per second, but at a cost of removing 28% points from the point cloud. From all the points labeled as noise, the voxel grid filter only removed 12% of them. By using this filter as noise removal, it was wrongly classified 87% of the points that it removed from the point cloud.

In short, the voxel grid filter offers a cheap and fast executing filter for weather denoising, but it presents bad accuracy for autonomous applications.

6.1.3 Statistical Outlier Removal

The SOR filter classifies points as outliers based on the neighbor information. It's one of the most mature state-of-art filters, and it has directed implementation in the PCL library. In this test, the PCL version was used. SOR is typically used in point clouds that aren't sparse, meaning that for autonomous applications it doesn't perform perfectly.

Depicted in Fig. **??** is the before/after comparison by using the SOR filter. Visually one can notice that this filter did in fact remove most of the labeled noise points. Thus, it's evident that some details of buildings, trees, and road artifacts were lost. It's noticeable the impact of the sparsity in the output. The most faraway objects, with red color, were removed. The tree's leaves were also somewhat removed. To help visualization, the figure is zoomed, not showing the whole point cloud, thus the filtering performance in distant objects can't be seen.

The Tab.**??** depicts the performance metrics obtained using the box system in ALFA-DVC. The SOR filter required, on average, 175 milliseconds per frame executed, achieving

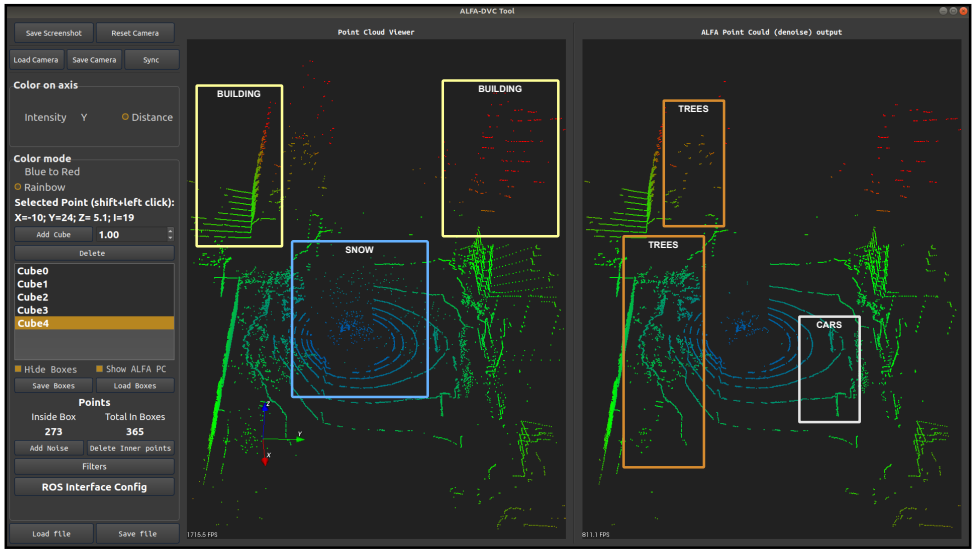


Figure 6.2: ALFA-Pd system architecture.

Table 6.3

Algorithm	PR	TP	FP	FN	FPT (ms)	FPS
SOR	7%	56%	5%	44%	175	5.16

up to 5 frames per second. Most rotor-based LiDAR sensors output point cloud frames at a rate of 10hz, meaning each second, the sensor outputs 10 frames. Furthermore, by requiring more than 175 milliseconds to process, SOR starts introducing delays in the execution flow. The retardment introduced by SOR means that some frames are discarded to maintain synchronism with the sensor.

SOR filter removed an average of 7% of all the points present in the point cloud. From all the labeled noise points, more than half were correctly removed. SOR wrongly classified 5% of the points in the point cloud as outliers, showing that is less impacted by the sparsity than ROR.

SOR presents a light way approach for weather denoising, at a cost of some impact of the sparsity of the point cloud. For low amounts of noise, SOR can be suitable for autonomous driving applications, but for intense noise sessions, SOR shows a lack of a true-positive ratio.

6.1.4 Fast cluster statistical outlier removal

The FCSOR filter is a combination of SOR with a voxel step. By applying the voxel step, the filter reduces the time required per frame. FCSOR was designed for mapping proposes, thus the author didn't provide any results for autonomous applications.

Depicted in Fig. **??**, before/after comparison by using the FCSOR filter. Being a merge of VG and SOR, one can see the resemblance of both visually. It is possible to see

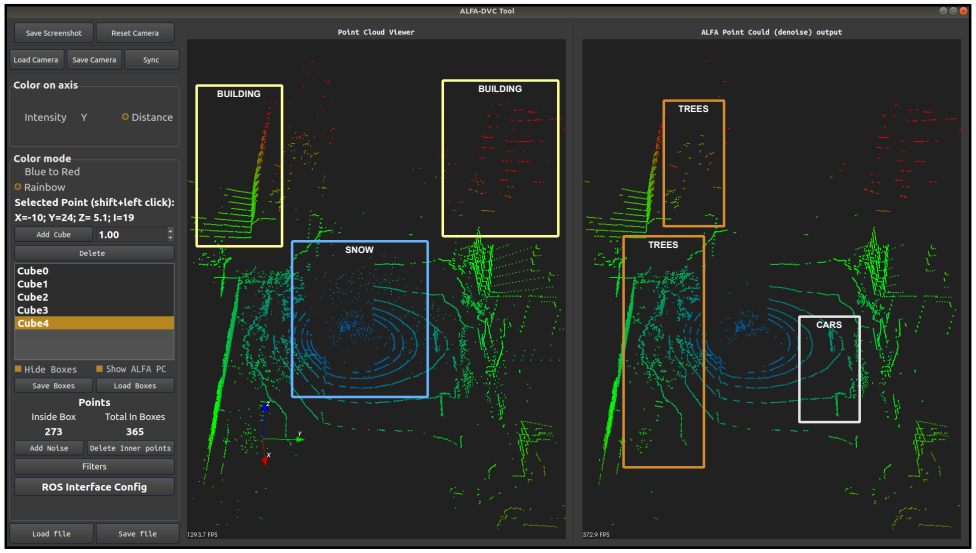


Figure 6.3: ALFA-Pd system architecture.

the downsampling by zooming in the point cloud, and, In faraway objects, a noticeable drop in information is visible. A very significant noise caused by the snowfall was removed close to the vehicle, in the center of the point cloud. Furthermore, similar to SOR, in the building, one can visualize the point difference.

Table 6.4

Algorithm	PR	TP	FP	FN	FPT (ms)	FPS
FCSOR	34%	59%	34%	40%	117	8.5

The Tab.**??** depicts the performance metrics obtained using the box system in ALFA-DVC. FCSOR achieved a mean frame processing time of 117 milliseconds, enabling the processing of almost 9 frames per second. Because SOR filters go through all the points that compose a point cloud, by reducing the number of points, downsampling the point cloud, FCSOR reduces the time required per frame when compared to native SOR. Thus, even by significantly increasing the performance, it is not capable of real-point cloud processing in the embedded system. Furthermore, the addition of the voxel step, increases the overall complexity of the algorithm, making it harder to implement.

Because of the downsampling step of FCSOR, the filter removed 34% of the points in the point cloud, but, of all labeled noise points, it removed almost 60% of them, increasing the performance when compared to SOR. Due to the voxel step, the false positive ratio is 34%, removing a lot of points in dense zones of the point cloud. The denser in a cluster of points, the more points the voxel step removes.

Overall, FCSOR presents a fast SOR solution, ideal for application when downsampling the point cloud inst an issue. Regarding the noise removed, in somewhat ideal

weather conditions, FCSOR can perform well in autonomous applications, but in extreme conditions like an intense downfall, FCSOR shows some lack of performance.

6.1.5 Radius outlier removal

The ROR filter removes points that have few neighbors in a given sphere around them. It's one of the most mature state-of-art filters, and it has directed implementation in the PCL library. In this test, the PCL version was used. ROR is typically used in point clouds that aren't sparse, meaning that for autonomous applications it doesn't perform perfectly.

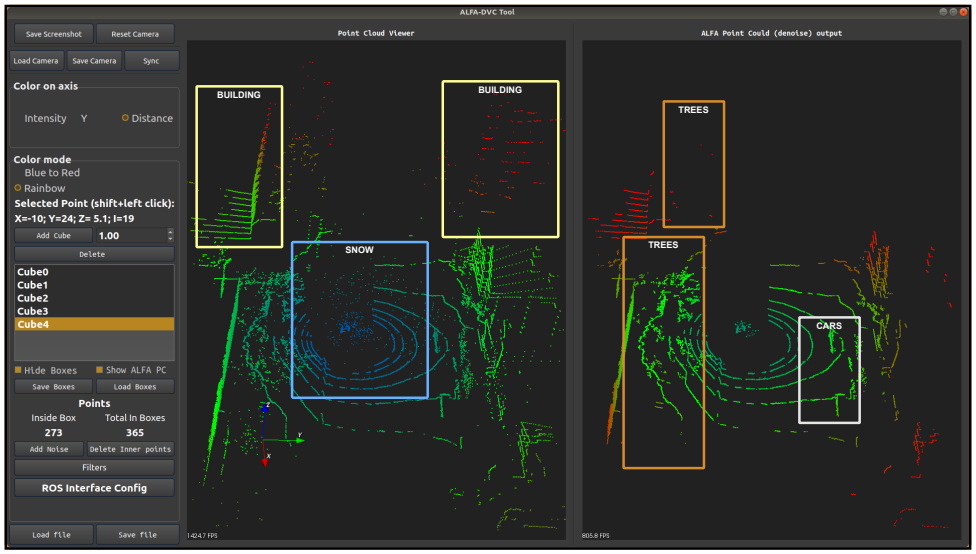


Figure 6.4: ALFA-Pd system architecture.

Depicted in Fig. **??**, before/after comparison by using the ROR filter. Visually, it is possible to observe that ROR removed most of the labeled noise points, only missing the cluster of snow that is gathered on top of the sensor. Because of the fixed radius, ROR fails in classifying far away objects, completing deleting buildings far away from the sensor. Visually, one can conclude that ROR did in fact remove almost all the noise present in the point cloud, but at a cost of critical information in faraway objects. The tree leaves were also heavily deleted by ROR, which can be critical for high-lever object classifying algorithms.

Table 6.5

Algorithm	PR	TP	FP	FN	FPT (ms)	FPS
ROR	19%	75%	18%	24%	180	5.50

Tab. **??** depicts the performance metrics obtained using the box system in ALFA-DVC. The ROR filter takes in averages 180 milliseconds to finish the processing of a point

cloud frame. With this frame processing time, ROR can output a mean of 5.5 frames per second, by executing on an embedded system. Like other filters, by having a frame per second lower than 10, ROR introduces a delay in the executing flow. Furthermore, to ensure that the most recent point cloud frame is processed, ALFA-Pd needs to discard unprocessed frames.

ROR removed approximately 19% of all points in the point cloud. Of the labeled noise points, 75% were removed by ROR, only missing 24% of them. But by analyzing the metrics one can see the impact of the sparse point clouds in this filter. ROR wrongly classified 24% of the points removed, furthermore, this validates what is visually analyzed at the buildings and trees that were removed from the point cloud.

Globally ROR presents a low-complexity outlier removal approach, that can perform well in extreme weather conditions, in an application where distant points aren't important. In autonomous applications LiDAR is used to identify faraway objects, so ROR is not suitable for those applications.

6.1.6 Dynamic radius outlier removal

The DROR filter, like ROR, removes points that have few neighbors in each sphere around them. Instead of having a fixed sphere radius, DROR uses a dynamic one, to address the sparsity of autonomous point clouds. Two versions of DROR were tested, the original, obtained through analyzes of pseudocode from the author, and using a multi-threading approach developed in this dissertation.

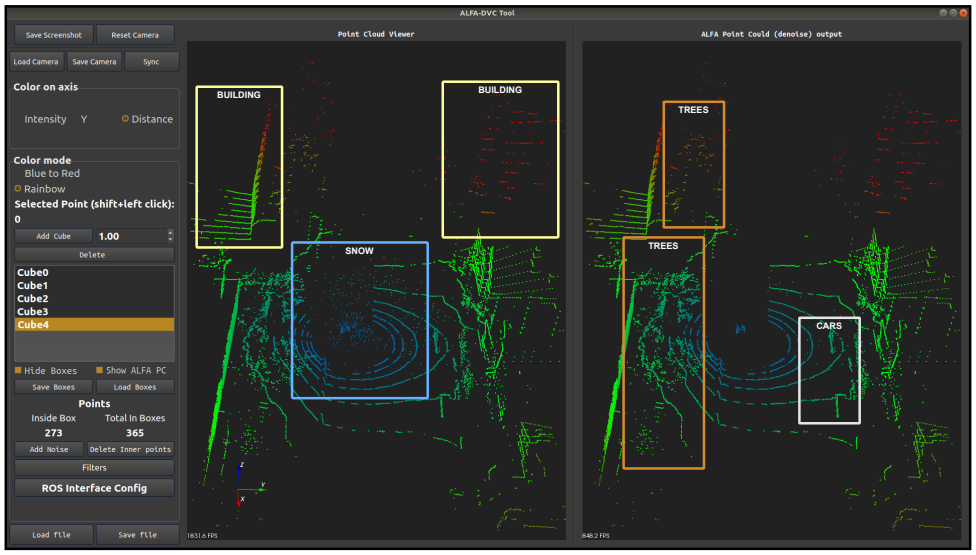


Figure 6.5: ALFA-Pd system architecture.

Depicted in Fig. **??**, before/after comparison by using the DROR filter. The visual output of the original and the multithreading were the same, thus only one figure is enough to show the output.

To the naked eye, DROR removed most of the labeled snow noise, only missing the cluster of points close to the vehicle. This is due to the high intensity of the cluster, because of the snow build-up in front of the sensor. While removing the labeled noise, DROR didn't remove any detail of faraway objects, eg: trees and buildings, proving to be a very capable denoising algorithm. In fact, the tree leaves that resemble noise were preserved, while in other filters, those were considered noise and removed from the point cloud.

Table 6.6

Algorithm	PR	TP	FP	FN	FPT (ms)	FPS
DROR 1T	2%	87%	1%	13%	2730	0,36
DROR 4T	2%	87%	1%	13%	1193	0.83

Tab. **??** depicts the performance metrics obtained using the box system in ALFA-DVC. Because of the search radius calculus and the neighbor search using that radius, DROR needs an average of 2730 milliseconds per frame. With using the multithreading configuration, the filter took 1193 milliseconds, reducing the processing time by 56% from the original algorithm. The base filter, that runs on a single thread configuration, can achieve 0.36 frames per second, but with the version developed in this dissertation, the algorithm was accelerated to 0.83 frames per second. Because of the slow time performance, using the Velodyne puck, for each point cloud processed 9 need to be discarded.

The PCL module used to do the neighbor search takes more executing time when increasing the search radius. Thus, by having a dynamic search radius, the further the point, the bigger the radius, increasing the execution time. The calculation of the search radius also increases the processing time, because it is needed to calculate the search radius for every point in the point cloud.

DROR removed 2% of the point cloud, which 87% of the labeled noise points were removed. Other than the labeled noise points DROR wrongly classified as an outlier approximately 1% of the total points removed, showing high efficiency in removing noise in sparse point clouds. Because the box system of ALFA-DVC is an approximation of all the noise points, the 1% missed, in reality, can be a lower value.

To summarize, DROR presents a robust denoising method that performs well with sparse point clouds. Furthermore, in extreme conditions, this filter is able to remove almost all the noise, without deleting relevant information for high-level applications. For autonomous applications, DROR presents a big drawback in its time performance, not meeting the time constraints for those applications.

6.1.7 Low-intensity outlier removal

The LIOR filter classifies points as inliers based on the intensity values. If a given point doesn't comply with these constraints, a second step is performed based on a ROR approach. To mitigate the problem caused by the sparsity of autonomous point clouds, the original algorithm only filters points that have a distance below 71 meters. Because this parameter limits the execution for snow noise, and for that specific LiDAR, that step was not implemented in this dissertation. This filter was based on the description provided by the author and was further accelerated using a multithread configuration. Both implementations were tested to analyze the impact of parallelizing the processing flow.

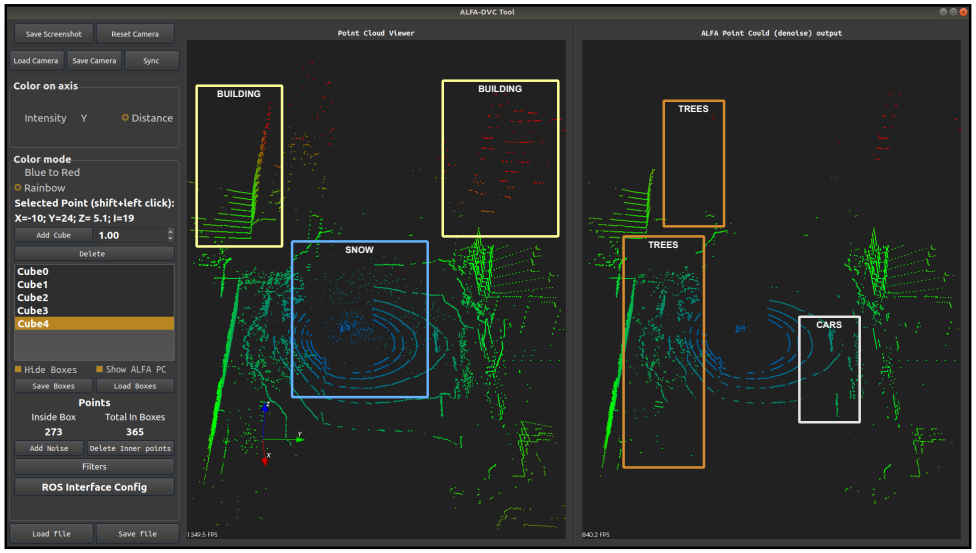


Figure 6.6: ALFA-Pd system architecture.

Depicted in Fig. **??**, before/after comparison by using the LIOR filter. Equally to DROR, the multithreading configuration only affects time-based metrics, thus only one image is shown to depict the output.

The output that is observed by the naked eye is similar to the ROR one. The filter shows to be capable of removing most of the labeled noise, but denser noise caused by the intense snowfall is still wrongly classified as inlier by the filter. Because LIOR has a fixed radius, for distant objects, the distance between points increases, thus, is observable that information of lost objects is lost. One can observe that trees leaves are also classified as outliers, which can cause further problems in high-level applications.

Tab. **??** depicts the performance metrics obtained using the box system in ALFA-DVC. The original filter required an average of 173 milliseconds per frame to execute. Using the multithread configuration, the time required to process a frame was reduced to 109 milliseconds, showing an improvement of 37% when comparing it to the original. With

Table 6.7

Algorithm	PR	TP	FP	FN	FPT (ms)	FPS
LIOR 1T	11%	74%	10%	25%	173	5,7
LIOR 4T	11%	74%	10%	25%	109	9,17

this frame processing time, the base algorithm can output 5.7 frames per second, meaning that in a second of processing, 5 frames are discarded. The multithread accelerated version can achieve up to 9 frames per second. Most LiDAR sensors output 10 frames per second. Having the multithread LIOR, to ensure that the latest frame is executed, it is only required to discard one frame to maintain the synchronism.

LIOR filter removed 11% of all the point cloud points. Because it is based on ROR, the true positive ratio is similar to it. LIOR removed approximately 74% of the labeled snow points, but it wrongly classified 10% of removed points.

Overall, LIOR presents a fast executing algorithm, with capable denoising features. For bad weather conditions, LIOR is a good approach to use, especially If a region of interest is applied as the authors did. By applying an ROI, far away objects still maintain the crucial information that normally is lost. For extreme weather conditions, LIOR presents a somewhat lack of performance when compared with other state of art algorithms.

6.1.8 Dynamic low-intensity outlier removal

The DIOR filter is the novel approach weather denoising filter proposed in this dissertation. This filter is a combination of LIOR and DROR, having the advantages of both filters. DIOR classifies points as inliers or outliers based on their intensity valuer. If the intensity verification fails, a second step is performed based on the DROR approach. To counteract the point cloud sparsity, DIOR uses a dynamic search radius to remove points that have few neighbors in each sphere around them.

To do a fair comparison between DIOR and other state of art algorithms, DIOR was tested in a single thread/multithread configuration, using the same configuration as LIOR and DROR, dividing the workload into four threads.

Depicted in Fig. **??**, before/after comparison by using the DIOR filter. The visual output of the original and the multithreading were the same, thus only one figure is enough to show the output.

To the naked eye, DIOR seems to remove all the labeled noise points caused by the intense snow. It is visually seen the difference between the original image and the filtered one. Regarding the far-away objects, at a first glance, no information was removed. Visually the results were like the provided by DROR. DIOR seems a very capable filter

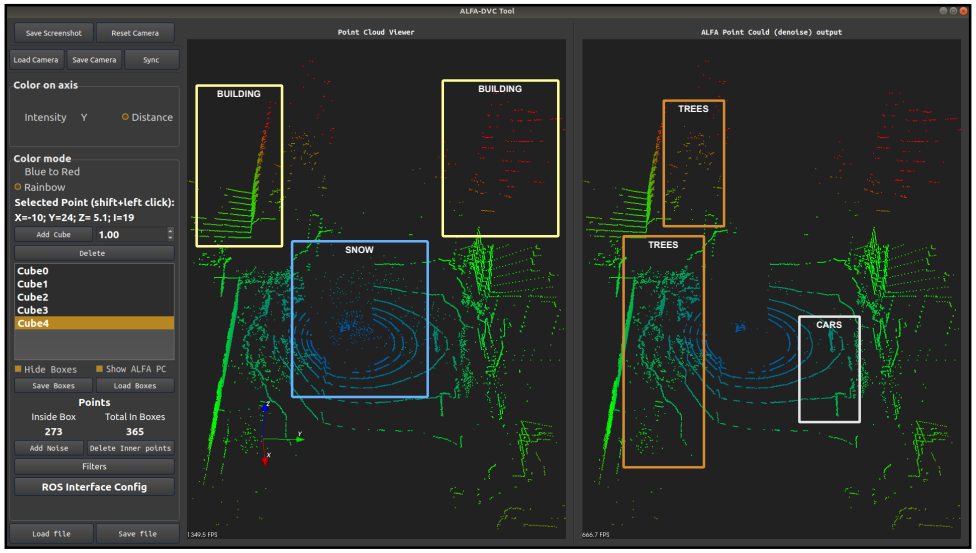


Figure 6.7: ALFA-Pd system architecture.

for extreme weather denoising noise, where no ROI needs to be applied in order to preserve point cloud data like LIOR.

Table 6.8

Algorithm	PR	TP	FP	FN	FPT (ms)	FPS
DIOR 1T	2%	87%	0%	14%	1009	0,99
DIOR 4T	2%	87%	0%	14%	442	2,26

Tab. **??** depicts the performance metrics obtained using the box system in ALFA-DVC. Without using multithreading DIOR can achieve a mean time of 1009 milliseconds per frame. Upon enabling the multithread configuration, the filter processing time was reduced to 442 milliseconds per frame. With this frame processing time, the base algorithm is capable of outputting approximately 1 frame per second, but with the multithread accelerated version, the throughput of frames increases to 2.26 frames per second. Related to the base algorithm, the multithreading increased the performance by 56%. Based on the frames per second that the best version of DIOR can output, to maintain synchronism between processed frames, each second 7 frames needs to be discarded.

DIOR filter removed 2% of all the points in the point cloud. Of all the labeled noise points, DIOR removed 87% of them, furthermore, the number of false classified noise was approximately 0. DIOR presents high efficiency on removing, it missed 14% of the label points due to the high density of the noise when it is very close to the vehicle. This filter achieved results similar to the ones obtained with DROR, but with a slight improvement over the false positive metric.

To summarize, DIOR presents a highly efficient filter for extreme weather-affected point clouds. By combining the working principles of LIOR and DROR, it can achieve

the high accuracy of DROR but reducing the processing time required to process a point cloud frame. Because DIOR still needs to discard frames to ensure that the most recent is processed, this filter is not suited for autonomous application in its current format, thus the reason for accelerating this filter using dedicated hardware accelerators.

6.1.9 Discussion

The VG algorithm outperforms the remaining methods in terms of the required processing time, i.e., it executes faster. However, it does not provide enough accuracy in finding noise, removing around 28% of the points with a false negative rate of nearly 87%.

Regarding DROR, it gives better results in all metrics when compared with ROR, but it requires nearly 6.6x more processing time for denoising one frame.

Concerning SOR and FCSOR, they provide a TP rate between 55% and 60% with a high FN rate of nearly 40%.

Respecting LIOR, which uses ROR principles, it can provide good accuracy results while processing point cloud frames faster than the other methods. Lastly, with DIOR, which proposes the same principle of LIOR combined with DROR, it is possible to achieve great results in terms of PR (2%), FP (0%), and FN (14%), but at the cost of a slightly high FPT.

Using the ALFA-Pd framework, it requires a higher processing time to filter in software one point cloud frame, which is mainly caused by the limited resources in the hardware platform, such as the PS and DDR memory speed.

However, because the framework provides acceleration capabilities, we will focus on improving the processing time of the methods that provide the best TP and FP rate, i.e., DROR, LIOR, and DIOR, which we consider the most important metric in detecting noise while removing only the necessary noise points from the point cloud. These algorithms will be offloaded to dedicated hardware accelerators deployed in the FPGA.

Although the other algorithms could also be accelerated, they would still keep their undesired TP, FP, and FN rate.

6.2 Hardware-based denoising algorithms

The evaluation of the hardware-based denoising algorithms includes the same setup used in the software version regarding the software and hardware platform, as well as the filter parameters for the selected denoising methods.

However, the hardware deployment resorts to the FPGA technology available in the hardware platform. The gathered results include the performance assessment of each algorithm for different combinations of PCs, the hardware resources required to deploy

the solution in the available FPGA fabric, and the required source lines of code (SLoC) to write the different hardware modules supported by the framework.

6.2.1 Dynamic radius outlier removal

DROR was one of the filters implemented in hardware, because of the good potential that it achieves in early software evaluation. The big downside of DROR was that it didn't meet the time constraints for autonomous applications thus the need to be accelerated.

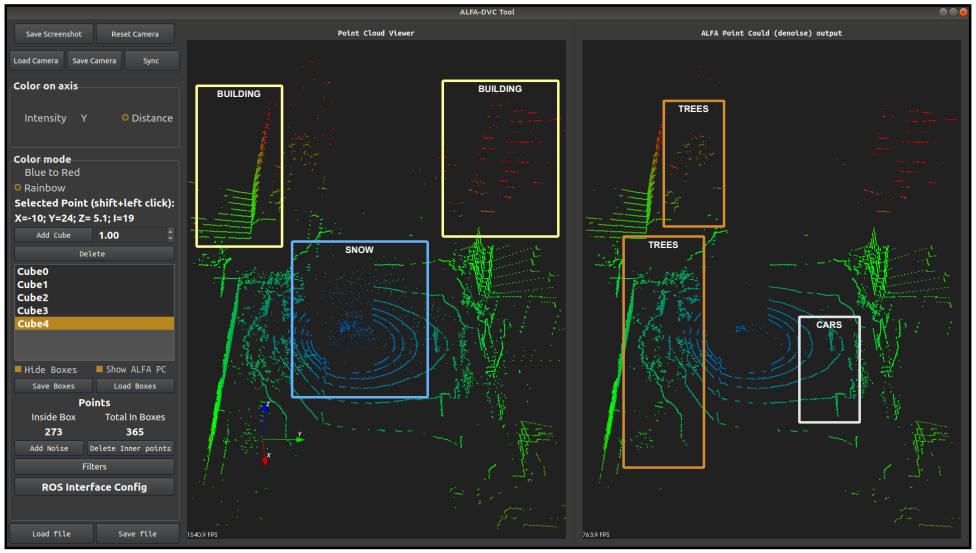


Figure 6.8: ALFA-Pd system architecture.

Depicted in Fig. **??**, before/after comparison by using the DROR filter. Every configuration had similar visual outputs, thus only one was displayed.

Visually, the output is very similar to the software counterpart. The hardware-accelerated filter removed most of the labeled points with a few points missing. The buildings and tree's point information was preserved, so the performance doesn't get impacted by the increase of the point distance to the sensor, showing that it can perform well when applied to sparse point clouds.

BRAM version

Using the BRAM version, the number of point clusters was increased to analyze the impact of those in the performance.

The tab. **??** depicts the results obtained through the evaluation of the DROR hardware BRAM version. The performance gained by increasing the number of point clusters is noticeable. The base implementation required 689 milliseconds to finish the processing of a complete point cloud. Furthermore, by increasing the number of point clusters to

Table 6.9

Algorithm	PCs	PR	TP	FP	FN	FPT (ms)	FPS
DROR	2	5.8%	95%	3%	5.4%	689	1.45
	4	5.3%	95%	3%	5.5%	350	2.8
	8	5.6%	93.2%	3.4%	6.8%	170	5.89
	16	5.35%	92%	3%	7%	84	11.9
	32	4%	87%	3%	13%	40	27.77

double, the processing time was reduced to 350 milliseconds, showing an improvement of almost 50%. Upon increasing the number of point clusters to 32, the filter required 40 milliseconds to complete the processing of a frame achieving 27.77 frames per second. To achieve real-time, the throughput of the filter needs to be higher than 10 frames per second. DROR can achieve that using 16 point clusters.

Due to different filter configurations, the filter performance differs from the software implementation, but overall, still similar to the metrics obtained when running on software. For the best time-related configuration, DIOR removed 4% of all the point cloud points. Of the labeled noise points, 87% were removed and only 3% of the removed points were wrongly classified as outliers.

DDR version

Table 6.10

Algorithm	PCs	PR	TP	FP	FN	FPT (ms)	FPS
DROR	2	6%	92%	4.3%	7.6%	1173	0.85
	4	5.9%	92%	4.2%	7.8%	545	1.84
	8	5.6%	90%	3.75%	9.4%	220	4.5
	16	5.35%	92%	3.4%	7.9%	84	11.9
	32	5.25%	91%	3.6%	8.3%	55	18.1

The performance of the DDR version is similar to the BRAM counterpart, but as expected, it has an increase in the frame processing time and a drop in the frame rate. The fastest implementation is capable of outputting 18.1 frames per second, while, same as the BRAM, the 16 point cluster configuration is enough to handle the sensor in use.

Overall, the DROR hardware implementation complies with the requisites to be used in autonomous applications. It provides a good true positive ratio, while still complaining of time constraints.

6.2.2 Low-intensity outlier removal

LIOR was one of the filters implemented in hardware, due to the potential shown in early software evaluation. In those tests, LIOR almost achieved the 10 frames per second barrier upon using the multithread configuration. Thus, LIOR was accelerated to provide a real-time execution option to the state-of-art filter when running on an embedded system.

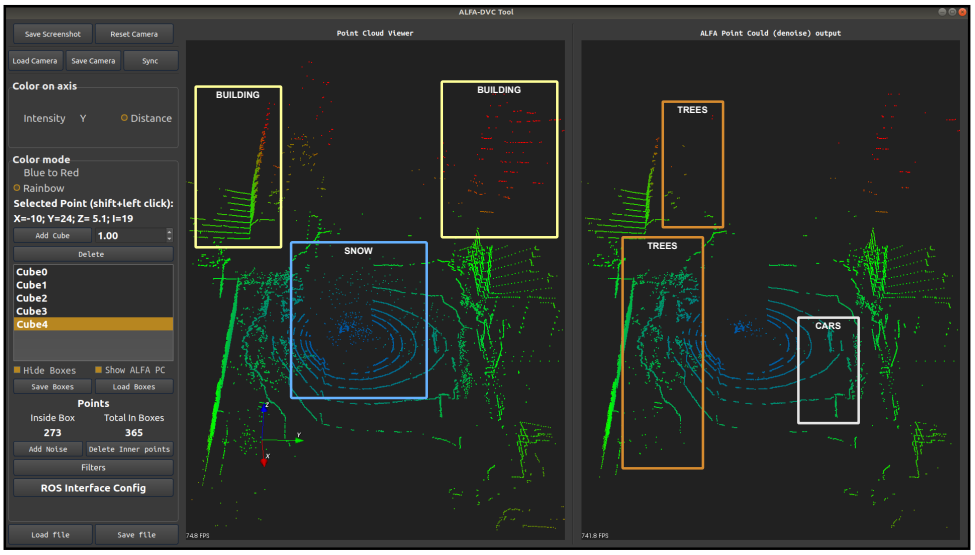


Figure 6.9: ALFA-Pd system architecture.

Depicted in Fig. **??**, before/after comparison by using the LIOR filter. Because to the naked eye, the output doesn't change upon changing the hardware configuration, only one figure is shown to visualize the output

The hardware version of LIOR has a lot of similarities to the software counterpart. The filter removed a big part of the labeled noise, but it is also possible to observe that some information of faraway objects was lost. The most affected areas of wrong classifying were the tree leaves, where one can see that they were almost removed from the point cloud. When compared to the software implementation, it is possible to see that the number of relevant information lost is smaller, showing a miner improvement over it.

BRAM version

The BRAM version proposes a faster execution version of the LIOR. As with all the hardware-implemented algorithms, it was evaluated with different point clusters to see their performance impact them.

The tab. **??** depicts the results obtained through the evaluation of the LIOR hardware BRAM version. On the base configuration, using 2 point clusters, the hardware LIOR

Table 6.11

Algorithm	PCs	PR	TP	FP	FN	FPT (ms)	FPS
LIOR	2	11%	73%	9%	26%	631	1.58
	4	10.65%	72.05%	8.9%	28%	308	5.31
	8	9%	73%	8.9%	30%	143	6.9
	16	9%	71%	7.6%	30%	56	19.6
	32	9%	68%	6.9%	34%	30	33.3

required 631 milliseconds to finish the processing flow. Furthermore, by increasing the number of point clusters, one can see the time required per frame decreases. Using 32 point clusters, LIOR takes 33 milliseconds to conclude the execution.

LIOR with the fastest configuration tested can achieve up to 33 frames per second. The only limit for the point cluster number is the resources available in the platform in use. Thus, the performance can still be increased by raising the number of point clusters. Using 16 point clusters hardware LIOR already meets the time requirements, being the most optimized version for resources.

LIOR removed an average of 9% of all points in the point cloud. Of the labeled points, 68% were correctly classified as inliers by the filter. Regarding the number of false positives, LIOR wrongly classified 7% of points as outliers, showing the deficiency when applied to sparse point clouds.

DDR version

The DDR version offers a more generic approach, easing the portability to other platforms. The same evaluation methodology applied in the BRAM version was also applied here.

Table 6.12

Algorithm	PCs	PR	TP	FP	FN	FPT (ms)	FPS
LIOR	2	14%	70%	12%	30%	970	1.03
	4	10.81%	67%	9.25%	33%	383	2.61
	8	11.48%	68%	10.25%	32%	150	6.71
	16	11.56%	68%	11.3%	31%	56	19.6
	32	12%	68%	12.4%	31%	49	20.408

The performance metrics that are not time-related were very similar to the ones obtained in the BRAM version. The DDR version shows more slow execution times but still complies with time constraints. The configuration using 32 point clusters is capable of outputting 20 processed frames per second. With these results, LIOR is capable of processing two Velodyne Pucks at the same time without introducing delays to the system.

To summarize, LIOR hardware improved the time performance of the original version. This filter now presents a fast-executing denoise method capable of achieving high frame rates. It still lacks performance under extreme weather conditions and, the impact of sparse point clouds is noticeable.

6.2.3 Dynamic low-intensity outlier removal

DIOR being the proposed method was accelerated in hardware, due to the promising results obtained in the software tests. But the software evaluation has also shown the need to further accelerate the algorithm thus it doesn't meet the time constraints for autonomous applications.

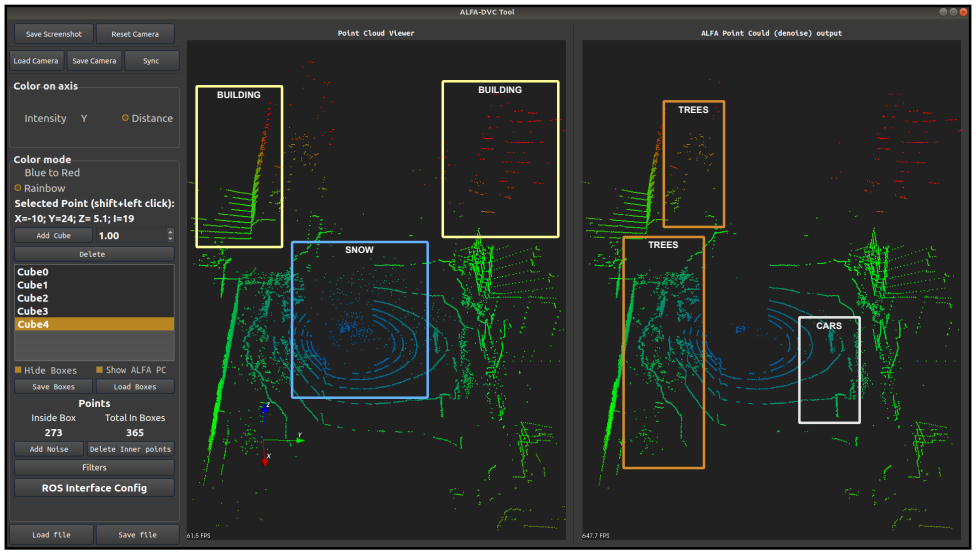


Figure 6.10: ALFA-Pd system architecture.

Depicted in Fig. **??**, before/after comparison by using the LIOR filter. To help readability, the figure used was obtained upon using 32 point cluster as configuration. Other frames were discarded due to the lack of major differences between them.

The hardware version of DIOR, as all the hardware algorithms, shows identical visual outputs when compared with the software ones. Most of the snow-generated noise was removed by this filter. Furthermore, the difference between the original and the filters are significant, even to the naked eye. Because of the features inherited from DROR, the efficiency of the filter is very high. Faraway objects were untouched, and tree leaves were preserved. Faraway tree leaves that were the most challenging objects to outlier-based methods were barely deleted by the algorithm.

BRAM version

The BRAM version proposes a faster execution version of the DIOR. As with all the hardware-implemented algorithms, it was evaluated with different point clusters to see their performance impact them.

Table 6.13

Algorithm	PCs	PR	TP	FP	FN	FPT (ms)	FPS
DIOR	2	5%	93%	3%	5.8%	476	2.1
	4	5%	93%	3%	5.6%	244	4.1
	8	5%	93%	3%	7%	124	8.1
	16	4.35%	91%	2.5%	7%	56	19.6
	32	4%	88%	2.8%	13%	30	33.3

The tab. **??** depicts the results obtained through the evaluation of the DIOR hardware BRAM version. Considering the base implementation the one using 2 point clusters, LIOR with its lighter version requires 476 milliseconds to process a point cloud frame. By increasing the number of point clusters it's noticeable the increase in time-related metrics. Using the fastest implementation, the one with 32 point cluster, DIOR achieved 30 milliseconds of frame processing time. This time is equal to what LIOR requires to execute, but with better ratios.

As with other hardware algorithms, there is no theoretical limit to the number of point clusters, so the performance times obtained, can still be improved by increasing the number of those modules.

The hardware DIOR presents very similar time performance to LIOR, achieving up to 33.3 frames per second. But to achieve real-time, the implementation using 16 point clusters suffice. Furthermore, using this configuration DIOR can execute 2 sensors without introducing any delays to the execution flow.

DIOR removed an average of 4% of all points in the point cloud. Of the labeled points, 90% were correctly classified as inliers by the filter. Regarding the number of false positives, DIOR only wrongly classified 3% of points as outliers, showing the efficiency of removing weather-generated noise in extreme conditions.

DDR version

The DDR version offers a more generic approach, easing the portability to other platforms. The same evaluation methodology applied in the BRAM version was also applied here.

The results obtained using the DDR version are very similar to the ones obtained with the BRAM version, apart from the time-related ones. By using a more generic

Table 6.14

Algorithm	PCs	PR	TP	FP	FN	FPT (ms)	FPS
DIOR	2	6.62%	92%	5.8%	7%	761	1.35
	4	4.9%	90%	3%	7%	365	2.74
	8	4.45%	89%	2.5%	7%	146	6.84
	16	4.35%	91%	2.5%	7%	56	19.6
	32	4%	88%	3.8%	13%	33	30.30

memory, the frame processing time, of lower point cluster configurations, increases. On a higher count of point cluster modules, this difference isn't that noticeable, due to the high parallelization of the algorithm. With the faster execution configuration, this version was still able to achieve 30.3 frames per second, being more than enough for the sensor in use.

Overall DIOR in hardware presents the good time performance of LIOR, with the excellent true positive ratio of DROR. Because in hardware, the methodology of finding neighbors is different, the dynamic search radius doesn't impact the processing time. Thus, it's possible to have good ratios without impacting the time requirements of the algorithm.

6.2.4 Hardware Resources

ALFA-Pd offers a modular approach where the user can choose the number of point clusters and the neighbor finder's modules. By increasing the number of those modules, the denoising algorithm gets more parallelized. Furthermore, the number of resources required to generate the hardware also increases. A study of the resources required per point cluster configuration was done.

Table 6.15

PCs	Version	LUTS	FFs	BRAM	DSP
2	BRAM	7.79%	4.05%	83.3%	0.92%
	DDR	3.41%	2.19%	3.84%	0.57%
4	BRAM	9.01%	4.18%	83.3%	1.85%
	DDR	4.77%	2.33%	3.84%	1.15%
8	BRAM	11.96%	4.39%	83.3%	3.7%
	DDR	7.19%	2.62%	3.84%	2.31%
16	BRAM	19.09%	5.09%	83.3%	7.4%
	DDR	13.11%	3.17%	3.84%	4.63%
32	BRAM	39.30%	5.68%	83.3%	14.81%
	DDR	26.13%	3.75%	3.84%	8.68%

The performance gain comes at the cost of FPGA hardware resources, which increases with the increase of the number of PC blocks.

Table 6.15 summarizes the trade-off between the hardware resources and the number of point clusters, varying from 2 to 32. The number of neighbor finders is fixed, 2 for the BRAM version, and 1 for the DDR version.

The three algorithms supported by the hardware version of the ALFA framework are all deployed inside a PC module, which can be activated on-the-fly by the ALFA-DVC tool when required.

BRAM version

Deploying the hardware with 32 PCs requires nearly 39% of the available lookup tables (LUTs), 5.68% of Flip-Flops (FFs), 83.3% of the available block RAM (BRAM), and 14.81% of digital signal processor (DSP) blocks. Note that the high percentage of BRAMs in use is due to the pre-allocation of this memory, thus, not meaning that they are always in use. With this allocation, ALFA-Pd offers compatibility with top-of-the-shelf sensors capable of outputting 800 000 points per frame. However, considering the frame rate achieved by the three algorithms in processing the selected point cloud, 16 PCs can already provide real-time capabilities to sensors with an output of 10Hz at the cost of a few hardware resources.

DDR version

The DDR version uses half of the neighbor finders when compared to the BRAM version. Despite that, in implementations with more than 4 point clusters, its required more than half of the resources of the BRAM version. This is due to the increased complexity of this version, escalating more quickly upon increasing the number of point clusters. Deploying 32 PCs requires almost 26% LUTs and 3.75% of the available FFs. The number of BRAM used decreased significantly, to 3.84%, and the number of DSP also decreases to 8.68%.

However, using the DDR version and for this dataset, is possible to achieve real-time using 16 point clusters. Thus, presenting a light way denoising method, capable of filtering noised point clouds without disturbing the execution flow of high-level applications.

6.3 Hardware vs Software Implementations

Performance-wise, the number of PC and NF dictates the required time to process one point cloud frame. Regarding the PR, TP, FP, and FN rates, the obtained values are identical to the software-only results depicted in Table **??**, while it is still visible that our approach provides better performance results than LIOR. Such results prove the correctness of the hardware implementation of the selected algorithms since the hardware

deployment only accelerates the point cloud frame processing time without changing the algorithm's behavior.

Regarding the processing time, and because we have deployed the algorithms with parallelization capabilities, increasing the number of PCs has an observable impact on the time required to process one point cloud frame. When the system was only deployed with 2 PCs, the processing time for DROR and DIOR is almost identical with their software implementation, while for the LIOR it results in a significant time increase from 109 ms to 631 ms. When the number of PCs is higher than 2, the results are always better than the software-only implementation, achieving an FPT of 40 ms for DROR, and 30 ms for LIOR and DIOR when the number of PCs is 32. This results in a frame rate of 27.7 FPS for DROR, and 33.3 FPS both for LIOR and DIOR.

There is no theoretical maximum to the number of point cluster and neighbor finder modules. In ideal conditions, without resources limitations, if the number of PCs and NF is equal to the point cloud size, the frame will be processed within one clock cycle. But using 32 PCs, the BRAM increased the performance of DROR almost 98 times when compared to the base version, and 42 times of the multithread version. Regarding LIOR, ALFA-Pd reduced 3.3 times the required processing time.

7. Conclusion

7.1 Conclusions

7.2 Future Work

References

[1] J. Guerrero-Ibáñez, S. Zeadally, and J. Contreras-Castillo, "Sensor Technologies for Intelligent Transportation Systems," *Sensors (Basel, Switzerland)*, vol. 18, p. 1212, Apr 2018.

[2] E. Marti, M. A. de Miguel, F. Garcia, and J. Perez, "A Review of Sensor Technologies for Perception in Automated Driving," *IEEE Intelligent Transportation Systems Magazine*, vol. 11, no. 4, pp. 94–108, 2019.

[3] B. Shahian Jahromi, T. Tulabandhula, and S. Cetin, "Real-Time Hybrid Multi-Sensor Fusion Framework for Perception in Autonomous Vehicles," *Sensors*, vol. 19, no. 20, 2019.

[4] A. S. Mohammed, A. Amamou, F. K. Ayevide, S. Kelouwani, K. Agbossou, and N. Zioui, "The Perception System of Intelligent Ground Vehicles in All Weather Conditions: A Systematic Literature Review," *Sensors*, vol. 20, no. 22, 2020.

[5] C. H. Kuo, C. C. Lin, and J. S. Sun, "Modified microstrip franklin array antenna for automotive short-range radar application in blind spot information system," *IEEE Antennas and Wireless Propagation Letters*, vol. 16, 2017.

[6] F. R. A. Zakuan, H. Zamzuri, M. A. A. Rahman, W. J. Yahya, N. H. F. Ismail, M. S. Zakariya, K. A. Zulkepli, and M. ZulfaqarAzmi, "Threat assessment algorithm for active blind spot assist system using short range radar sensor," *ARPN Journal of Engineering and Applied Sciences*, vol. 12, 2017.

[7] M. Ash, M. Ritchie, and K. Chetty, "On the application of digital moving target indication techniques to short-range fmcw radar data," *IEEE Sensors Journal*, vol. 18, 2018.

[8] H. Iqbal, A. Löffler, M. N. Mejdoub, D. Zimmermann, and F. Gruson, "Imaging radar for automated driving functions," *International Journal of Microwave and Wireless Technologies*, vol. 13, 2021.

[9] F. Pfeiffer and E. M. Biebl, "Inductive compensation of high-permittivity coatings on automobile long-range radar radomes," *IEEE Transactions on Microwave Theory*

and Techniques, vol. 57, 2009.

[10] Y. Yu, W. Hong, H. Zhang, J. Xu, and Z. H. Jiang, "Optimization and implementation of siw slot array for both medium- and long-range 77 ghz automotive radar application," *IEEE Transactions on Antennas and Propagation*, vol. 66, 2018.

[11] F. E. Sahin, "Long-range, high-resolution camera optical design for assisted and autonomous driving," *Photonics*, vol. 6, 2019.

[12] H. Rashed, E. Mohamed, G. Sistu, V. R. Kumar, C. Eising, A. El-Sallab, and S. Yogamani, "Generalized object detection on fisheye cameras for autonomous driving: Dataset, representations and baseline," 2021.

[13] B. Mohammadian, M. Sarayloo, J. Heil, H. Hong, S. Patil, M. Robertson, T. Tran, V. Krishnan, and H. Sojoudi, "Active prevention of snow accumulation on cameras of autonomous vehicles," *SN Applied Sciences*, vol. 3, 2021.

[14] E. Synge, "XCI. A method of investigating the higher atmosphere," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 9, no. 60, pp. 1014–1020, 1930.

[15] M. A. Tuve, E. A. Johnson, and O. R. Wulf, "A new experimental method for study of the upper atmosphere," *Journal of Geophysical Research*, vol. 40, no. 4, p. 452, 1935.

[16] J. Ring, "The Laser in Astronomy," *New Scientist*, vol. 344, 1963.

[17] A. K. Biswas and W. E. K. Middleton, "Invention of the Meteorological Instruments," *Technology and Culture*, vol. 12, no. 2, 1971.

[18] J. U. Eitel, B. Höfle, L. A. Vierling, A. Abellán, G. P. Asner, J. S. Deems, C. L. Glennie, P. C. Joerg, A. L. LeWinter, T. S. Magney, G. Mandlburger, D. C. Morton, J. Müller, and K. T. Vierling, "Beyond 3-D: The new spectrum of lidar applications for earth and ecological sciences," *Remote Sensing of Environment*, vol. 186, pp. 372–392, dec 2016.

[19] Y. Li and J. Ibanez-Guzman, "Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems," *IEEE Signal Processing Magazine*, vol. 37, pp. 50–61, jul 2020.

[20] A. Süss, V. Rochus, M. Rosmeulen, and X. Rottenberg, "Benchmarking time-of-flight based depth measurement techniques," in *Smart Photonic and Optoelectronic Integrated Circuits XVIII*, vol. 9751, p. 975118, 2016.

[21] G. Atanacio-Jiménez, J. J. González-Barbosa, J. B. Hurtado-Ramos, F. J. Ornelas-Rodríguez, H. Jiménez-Hernández, T. García-Ramirez, and R. González-Barbosa, "LIDAR velodyne HDL-64E calibration using pattern planes," *International Journal*

of Advanced Robotic Systems, vol. 8, no. 5, pp. 70–82, 2011.

[22] T. Raj, F. H. Hashim, A. B. Huddin, M. F. Ibrahim, and A. Hussain, "A survey on LiDAR scanning mechanisms," 2020.

[23] R. H. Rasshofer and K. Gresser, "Automotive Radar and Lidar Systems for Next Generation Driver Assistance Functions," *Advances in Radio Science*, vol. 3, pp. 205–209, may 2005.

[24] H. Moosmüller, C. Mazzoleni, P. W. Barber, H. D. Kuhns, R. E. Keislar, and J. G. Watson, "On-Road Measurement of Automotive Particle Emissions by Ultraviolet Lidar and Transmissometer: Instrument," *Environmental Science and Technology*, vol. 37, pp. 4971–4978, nov 2003.

[25] R. Isermann, M. Schorn, and U. Stählin, "Anticollision system PRORETA with automatic braking and steering," *Vehicle System Dynamics*, vol. 46, pp. 683–694, sep 2008.

[26] T. Gong, Y. Wang, L. Kong, H. Li, and A. Wang, "Chaotic lidar for automotive collision warning system," *Zhongguo Jiguang/Chinese Journal of Lasers*, vol. 36, no. 9, pp. 2426–2430, 2009.

[27] P. Lindner and G. Wanielik, "3D Lidar processing for vehicle safety and environment recognition," in *2009 IEEE Workshop on Computational Intelligence in Vehicles and Vehicular Systems, CIVVS 2009 - Proceedings*, 2009.

[28] W. Pananurak, S. Thanok, and M. Parnichkun, "Adaptive cruise control for an intelligent vehicle," in *2008 IEEE International Conference on Robotics and Biomimetics, ROBIO 2008*, pp. 1794–1799, 2009.

[29] T. Ogawa, H. Sakai, Y. Suzuki, K. Takagi, and K. Morikawa, "Pedestrian detection and tracking using in-vehicle lidar for automotive application," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, pp. 734–739, IEEE, jun 2011.

[30] K. Granstrom, S. Renter, M. Fatemi, and L. Svensson, "Pedestrian tracking using Velodyne data-Stochastic optimization for extended object tracking," in *IEEE Intelligent Vehicles Symposium, Proceedings*, pp. 39–46, 2017.

[31] C. Lundquist, K. Granstrom, and U. Orguner, "An Extended Target CPHD Filter and a Gamma Gaussian Inverse Wishart Implementation," *IEEE Journal of Selected Topics in Signal Processing*, vol. 7, pp. 472–483, jun 2013.

[32] S. M. Patole, M. Torlak, D. Wang, and M. Ali, "Automotive radars: A review of signal processing techniques," *IEEE Signal Processing Magazine*, vol. 34, pp. 22–35, mar 2017.

[33] . Guo Liren, . Hu Yihua, . Li Zheng, and . Xu Shilong, "Research on Infulence

of Acousto-Optic Frequency Shifter to Micro-Doppler Effect Detection," *Acta Optica Sinica*, vol. 35, no. 2, p. 0212006, 2015.

[34] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Communications of the ACM*, vol. 62, pp. 61–67, feb 2019.

[35] J. Jiménez, "Laser diode reliability: Crystal defects and degradation modes," *Comptes Rendus Physique*, vol. 4, 2003.

[36] "2-d optical-cdma modulation in automotive time-of-flight lidar systems," vol. 2020-July, 2020.

[37] T. Fersch, R. Weigel, and A. Koelpin, "A CDMA Modulation Technique for Automotive Time-of-Flight LiDAR Systems," *IEEE Sensors Journal*, vol. 17, no. 11, pp. 3507–3516, 2017.

[38] "Avm / lidar sensor based lane marking detection method for automated driving on complex urban roads," 2017.

[39] M. E. Warren, "Automotive LIDAR Technology," in *2019 Symposium on VLSI Circuits*, pp. C254–C255, June 2019.

[40] M. Jokela, M. Kutila, and P. Pyykönen, "Testing and validation of automotive point-cloud sensors in adverse weather conditions," *Applied Sciences*, vol. 9, no. 11, 2019.

[41] J. R. Vargas Rivero, T. Gerbich, V. Teiluf, B. Buschardt, and J. Chen, "Weather Classification Using an Automotive LIDAR Sensor Based on Detections on Asphalt and Atmosphere," *Sensors*, vol. 20, p. 4306, aug 2020.

[42] P. H. Chan, G. Dhadyalla, and V. Donzella, "A Framework to Analyze Noise Factors of Automotive Perception Sensors," *IEEE Sensors Letters*, vol. 4, no. 6, pp. 1–4, 2020.

[43] R. Heinzler, P. Schindler, J. Seekircher, W. Ritter, and W. Stork, "Weather Influence and Classification with Automotive Lidar Sensors," *IEEE Intelligent Vehicles Symposium, Proceedings*, vol. 2019-June, pp. 1527–1534, jun 2019.

[44] R. Heinzler, F. Piewak, P. Schindler, and W. Stork, "CNN-Based Lidar Point Cloud De-Noising in Adverse Weather," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 2514–2521, 2020.

[45] T.-H. Sang, S. Tsai, and T. Yu, "Mitigating Effects of Uniform Fog on SPAD Lidars," *IEEE Sensors Letters*, vol. 4, no. 9, pp. 1–4, 2020.

[46] T. Yang, Y. Li, Y. Ruichek, and Z. Yan, "LaNoising: A Data-driven Approach for 903nm ToF LiDAR Performance Modeling under Fog," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10084–10091,

2020.

[47] Y. Li, P. Duthon, M. Colomb, and J. Ibanez-Guzman, "What Happens for a ToF LiDAR in Fog?," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 11, pp. 6670–6681, 2021.

[48] C. Goodin, D. Carruth, M. Doude, and C. Hudson, "Predicting the Influence of Rain on LIDAR in ADAS," *Electronics*, vol. 8, p. 89, jan 2019.

[49] S. Hasirlioglu and A. Riener, "A General Approach for Simulating Rain Effects on Sensor Data in Real and Virtual Environments," *IEEE Transactions on Intelligent Vehicles*, vol. 5, no. 3, pp. 426–438, 2020.

[50] M. Byeon and S. W. Yoon, "Analysis of Automotive Lidar Sensor Model Considering Scattering Effects in Regional Rain Environments," *IEEE Access*, vol. 8, pp. 102669–102679, 2020.

[51] J. P. Espineira, J. Robinson, J. Groenewald, P. H. Chan, and V. Donzella, "Realistic LiDAR With Noise Model for Real-Time Testing of Automated Vehicles in a Virtual Environment," *IEEE Sensors Journal*, vol. 21, no. 8, pp. 9919–9926, 2021.

[52] N. Charron, S. Phillips, and S. L. Waslander, "De-noising of lidar point clouds corrupted by snowfall," in *Proceedings - 2018 15th Conference on Computer and Robot Vision, CRV 2018*, pp. 254–261, 2018.

[53] J. Wu, H. Xu, Y. Tian, R. Pi, and R. Yue, "Vehicle Detection under Adverse Weather from Roadside LiDAR Data," *Sensors*, vol. 20, no. 12, 2020.

[54] S. Hasirlioglu, A. Kamann, I. Doric, and T. Brandmeier, "Test methodology for rain influence on automotive surround sensors," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pp. 2242–2247, IEEE, nov 2016.

[55] M. Kutila, P. Pyykonen, W. Ritter, O. Sawade, and B. Schaufele, "Automotive LIDAR sensor development scenarios for harsh weather conditions," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pp. 265–270, IEEE, nov 2016.

[56] S. Hasirlioglu, I. Doric, A. Kamann, and A. Riener, "Reproducible Fog Simulation for Testing Automotive Surround Sensors," in *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, vol. 2017-June, pp. 1–7, IEEE, jun 2017.

[57] R. Redington, "Elements of infrared technology: Generation, transmission and detection," *Solid-State Electronics*, vol. 5, p. 361, sep 1962.

[58] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–4, 2011.

[59] H. Balta, J. Velagic, W. Bosschaerts, G. De Cubber, and B. Siciliano, "Fast Statistical Outlier Removal Based Method for Large 3D Point Clouds of Outdoor Environments," *IFAC-PapersOnLine*, vol. 51, no. 22, pp. 348–353, 2018.

[60] J. I. Park, J. Park, and K. S. Kim, "Fast and Accurate Desnowing Algorithm for LiDAR Point Clouds," *IEEE Access*, vol. 8, pp. 160202–160212, 2020.

[61] R. Roriz, A. Campos, S. Pinto, and T. Gomes, "Dior: A hardware-assisted weather denoising solution for lidar point clouds," *IEEE Sensors Journal*, pp. 1–1, 2021.

[62] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," *Proceedings - IEEE International Conference on Robotics and Automation*, no. May 2011, 2011.

[63] R. Kelley, "Richard kelley," 2018.