MASTER IN COMPUTER SCIENCE

Palicus

Building a Domain-Specific Accelerator to Pre-Process LiDAR-generated Point Clouds

Master Thesis

Sophie Gabriela Pfister

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

June 2025

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Light Detection And Ranging (LiDAR) sensors are becoming increasingly popular, especially in 3D vision applications. They produce dynamic point clouds, which are considered hard to process due to the large amount of data and the lack of structure. Nevertheless, some applications need to process the data in real-time. A common approach is to accelerate the processing using application-specific circuits. However, such hardware is often fixed-function, and as a result, the systems are costly to maintain or evolve.

This thesis shows that real-time latency requirements can also be met with a domain-specific accelerator (DSA). We pursue a classical data management approach, *i.e.,* defining a data model, to design specialized processing elements and combine them with a crossbar scheduling mechanism to support a wide range of expressions. Our prototype, PALICUS, is a DSA targeting the pre-processing stage in LiDAR-based 3D vision systems. It supports the four dominating point cloud representations for object detection. We further show that PALICUS can build them in real-time and with satisfying accuracy, such that it can serve as a building block handling pre-processing in more complex data processing pipelines.

Philippe Cudré-Mauroux, eXascale Infolab, Department of Informatics, University of Fribourg, Supervisor

Alberto Lerner, eXascale Infolab, Department of Informatics, University of Fribourg, Co-Supervisor

Contents

1 Introduction

Light detection and ranging (LiDAR) sensors have become increasingly popular in recent years and nowadays find applications in many domains such as traffic management, robotics, agriculture, forestry, and archaeology [65]. They perceive their surroundings by emitting laser pulses and detecting the reflected energy. For each pulse, the sensor computes the point of reflection based on the shooting direction and the time of flight [64]. The sensor periodically repeats this procedure, generating a *dynamic point cloud*, a sequence of static frames where each frame describes the sensor's field of view at a given point in time. LiDAR sensors thus produce a steady data stream, which is commonly partitioned and transmitted packet-wise over a network.

Processing point clouds is considered hard due to the large number of points and the unstructured nature of the data, but also the sparsity of the representation [43]. LiDAR-generated point clouds additionally suffer from occlusion [25], and the density of the point cloud is variable [81]. Despite these challenges, some applications need to process the LiDAR data in real time.

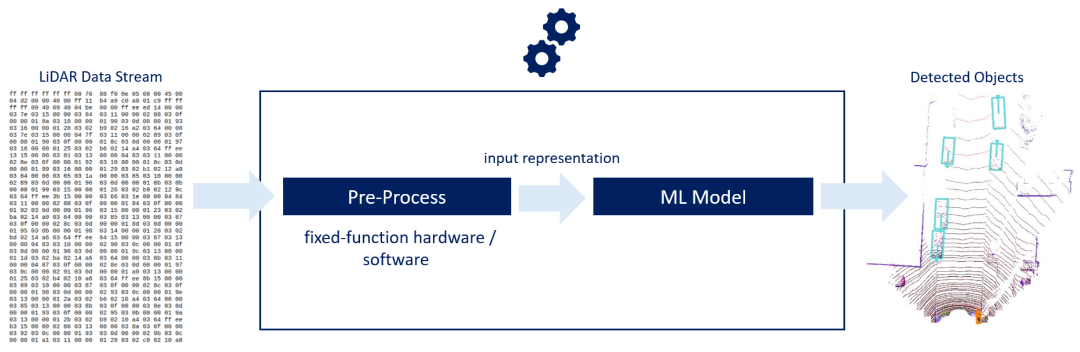


Figure 1.1: **State of the Art (SOTA) System for Object Detection in Point Clouds**. The processing pipeline comprises pre-processing and ML inference. The pre-processing stage is often hard-coded and sometimes implemented into fixed-function hardware.

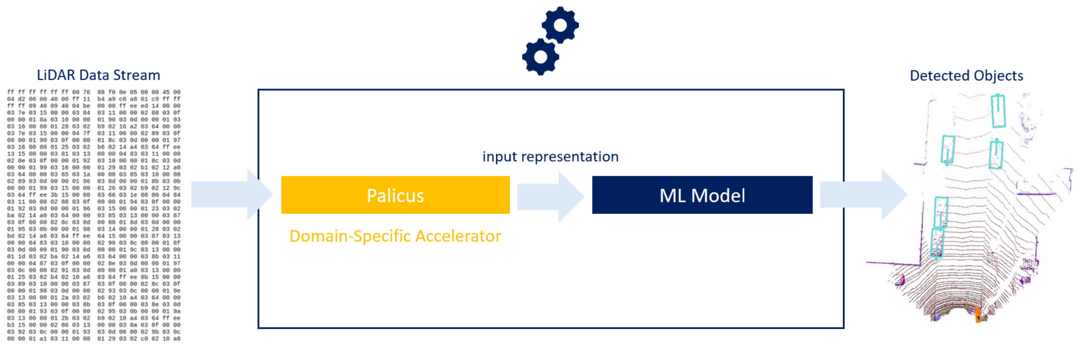


Figure 1.2: **DSA Approach for Object Detection in Point Clouds**. Pre-Processing is handled by a DSA and is thus programmable. This allows to reuse PALICUS for multiple systems and makes the resulting systems more evolvable.

Previous research on processing LiDAR data focused on developing systems that master a given task. Unsurprisingly, most of these systems leverage machine learning techniques, *i.e.,* deep neural networks [24, 35, 52, 86]. These systems are often implemented as a black box that accepts the steady data stream generated by a LiDAR sensor as input and outputs the solution to the target task. As depicted in Figure 1.1, we can divide the data processing pipeline into two stages: (1) *pre-processing* and (2) ML inference. Although this approach may succeed regarding the quality of the solution, many of these systems do not meet real-time latency requirements. Approaches to accelerate the processing pipelines typically optimize the hardware on which the pipeline is running, *i.e.,* by exploiting the parallel computation capabilities of GPUs or application-specific circuits. While powerful GPUs require too much energy to run on the edge, and embedded GPUs are often not able to accelerate the pipelines sufficiently, application-specific circuits can be implemented on low-power devices such as field programmable gate arrays (FPGAs, see Section 2.1). However, such hardware is fixed-function and thus only supports the target system; maintaining or evolving it after deployment is therefore rather costly.

We believe that LiDAR sensors need a *domain-specific accelerator (DSA)*. DSAs are specialized hardware computing engines that speed up data processing by (a) providing specialized operations, (b) exploiting parallel computing, and (c) optimizing memory utilization [17]. Figure 1.2 illustrates our approach: we design a DSA for the pre-processing stage in SOTA systems. DSAs are programmable and can thus serve as a building block in several systems. Moreover, this allows for evolving a system after deployment. For instance, we can switch to a better machine learning model, even if it expects a different input representation.

This thesis demonstrates how to design a domain-specific accelerator by pursuing a data management approach. The DSA is designed for the *pre-processing stage* of *traffic-related applications* that tackle one of the following target tasks:

1. **Semantic segmentation** aims to cluster similar points semantically meaningfully and classify them [43]. We may further distinguish *scene segmentation*, where each point is annotated with a label describing its class, and *instance segmentation*, where each point is assigned to a concrete instance of a class. For example, two points far away from each other may be labeled as "car" (scene) but do not belong to the same car (instance).
2. **Object detection** aims at locating and classifying (foreground) objects in the point cloud. We aim at providing a list of labeled bounding boxes that encompass the detected objects [27].

This thesis thus comprises three major results: (1) a *survey* that helps us understand the domain and its requirements, (2) our formal *data model* – DAMOCLES – which allows us to express complex point cloud manipulations by combining simple operators, and (3) our *DSA for point cloud pre-processing* – PALICUS – that is able to implement DAMOCLES expressions ad-hoc. We will further show that PALICUS can support real-time point cloud processing. For this purpose, we will build various point cloud representations from real-world LiDAR data.

This thesis starts by providing some context and discussing the state-of-the-art when processing LiDAR-generated point clouds in Chapter 2. Chapter 3 briefly discusses our survey and describes the most common point cloud representations. Chapter 4 introduces DAMOCLES. Chapter 5 describes the architecture of PALICUS and chapter 6 discusses its evaluation. Finally, Chapter 7 concludes this thesis and points to future research directions.

2
Background & Related Work

This chapter provides the technological and conceptual background of the thesis. Section 2.1 introduces field-programmable gate arrays (FPGAs). Section 2.2 explains the concept of point clouds, while Section 2.3 has a closer look at LiDAR sensors. Finally, Section 2.4 explores state-of-the-art systems to accelerate the processing of LiDAR-generated point clouds, *i.e.,* to achieve real-time latency.

2.1 Field-Programmable Gate Arrays (FPGAs)

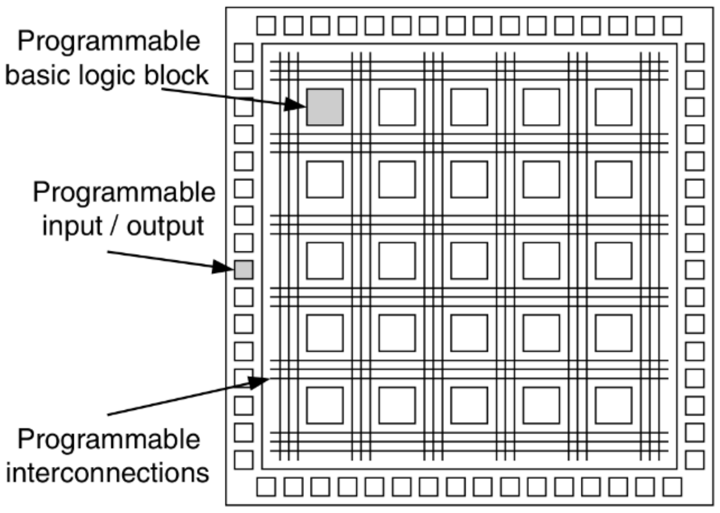
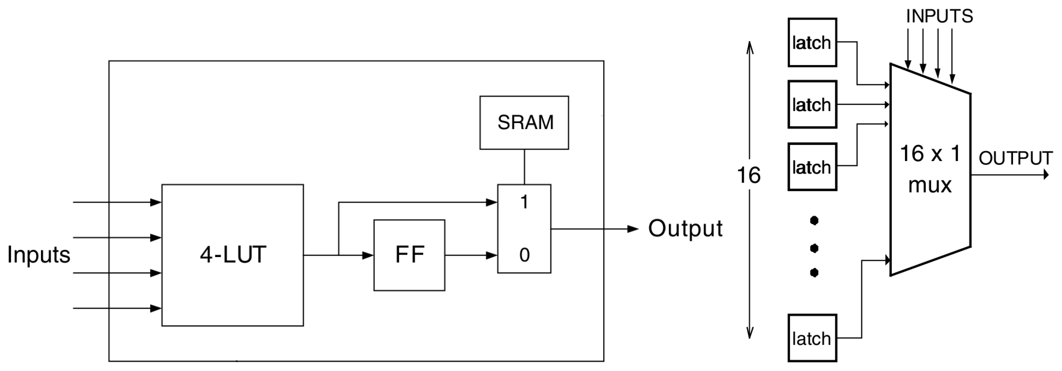

Programmable basic logic block
Programmable input / output
Programmable interconnections

Figure 2.1: **Basic FPGA Architecture** as depicted in [18]. We can describe it as a two-dimensional array of logic blocks and programmable switches, combined with I-/O-cells to access macro cells.



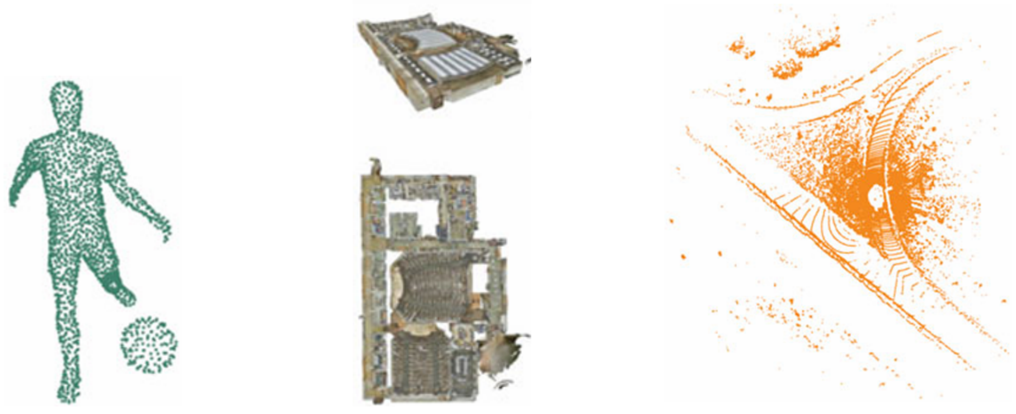
((a)) Logic Cell ((b)) Lookup Table (LUT)

Figure 2.2: **Detail-View of a Logic Cell** as depicted in [18]. (a) shows a single logic cell. The lookup table (LUT) implements combinational logic, while the flip-flop (FF) allows state to be added to the output. (b) depicts a lookup table with four input signals (4-LUT). The inputs select one of $2^4 = 16$ values stored in registers.

A field-programmable gate array (FPGA) is a programmable logic device. We can abstract it as a two-dimensional array of *logic blocks* and *switches*. Each logic block can perform a simple function, while the switches define the interconnections [16]. A logic block contains multiple logic cells that are further subdivided into a lookup table (LUT) and a flip-flop (FF) [50], as depicted in Figure 2.2(a). The LUT implements a boolean function f_B. As depicted in Figure 2.2(b), we can think of it as a $(2^n \times 1)$-multiplexer (MUX) where n denotes the number of control signals (= LUT inputs). The 2^n registers connected to the MUX store the truth table of f_B and are written when the FPGA is programmed [18]. The inputs select which value to output. Thus, the first register stores $f_B(0,0,0,0)$, the second register $f_B(0,0,0,1)$, *etc.* The flip-flop allows adding state to the cell's output, *i.e.,* maintaining it for multiple clock cycles [50].

As a result, each logic block can implement a small part of a circuit. The functionality of FPGAs is completed by macro cells, *e.g.,* memory blocks or specialized circuits for digital signal processing (DSP), clock management, and I/O interface control [16]. FPGAs are thus very flexible and allow the implementation of a wide range of applications [50].

Programming an FPGA means configuring the cells and switches such that the array implements a custom circuit. The circuit is written in a hardware definition language (HDL) and undergoes several processing steps to be converted into a *bitstream file* which is similar to an executable program in software engineering. *Synthesis* breaks down the HDL code and maps these pieces to FPGA resources, *i.e.,* LUTs, flip-flops, memory, or DSP blocks. The main processing steps of implementation are called *place and route*. The first stage assigns the resources elaborated during synthesis to physical locations on the board. The second stage determines how to wire them together. The output of this process is a bitstream file that allows the programming of the FPGA [50].



((a)) ModelNet [71] ((b)) S3DIS [4] ((c)) KITTI [26]

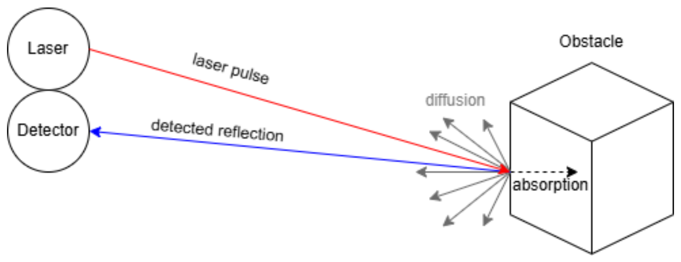
Figure 2.3: **Point Clouds From Three Datasets.** (a) is a synthetic point cloud generated by sampling surface points from CAD-generated meshes. (b) was captured by a 3D Camera that combines 2D and 3D sensors. (c) was collected using a LiDAR sensor.

2.2 Point Clouds

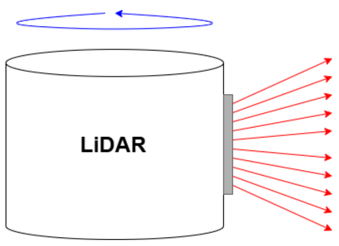
A point cloud is generally defined as a discrete set of points in space [68]. This thesis targets three-dimensional (3D) point clouds. Traditionally, we use an Euclidean space and describe each point's *location* in Cartesian coordinates (x, y, z). The points may further carry *attribute data* such as color information, surface normals, and timestamps [11, 43]. The attributes depend on how the point cloud is generated [43]. Three examples are shown in Figure 2.3. We further distinguish *static* and *dynamic* point clouds. A dynamic point cloud is a sequence of static point clouds, analogous to frames in a video. It represents the continuous change of a point cloud over time [11].

Processing point clouds is considered hard [43]. They lack structure [11] as they are unordered. Unlike images, the data cannot be organized in a regular structure. In addition, they often contain a large number of points while still suffering from sparsity [43]. Processing dynamic point clouds is even more challenging as there is no point-to-point correspondence between successive frames [11].

Despite these challenges, point clouds play an important role in many disciplines, such as 3D computer vision, computer graphics, and photogrammetry. They thus find applications in various domains, *e.g.,* computer-aided design, augmented and virtual reality, or advanced driver assistance systems [43].



((a)) Collecting a Data Point.



((b)) Collecting a Point Cloud.

Figure 2.4: **LiDAR Data Collection**. (a) depicts the collection of a single point. A laser source emits a laser pulse. The pulse travels through air until it hits an obstacle. Some of the emitted energy is reflected and collected by a detector. (b) illustrates the collection procedure for a surround LiDAR sensors. It holds several laser sources pointing in different directions that rotate horizontally around the center of the device.

2.3 LiDAR Technology

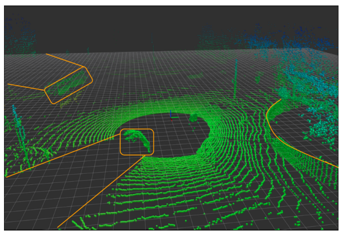
"Light Detection And Ranging" (LiDAR) refers to a class of sensors that combines laser and photoelectric detection technology. It comprises a wide range of devices, and there are several classification criteria, such as the ranging mode or the detection and recording methodology [65]. In this thesis, we use pulsed, discrete return LiDAR sensors. This type is the most widely used in commercial applications [65]. Such a sensor collects a single data point by emitting a laser pulse and detecting the reflected energy [65]. It computes the point of reflection based on the shooting direction and the time of flight [64]. Figure 2.4(a) illustrates this process.

LiDAR sensors scan their environment by emitting laser pulses in several directions. In this thesis, we will use "surround" LiDAR sensors. As depicted in Figure 2.4(b), they hold multiple laser sources, each pointing in a different direction. They are fired one after the other and in a fixed order – in a so-called *firing sequence*. This procedure is repeated periodically while rotating the sources horizontally around the LiDAR's center [64]. The sensor thus generates a dynamic point cloud. Each revolution captures a frame, scanning a 360° field of view. State-of-the-art sensors collect up to 20 frames per second (*e.g.,* [30]) and allow to observe changes in the environment in real time.

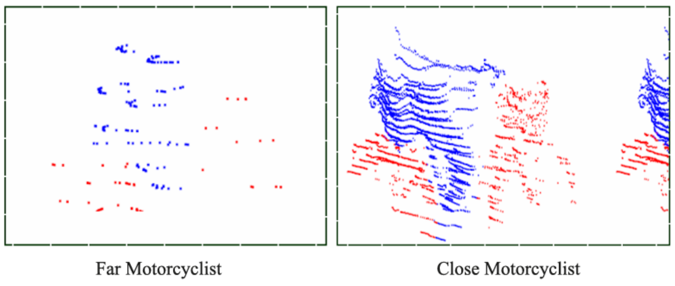
Besides the point's location, many LiDAR sensors also measure the *intensity* of the reflected pulse. Some of the emitted energy gets lost, for example, due to absorption [63]. Some sensors are able to measure the *elongation* "of the laser pulse beyond its nominal width" [63, p. 2449]. This feature may help mitigate spurious obstacle detections caused by bad weather conditions such as dust, fog, or rain. A highly elongated low-intensity return is a strong indicator for this scenario [63].

LiDAR-generated point clouds, however, impose further challenges due to their collection procedure. They suffer from occlusion, which, for example, complicates object tracking [25]. As depicted in Figure 2.5(a), the LiDAR does not capture the scene behind an obstacle. Thus, an object may disappear behind another obstacle for several frames before being visible again. Tracking algorithms should still be able to estimate the object's trajectory.

Also, the resolution of the scan decreases linearly in the radial distance as distinct laser pulses are not emitted in parallel. As a result, the same object is described by a much lower number of points when being captured at a large distance compared to being captured at a small distance.



((a)) Occlusion [8]



((b)) Varying Density [81]

Figure 2.5: **Challenges when Processing LiDAR-Generated Point Clouds.** (a) Parts of the environment are not visible due to occlusion. (b) As neighboring laser pulses do not parallel, the point cloud density decreases linearly with the radial distance.

Notation

Point Attributes: Table 2.1 lists the most common properties of points collected by LiDAR sensors. There are three sets of coordinates: Cartesian, spherical, and LiDAR. The location of a point in a point cloud is usually given in Cartesian coordinates (x, y, z). However, due to the data collection methodology, LiDAR devices often describe the locations in spherical coordinates (r, θ, ϕ). r denotes the radial distance, θ the polar angle, and ϕ the azimuthal angle. The polar angle θ is commonly replaced by the elevation $\omega := 90 - \theta$, which measures the angle between the shooting direction and the x-y plane. This thesis uses the term *LiDAR coordinates* for the triple (r, ϕ, ω).

Property	**Variables**
Cartesian coordinates	(x, y, z)
spherical coordinates	(r, θ, ϕ)
LiDAR coordinates	(r, ϕ, ω)
reflection intensity	i
elongation	e

Table 2.1: **LiDAR Point Properties and Notation**

LiDAR Characteristics: Surround LiDAR sensors capture a 360° horizontal field of view and a vertical field of view restricted to $[\omega_{min}, \omega_{max}]$ with angular resolutions ϕ_{RES} and ω_{RES}. ϕ_{RES} denotes the horizontal angular resolution and denotes the average azimuthal difference between neighboring firing sequences. ω_{RES} denotes the vertical angular resolution and denotes the average polar difference between neighboring channels.

2.4 Point Cloud Processing in Real Time

Despite the challenges faced when processing (LiDAR-generated) point clouds, LiDAR sensors have become increasingly popular in remote-sensing applications that require 3D vision [43]. Furthermore, some of these applications are time-sensitive and may need to process the LiDAR frames in real time. This thesis classifies a system as real-time if it can process all point cloud data at frame rate. For example, if a LiDAR sensor collects 10 frames per second, then the end-to-end latency of a data processing pipeline must be no more than $100\ ms$.

Several systems are claiming to process LiDAR data in real-time, or tackle the acceleration of such a system. We listed a few examples in Table 2.2 and classified them according to the task they perform, the platform they use, and the reported latency. As the table shows, some systems fail to process the data in real time, whereas others neglect pre-processing and only report the latency for ML inference. Moreover, latency is often evaluated on one of the dominating benchmark data sets (*i.e.,* KITTI [26] or nuScenes [10]), which do not provide the raw LiDAR data. For instance, the data stream has already been split up into frames, and the point locations are given in Cartesian coordinates. Nevertheless, we can identify a common pattern: the systems accelerate the processing pipeline with hardware, typically using graphics processing units (GPUs) or application-specific circuits, implemented on an FPGA platform.

GPUs accelerate point cloud processing mainly due to their ability to execute multiple operations simultaneously. Several systems achieve real-time processing speed thanks to powerful GPUs, *e.g.,* [2, 56, 66, 70]. These GPUs, however, cannot run on the edge due to power and thermal constraints. In a real-world application, such a system must transmit the collected LiDAR data to the cloud, costing additional latency. Mobile systems (*e.g.,* vehicles, drones, or robots) may suffer from an unstable network quality and thus cannot rely on such a solution for safety-related applications.

Another approach is to process the data on an embedded device for edge computing with less powerful GPUs such as NVIDIA's Jetson series (*e.g.,* [13, 66, 70]). Wu *et al.* [70] implemented real-time foreground segmentation on an embedded platform. However, the results of McLean *et al.* [49] and Wang *et al.* [66] imply that this approach may not succeed for more complex target tasks such as object detection. For example, RangeSeg [13] can segment point clouds in real time in the "average case" but not in the worst case.

An application-specific circuit is a piece of custom hardware designed to support a specific application and is often used in combination with other hardware *i.e.,* CPUs. For example, Stanisz *et al.* [62] and Latotzke *et al.* [36] both implement a 3D object detection pipeline based on PointPillars [35] and offload parts of the pipeline to specialized hardware kernels. While Stanisz *et al.* miss real-time latency by far ($20\ s/frame$), Latotzke *et al.* can process almost two frames per second despite fusing the output of two high-resolution LiDAR sensors. As most of the processing time ($374\ ms$) is spent on point cloud registration, their approach may be successful in a smaller setup, *e.g.,* when using only one LiDAR sensor, scanning the environment with lower resolution, or restricting the field of view.

Similar systems are found for other tasks, such as clustering (*e.g.,* FCC [82] and Durin [40]) or road segmentation (*e.g.,* ChipNet [46]). As embedded devices, heterogeneous hardware achieves real-time latency for these less complex tasks.

		System	Platform	Reported Latency	
Detection		YOLO-3D [1]	Titan X	inference	41 ms
		FaF [45]	Titan XP	e2e	30 ms
		Complex-YOLO [60]	Titan X	e2e	20 ms
			JETSON TX2		250 ms
		PIXOR [74]	Titan XP	e2e	93 ms
		BirdNet+ [6]	Titan XP	inference	100 ms
		BEVDetNet [52]	JETSON AGX XAVIER	inference	6 ms
		PointPillars (1) [62]	Zynq ZCU104*	e2e	20 s
		PointPillars (2) [36]	Alveo U280*	e2e	522 ms
Segmentation	Semantic	MPF [2]	N/A	inference	49 ms
		RangeSeg [13]	JETSON AGX XAVIER	e2e	57 ms
		CPSeg [37]	TESLA V100	inference	33 ms
		3D-ARSS [66]	GTX 3090	inference	88 ms
	Foreground	SqueezeSeg [70]	Titan X	N/A	14 ms
			Drive PX 2 AutoChaffeur		38 ms
		ChipNet [46]	Kintex XCKU115*	e2e	18 ms
		FuseMODNet [56]	Titan X Pascal	inference	55 ms

Table 2.2: **Overview of LiDAR-Based Systems in the Traffic Domain that Tackle Real-Time Latency or Acceleration**. All systems take 10 frames per second as input. The threshold for real-time latency is thus 100 ms. Note that most systems were evaluated on pre-processed LiDAR data, *i.e.,* frame splitting and coordinate conversion (LiDAR $\rightarrow$ Cartesian) are already performed. * marks platforms that allow the implementation of custom circuits on an FPGA.

3
LiDAR Point Cloud Representations

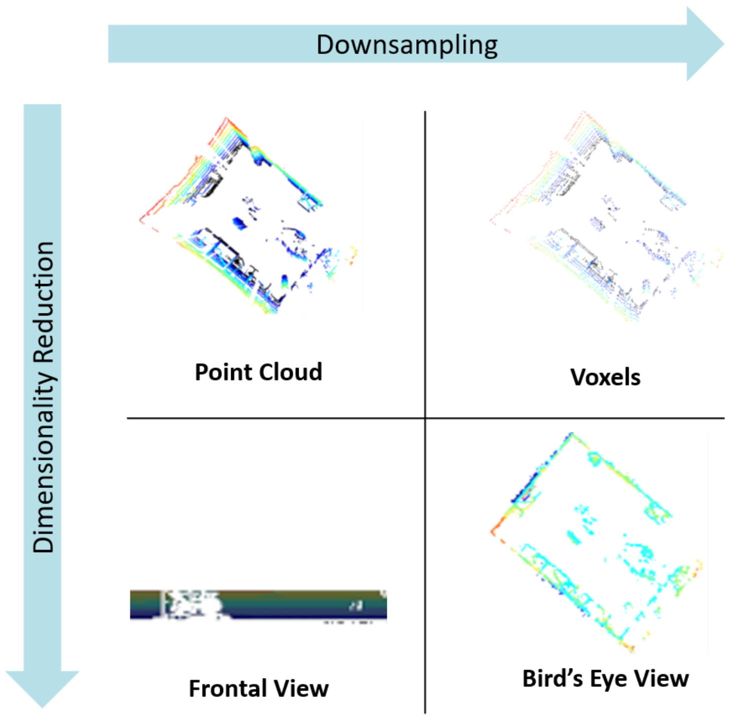


Figure 3.1: **Four Dominating Point Cloud Representations in Traffic-Related Applications.** The associated processing techniques can be summarized by two major paradigms. *Dimensionality reduction* projects the point cloud from 3D to a two-dimensional image plane. *Downsampling* summarizes data, thus reducing the amount of data to be processed later in the pipeline. The Bird's Eye View (BEV) representation combines both paradigms.

This chapter investigates traffic-related systems that process LiDAR-generated point clouds, aiming to identify common processing patterns. We surveyed 50 publications that tackle semantic segmentation, object detection, or closely related tasks. Not surprisingly, most systems utilize a deep neural network (DNN) to solve the target task. However, none of these models operate on the raw data as output by the LiDAR sensor. Therefore, the systems need to pre-process the point cloud, *i.e.,* parsing the LiDAR output into points, splitting the data stream into frames, and building a more suitable input representation for the algorithm. We thus classified the systems according to the target task, the input representation(s), and whether they fuse the point cloud with other data, *i.e.,* RGB images from a camera or GIS data. The table in Appendix A provides a more detailed view of the results

Most of the surveyed systems (42) tackle some variant of object detection, sometimes in combination with tracking (5). Seven systems tackle segmentation tasks, and one system aims at classifying objects after detecting them. Most systems (35) try to solve the target task solely based on LiDAR data, while 15 fuse them with other data. There are 45 systems that rely on a single representation, while the remaining combine up to three. We identified four representations that cover a vast majority (> 80 %) of the systems. They are illustrated in Figure 3.1. The BEV pseudo-image representation (Section 3.4) is the most popular one, used in almost 40 % of the systems. It is followed by voxel (25.5 %, Section 3.3) and FV image representation (23.6 %, Section 3.2), the latter being especially popular in systems tackling segmentation tasks ($5/8$). Eight systems attempt to solve the target task directly on the point cloud (Section 3.1), while another seven systems utilize alternative representations, such as clusters (6) or some other projection of the point cloud (1).

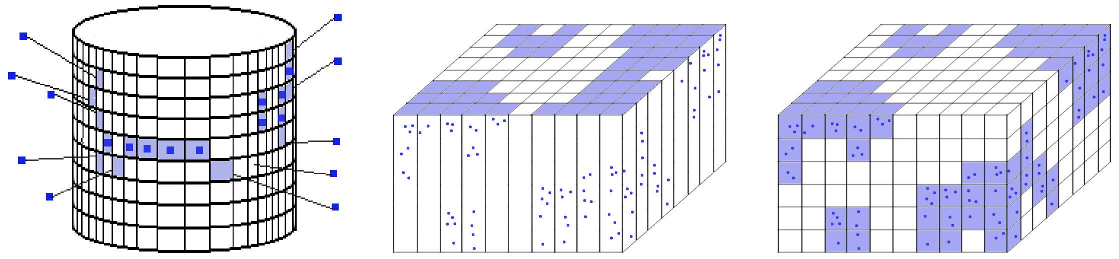
If we only consider the four dominating point cloud representations, we can summarize the associated processing techniques in two paradigms: dimensionality reduction and downsampling. *Dimensionality reduction* projects the point cloud to a two-dimensional image plane. Using such a simplified representation, the system can leverage image processing techniques, *i.e.,* a convolutional neural network, to solve the target task. *Downsampling* reduces the amount of data to be processed later in the pipeline. It is commonly implemented by aggregating data from multiple points. In the upcoming sections, we will briefly discuss each of the four representations and the associated processing techniques.

3.1 Point Clouds

Systems operating directly on point clouds rely on the ability of DNNs to learn how to represent data in a semantically meaningful way. However, such systems still need to pre-process the LiDAR data. For instance, all systems process the frames individually and thus need to split the continuous data stream output by the LiDAR into a sequence of static frames. Moreover, the system must extract the point location and attribute data from the traffic (*i.e.,* parsing). Most systems further operate on Cartesian coordinates and thus compute them based on the coordinate set given by the LiDAR:

$$\begin{aligned} x &= r \cdot \cos\omega \cdot \sin\phi \\ y &= r \cdot \cos\omega \cdot \cos\phi \\ z &= r \cdot \sin\omega \end{aligned} \tag{3.1}$$

Some applications additionally restrict the sensor's field of view to a region of interest. For example, several systems (*e.g.,* [59, 67, 84]) search for objects in the image stream captured by a camera. For each detection, they then investigate the corresponding portion of the point cloud to determine its location more precisely.



((a)) Frontal View (FV) ((b)) Bird's Eye View (BEV) ((c)) Voxelization

Figure 3.2: **Point Cloud Projections for Downsampling and Dimensionality Reduction**. The dark-blue points represent a small point cloud. The light-blue coloring marks occupied pixels or voxels.

3.2 Frontal View Pseudo-Image

A *frontal view pseudo-image* (FV image) projects the points one by one to an image plane located on the lateral surface of a cylinder centered at the LiDAR sensor. Figure 3.2(a) provides a visual intuition of the projection methodology. The pixel coordinates p_x, p_y are computed based on the LiDAR coordinates of the points, *i.e.,* azimuth ϕ and elevation ω:

$$\begin{aligned} p_x &= \frac{\phi}{\phi_{RES}} \\ p_y &= \frac{\omega - \omega_{min}}{\omega_{RES}} \end{aligned} \tag{3.2}$$

The image channels may hold arbitrary information about the corresponding point. Table 3.1 provides an overview. A widespread feature is the distance between the point and the sensor, possibly restricted to the x-y plane. Such a representation is termed a *depth map* and will serve as an anchor example throughout this thesis.

Property	**Feature**	
Location	horizontal depth $d_h = \sqrt{x^2 + y^2}$	[5]
	Cartesian coordinates	[14, 20, 33]
	Spherical coordinates	[13, 14, 20, 33, 51]
Other	reflection intensity	[5, 13, 14, 20, 33, 51]
	occupancy	[13, 33, 51]
	elongation	[20]

Table 3.1: **Overview of FV Image Channel Information**.

3.3 Voxel Representation

Voxelization is a common downsampling technique applied to point clouds [43]. It is computed by projecting the points to a regular 3D grid and building a representation for each cell, as depicted in Figure 3.2(c). Each cell is termed a *voxel*.

Let $x_{min}, y_{min}, z_{min} \in \mathbb{Z}$ denote the lower bounds of the scanned scene along the x, y, and z axis, and $w \in \mathbb{N}_+$ the width of the voxel cells. Then, we can compute the voxel coordinates v_x, v_y, v_z for each point by applying simple integer arithmetic:

$$\begin{aligned} v_x &= \frac{x - x_{min}}{w} \\ v_y &= \frac{y - y_{min}}{w} \\ v_z &= \frac{z - z_{min}}{w} \end{aligned} \tag{3.3}$$

Early systems rely on statistical features to represent the cells. For example, VoxNet [48] explores different cell occupancy representations. More recent systems, however, pursue an end-to-end approach. They stack the points voxel-wise and train a backbone network to learn a suitable encoding function (*e.g.,* [86]). More recent approaches advocate a cylindrical partitioning of the point cloud instead of the traditional cubic voxels. The voxel indices are computed based on (r, ϕ, z) instead of the Cartesian coordinates. This results in a more balanced point distribution, *i.e.,* fewer empty cells [87]. Independent of the partitioning and encoding of the voxels, the resulting representation is more compact and provides more structure than a raw point cloud.

3.4 Bird's Eye View Pseudo-Image

A *bird's eye view pseudo-image* (BEV image) views the scene from above. It combines downsampling and dimensionality reduction by projecting the point cloud to a regular 2D grid on the x-y plane, as depicted in Figure 3.2(b). As the cells are not restricted along the z-axis, the cells are referred to as *pillars*. The pillar coordinates (p_x, p_y) are computed in the same way as the voxel indices in Equation 3.3, *i.e.,* $p_x := v_x,\ p_y := v_y$.

Each pillar maps to a pixel, while the image channels represent the associated points. As for voxels, the representation can be learned in an end-to-end approach (*e.g.,* [12, 35]) or computed by extracting statistical features for each pillar as listed in Table 3.2. As BEV images closely relate to voxels, we often find hybrid representations in the form of "slice-wise" features in BEV images. Such features are computed by dividing the pillar along the z-axis into regular slices and extracting a given attribute for each slice.

Property	**Feature**	
Height	maximal height	[6, 7, 42, 52, 60]
	slice-wise maximal height*	[14, 34]
Density & Distribution	normalized density	[6, 7, 14, 34, 42, 60]
	occupancy bit	[52]
	slice-wise occupancy bits*	[42, 45, 74]
Reflection Intensity	mean reflection intensity	[6, 7, 42, 52, 74]
	I of the highest point	[14, 60]

Table 3.2: **Overview of Statistical BEV Features.** * marks voxel-like features.

4 DAMOCLES

This chapter introduces DAMOCLES, a formal data model to express point cloud manipulations. It is highly modular and thus allows the expression of complex manipulations by combining simple operators. This chapter first introduces the underlying data representation (Section 4.1) and then describes the basic operators. As previously discussed, LiDAR sensors produce dynamic point clouds. The surveyed applications, however, process the frames separately. Moreover, LiDAR sensors typically generate a constant data stream and are oblivious to frames. The first class of operators (4.2) thus operate on the steady data stream and splits it into a sequence of frames. The second class of operators (4.3) manipulates one frame at a time. The last class of operators (4.4) manipulates the elements in a representation individually, *e.g.,* they compute a new feature for all points in a point cloud. We will complete this chapter by discussing two motivating examples in Section 4.5.

4.1 Data Representation and Notation

Chapter 3 discussed different ways of representing a point cloud. Each representation uses a primary element to describe the point cloud, *i.e.,* points, voxels, pillars, or clusters. All elements in a representation are described by the same set of features. We formalize this idea by mapping a point cloud representation to a real $n \times m$ matrix, *e.g.,*

$$C := \begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}, c_{ij} \in \mathbb{R}.$$

n denotes the number of elements and m the number of features. As most systems process the frames individually, a representation C will typically hold elements belonging to the same frame. Each row is a feature vector describing one of the representations' primary elements, while the column implies the semantics of a feature. Thus, $c_{11}, c_{21}, \dots c_{n1}$ describe the same attribute for different elements. For example, a standard point cloud will hold the $x,\ y$, and z coordinates in columns 1 to 3, and the reflection intensity i in the fourth column; or a depth map holds the pixel coordinates p_x, p_y in the first two columns and the radial distance r in the third column.

Applying DAMOCLES expressions to a given representation C_{in} transforms it into another representation C_{out}. In general, the expressions operate on the columns of C. We will thus use the index $[i]$ to refer to the i-th column. Thus, we can rewrite C as a row of column vectors, *i.e.,* $C = [[1], [2], \ldots, [m]]$. We may also use the feature semantics to describe the representation if they are known. For example, $C := [X, Y, Z, I]$ denotes a point cloud where each point is described by its Cartesian coordinates and reflection intensity. This notation is especially useful to describe the input and the output of DAMOCLES expressions.

4.2 Stream-Level Operators

Stream-Level operators manipulate the steady data stream $\mathbf{D}$ produced by the LiDAR sensor. The main objective of these operators is to convert it into a sequence of static LiDAR frames. We formalize the entirety of stream-level operators by a single operator

$$\mathcal{S} : \mathbf{D} \mapsto C_{in,1}, C_{in,2}, \ldots \; .$$

$\mathcal{S}$ depends on the LiDAR device, or more precisely, on the payload format of its output. It involves extracting the individual measurements from the payload (*parsing*) and a heuristic to draw the boundary between two frames (*frame splitting*). The frames $C_{in,j}$ are static point clouds with the same set of columns, *i.e.,* all points are described by the same set of features in the same order. However, the frames may differ in size, meaning that the number of rows may vary.

4.3 Frame-Level Operators

Most of DAMOCLES' frame-level operators are derived from linear algebra, as depicted in Table 4.1. Note that relational algebra operates on relations (*e.g.,* a set of tuples) and the rows are thus unordered. DAMOCLES operates on matrices. Thus, it preserves the order of rows whenever possible, *i.e.,* when performing projection or selection.[1] Another difference between the two models is that the columns of a relation are labeled, while we use indices $[F_i]$ to refer to the column of a point cloud representation.

	Relational Algebra	**DAMOCLES**
Projection	$\pi_{A_{j_1}, A_{j_2}, \ldots, A_{j_k}}(R)$	$\mathcal{P}_{[F_1, F_2, \ldots, F_k]}(C)$
Selection	$\sigma_\theta(R)$	$\mathcal{F}_\phi(C)$
Aggregation	$\Gamma(R, (A_{j_1}, A_{j_2}, \ldots, A_{j_k}), \text{agg}, A_x)$	$\mathcal{A}(C, [F_1, F_2, \ldots, F_k], \text{agg}, [F_x])$
Join*	$R \bowtie_\theta S$	$C \bowtie_\tau C'$

Table 4.1: **Relational DAMOCLES Operators.** Relational expressions are applied to relations R, S where R holds m attributes $(A_1, A_2, \ldots, A_m)$. The attributes $\{A_{j_1}, A_{j_2}, \ldots A_{j_k}, A_m\} \subseteq \{A_1, A_2, \ldots, A_m\}$. DAMOCLES expressions are applied to Point Clouds C, C' where C holds m columns and all indices $F_1, F_2, \ldots, F_k, F_x \in \{1, 2, \ldots, m\}$.

[1]However, aggregation defines a new order as it transforms the primary element of the representation (*e.g.,* points $\rightarrow$ voxels).

Projection, implemented by the operator $\mathcal{P}_\gamma$, allows the selection and ordering of the columns in C. γ is a list of column indices $[F_1, F_2, \dots F_k]$ where $F_i \in \{1, 2, \dots, m\}$. The column selector $\mathcal{P}$ will output the specified columns in the given order, thereby preserving the order of the rows.

Selection refers to removing elements (= rows) from a representation based on some criterion. DAMOCLES implements selection by the means of a filter operator $\mathcal{F}_\phi$. ϕ is a predicate to constrain the attribute domains. Let $e_1, e_2, \dots, e_n$ denote the elements in a point cloud representation C. Then $\mathcal{F}_\phi$ removes any element e_i from the representation that does not satisfy ϕ.

The **aggregation** operator $\mathcal{A}$ is similar to the relational GroupBy operator Γ. It allows to define the set-membership of each element by one or multiple indices $[F_1, \dots, F_k]$. For each partition, it then evaluates an aggregation function $\mathrm{agg} : \mathbb{R}^* \to \mathbb{R}$ over the column indexed by F_x. Typical aggregation functions are $\{\mathrm{min}, \mathrm{max}, \mathrm{mean}, \mathrm{var}, \mathrm{count}\}$.

Finally, DAMOCLES uses an **equi-join** $\bowtie_\tau$ to combine the outputs of several aggregators into a single point cloud representation. τ holds a list of bi-tuples $[F_i, F_j]$ defining which columns to match for the join. For example, let $C := [G_1, G_2, A]$ and $C' := [G_1, G_2, A']$ denote the outputs of two aggregators that applied aggregation on the same input representation C_{in} using the same set of index features $[G_1, G_2]$. Then,

$$C \bowtie_{([1,1],[2,2])} C' := [G_1, G_2, A, A']$$

holds the index g_{j_1}, g_{j_2} and the corresponding values a_j, a'_j for each group.

Additionally to the relational operators, we define the **zip** operator $\|$ to concatenate two point cloud representation. For example, let $C \in \mathbb{R}^{n \times m}$ and $C' \in \mathbb{R}^{n \times k}$. Then

$$C \;\|\; C' = \begin{bmatrix} c_{11} & \dots & c_{1m} & | & c'_{11} & \dots & c'_{1k} \\ c_{21} & \dots & c_{2m} & | & c'_{21} & \dots & c'_{2k} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} & | & c'_{n1} & \dots & c'_{nk} \end{bmatrix}.$$

Although the zip operator can concatenate any two matrices C, C' with the same number of rows, we should only use it when the rows refer to the same element, *i.e.,* to append the output of an element-level operator to the representation.

4.4 Element-Level Operators

The third family of operators can be applied individually to the elements (rows), *i.e.,* to compute new features and annotate the representation. DAMOCLES provides several operators to compute new features. They all take a point cloud representation as input and output one new feature in a column vector. However, they differ in the remaining input parameters and the function type they apply to the point cloud. An overview of the operators' functional scopes and a few examples are listed in Table 4.2.

A **unary operator** $\mathcal{U}$ applies a unary function $u : \mathbb{R} \to \mathbb{R}$ element-wise to a given column $[F]$ of a point cloud representation C:

$$\mathcal{U}(C, [F], u) = \begin{bmatrix} u(c_{1F}) \\ u(c_{2F}) \\ \vdots \\ u(c_{nF}) \end{bmatrix}.$$

Examples for u are trigonometric functions and their inverses, the logarithm, drawing the square root, or rounding. However, the unary operator may support any real, unary function.

A **binary operator** applies a binary function $b : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ element-wise to a point cloud representation. DAMOCLES distinguishes two types of binary operators. First, the constant binary operator $\mathcal{B}_{const}$ that takes a column index $[F]$ and a constant value $\bar{c} \in \mathbb{R}$ as inputs. The second argument of b is thus constant, *i.e.,*

$$\mathcal{B}_{const}(C, [F], \bar{c}, b) := \begin{bmatrix} b(c_{1F}, \bar{c}) \\ b(c_{2F}, \bar{c}) \\ \vdots \\ b(c_{nF}, \bar{c}) \end{bmatrix}.$$

Second, the non-constant binary operator $\mathcal{B}_{ftr}$ that takes two column indices $[F, G]$ as inputs and applies b to the corresponding values, *i.e.,*

$$\mathcal{B}_{ftr}(C, [F, G], b) := \begin{bmatrix} b(c_{1F}, c_{1G}) \\ b(c_{2F}, c_{2G}) \\ \vdots \\ b(c_{nF}, c_{nG}) \end{bmatrix}.$$

Examples for binary functions are the numerical operators or the two-argument arc tangent $\arctan2$.

	Scope	Functions	
Unary	$\mathbb{R} \to \mathbb{R}$	sine	$\sin$
		cosine	$\cos$
		arcsine	$\arcsin$
		logarithm	$\log$
		square root	$\sqrt{\cdot}$
Binary	$\mathbb{R}^2 \to \mathbb{R}$	addition	add
		subtraction	sub
		multiplication	mul
		division	div
		two-argument arctangent	$\arctan2$

Table 4.2: **Overview of Surveyed Feature-Extraction Functions**. The table lists all functions that we found in the survey and assigns them to one of the feature-extraction types of the data model. However, the model can be extended to functions that match any of the scopes.

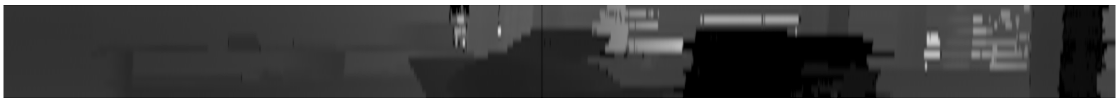


Figure 4.1: **Depth Map of a Point Cloud Generated by a VLP-16 LiDAR Sensor** [64]. The field of view was restricted to an azimuth range of approximately $120°$, and the image was stretched vertically by a factor of six using cubic interpolation.

4.5 Examples

This section introduces two relevant examples that we will use throughout the thesis. The first is a depth map, an FV pseudo-image (Section 3.2) storing the radial distance r for each pixel, as depicted in Figure 4.1. The second representation is a height map, a BEV pseudo-image (3.4) storing each pixel's maximal height $\max(z)$, see Figure 4.2. For the sake of simplicity, we operate in the integer domain $\mathbb{Z}$ for now. Thus, all features (input, intermediary, and output) are integers, *i.e.,* the result of a division is rounded down. Recall that all indices start at one.

Example 1: Depth Map

Let $C_{in} = [R, \Phi, \Omega, I]$ denote a single LiDAR frame where each point is described by its LiDAR coordinates (r, ϕ, ω) and the reflection intensity i. We aim at constructing the output representation $C_{out} = [P_x, P_y, R]$ holding pixel coordinates and the radial distance for each point. The pixel coordinates are computed using Equation 3.2. We can thus compute C_{out} by applying a sequence of unary and binary operators to the point cloud, appending the new features to it, and performing feature selection in the end:

$$C_{out} = \mathcal{P}_{[6,7,1]}\Big(C_{in} \parallel \overbrace{\mathcal{U}(\mathcal{B}_{const}(C_{in}, [2], \phi_{RES}, \text{div}), [1], \text{floor})}^{P_x} \parallel \overbrace{\mathcal{U}(\mathcal{B}_{const}(\underbrace{\mathcal{B}_{const}(C_{in}, [3], \omega_{min}, \text{sub})}_{\Omega'}, [1], \omega_{RES}, \text{div}), [1], \text{floor})}^{P_y}\Big)$$

Let's break up the expression into several steps and slightly reorder the computation steps for the sake of simplicity. First, we compute $\omega' = \omega - \omega_{min}$ directly from C_{in}:

$$\begin{aligned}\Omega' &= \mathcal{B}_{const}(C_{in}, [3], \omega_{min}, \text{sub})\\ C_1 &= C_{in} \parallel \Omega'\end{aligned}$$

C_1 now holds the features $[R, \Phi, \Omega, I, \Omega']$. In the next step, we compute the pixel coordinates p_x, p_y:

$$\begin{aligned}P_x &= \mathcal{B}_{const}(C_1, [2], \phi_{RES}, \text{div})\\ P_y &= \mathcal{B}_{const}(C_1, [3], \omega_{RES}, \text{div})\\ C_2 &= C_1 \parallel P_x \parallel P_y\end{aligned}$$

C_2 holds the features $[R, \Phi, \Omega, I, \Omega', P_x, P_y]$ for each point. To build C_{out}, we just have to select the desired features:

$$C_{out} = \mathcal{P}_{[6,7,1]}(C_2)$$

C_{out} lists all non-empty pixels in the depth map.

Example 2: Height Map

Let $C_{in} = [X, Y, Z, I]$ denote a single LiDAR frame where each point is described by its Cartesian coordinates and the reflection intensity. We aim at constructing a height map $C_{out} = [P_x, P_y, Z_{max}]$, a one-channel BEV image holding the maximum height of each pillar. In other words, we aim at computing the pixel coordinates $p_x := v_x,\ p_y := v_y$ using equation 3.3; then group the points per pixel and extract the maximum height value ($\max(z)$) for each non-empty pillar. We further restrict the sensor's horizontal field of view to a square centered at the LiDAR, *i.e.,* $x, y \in [-r_{max}, r_{max}],\ r_{max} > 0$ and "pillarize" the point cloud with resolution Δ. For this representation, we need to apply most of the discussed DAMOCLES operators to the point cloud:

$$\begin{aligned}
C_{out} = \mathcal{A}\Big(&\mathcal{F}_{[1]\geq -r_{max} \wedge [1]\leq r_{max} \wedge [2]\geq -r_{max} \wedge [2]\leq r_{max}}\big(\\
&\quad C_{in} \parallel \\
&\quad \overbrace{\mathcal{B}_{const}(\mathcal{B}_{const}(C_{in}, [1], -r_{max}, \text{sub}), [1], \Delta, \text{div})}^{P_x} \parallel \\
&\quad \overbrace{\mathcal{B}_{const}(\mathcal{B}_{const}(C_{in}, [2], -r_{max}, \text{sub}), [1], \Delta, \text{div}, [1])}^{P_y}\big), \\
&\qquad [7, 8], \max, [3])\Big)
\end{aligned}$$

As for the previous example, we can break this large expression into smaller, more comprehensible steps. We start by computing the pixel indices p_x, p_y using a sequence of arithmetic operations:

$$\begin{aligned}
X' &= \mathcal{B}_{const}(C_{in}, [1], -r_{max}, \text{sub}) \\
Y' &= \mathcal{B}_{const}(C_{in}, [2], -r_{max}, \text{sub}) \\
C_1 &= C_{in} \parallel X' \parallel Y' \\
\\
P_x &= \mathcal{U}(\mathcal{B}_{const}(C_1, [5], \Delta, \text{div}), [1], \text{floor}) \\
P_y &= \mathcal{U}(\mathcal{B}_{const}(C_1, [6], \Delta, \text{div}), [1], \text{floor}) \\
C_2 &= C_1 \parallel P_x \parallel P_y
\end{aligned}$$

Now $C_2 := [X, Y, Z, I, X', Y', P_x, P_y]$ holds all features we need for aggregation. However, we want to filter out all points that do not lie in our region of interest. Thus, we apply

$$C_3 = \mathcal{F}_{[1]\geq -r_{max} \wedge [1]\leq r_{max} \wedge [2]\geq -r_{max} \wedge [2]\leq r_{max}}(C_2).$$

Now we build C_{out} by applying aggregation to C_3:

$$C_{out} = \mathcal{A}(C_3, [7, 8], \max, [3]).$$

C_{out} lists the maximal height for all non-empty pixels in the height map.

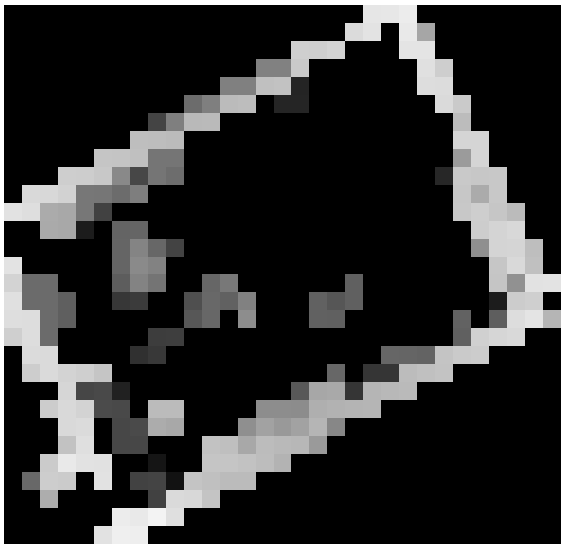


Figure 4.2: **Height Map of a Point Cloud Generated by a VLP-16 LiDAR Sensor**[64]. A $6\ m \times 6\ m$ region of interest ($r_{max} = 3\ m$) was pillarized with a resolution $\Delta = 20\ cm$.

5 PALICUS

This chapter introduces PALICUS, our hardware architecture designed to accelerate DAMOCLES expressions on the edge. Figure 5.1 depicts the pipeline's high-level architecture. It is connected to a LiDAR sensor and a host application through a physical LAN. The core of PALICUS is a so-called *LiDAR Processing Unit* (LPU), a piece of programmable hardware that can implement DAMOCLES expressions in an ad-hoc fashion. The design is completed by modules responsible for networking, parsing, and serialization. They are responsible for (1) connecting the pipeline to the network, (2) extracting the point cloud from the LiDAR data stream, and (3) wrapping the computed point cloud representation back into UDP packets. Besides the data pipeline, PALICUS also provides a configuration parser module. As we will discuss in Section 5.5, we can configure a pipeline via UDP. The configuration parser extracts the "PALICUS program" from the payload and writes it to the corresponding registers.

This chapter starts by introducing all modules associated with the data processing pipeline, beginning with the networking modules (5.1) and the parsing module (5.2). Section 5.3 then has a closer look at the LPU, and Section 5.4 describes the serializer module and the payload format of PALICUS' output data stream. Finally, Section 5.5 discusses how to program PALICUS and discusses the examples from Section 4.5.

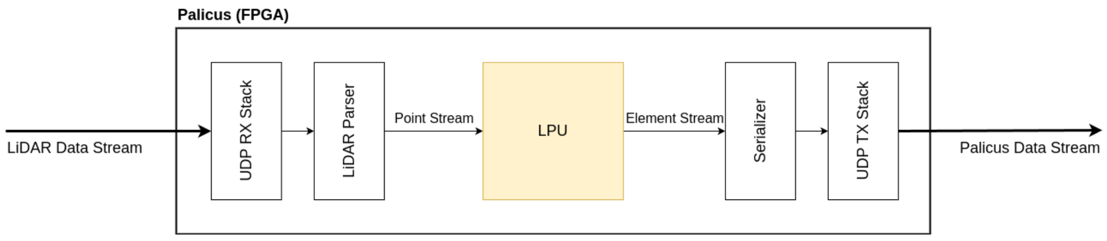


Figure 5.1: **High-Level View of PALICUS' Data Processing Architecture.**

5.1 Networking Modules

PALICUS holds two modules to handle the incoming and outgoing network traffic, one for the RX and one for the TX path. Both stacks are built from Corundum [22] modules, an open source library for an FPGA-based NIC. The *RX stack* retrieves the incoming data packets, deserializes its header values (ETH, IP, and UDP), and outputs the UDP payload byte-wise in an AXI-Stream interface.[1] PALICUS uses the incoming packets' IP addresses and UDP port specifications to route them internally. Packets associated with the LiDAR data traffic are input into the data pipeline, whereas packets related to a configuration message from the host are forwarded to the configuration parser. All other packets are dropped.

The *TX stack* accepts the header values in parallel and the UDP payload as an AXI-Stream and serializes all data into a valid network packet. In the current implementation, some of the header values are fixed, *i.e.,* in the form of architectural parameters that must be defined before synthesizing the design. Other values are computed on the fly, *i.e.,* the checksum and length values at the various levels.

Both the RX and the TX module include a FIFO for incoming and outgoing packets. This allows for handling backpressure up to a certain limit. For example, if the incoming packets are not processed immediately, they are stored in the FIFO. However, if an incoming packet does not fit in the FIFO, it is dropped, in accordance with the UDP procotol.

5.2 Parser Module

The parsing module parses the payload of LiDAR data packets into a point stream. The payload structure is often sensor-specific, although most manufacturers define their own standard. We thus decided to support a specific device for now: a velodyne VLP-16 surround LiDAR sensor [64].

Our parsing module accepts the UDP payload in an AXI-Stream interface and produces a point cloud in several steps. First, it de-serializes the incoming data and retrieves all provided information for each measurement: azimuth, channel number, radial distance, and reflection intensity. In a second step, it annotates each point, *i.e.,* it replaces the channel number by the elevation and computes the Cartesian coordinates (Eq. 3.1). At this stage, the parser produces a point stream where each point is described by $C_{in} := [X, Y, Z, R, \Phi, \Omega, I]$. The features are represented with different precision; the numbers for our prototype are provided in Table 5.1.

In the last stage of the parser module, the point stream is split into frames and *quarter frames*. The frame boundary is set at $\phi = 0°$. Quarter frames further divide each frame into four quarters based on the azimuth, *i.e.,* $q_0 := [0°, 90°), q_1 := [90°, 180°), q_2 := [180°, 270°), q_3 := [270°, 360°)$. The point stream is annotated with a `next_frame` and a `next_quarter` signal that are only set for the first point of a new frame or quarter, respectively. The reason to use quarter frames will become clear when we discuss the implementation of aggregation.

Feature	Precision
x, y, z	$4\,mm$
r	$2\,mm$
ϕ	0.01°
ω	1°
i	1

Table 5.1: **Feature Precision of PALICUS' Input Representation**.

[1]AXI-Stream is an interface standard to transmit serialized data.

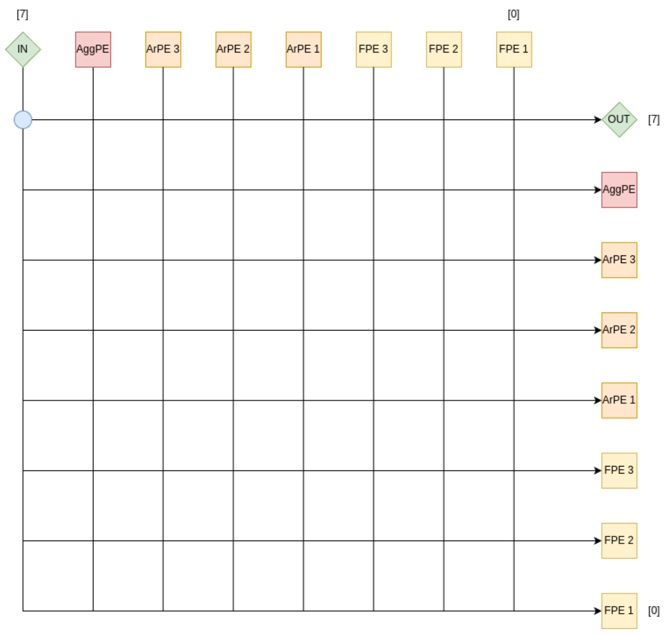


Figure 5.2: **PALICUS' LPU Architecture**. The LPU comprises 3 filtering processing elements (yellow), 3 arithmetic processing elements (orange), and 1 aggregation processing element (red), as well as an input and an output (green). The PEs are duplicated for clarity, the top row illustrating their output and the right column their input. The blue dot in the top left corner of the crossbar symbolizes an active cross point. Thus, the input is forwarded to the output without being processed.

5.3 LiDAR Processing Unit (LPU)

The LPU module takes the point stream from the parser as input and feeds it to the configured pipeline. The pipeline consists of an arbitrary number of processing elements (PEs), each applying a small DAMOCLES expression to the incoming elements. By applying the same expression to all points in a LiDAR frame, the LPU transforms the input $C_{in} := [X, Y, Z, R, \Phi, \Omega, I]$ into a different representation C_{out}, delivered as an *element stream*.

There are three types of PEs, each able to implement a family of DAMOCLES expressions: (1) filter processing elements (FPEs), (2) arithmetic processing elements (ArPEs), and (3) aggregation processing elements (AggPEs). We can configure the PEs independently from each other. We further use a crossbar as a scheduling approach to arrange them arbitrarily in a pipeline.

LPU Architecture

As depicted in Figure 5.2, PALICUS' LPU holds three FPEs, three ArPEs, and one AggPE. These are architectural parameters and may be changed before synthesizing the design.[2] They are arranged in a crossbar such that the output of a processing element can be input into any other processing element, including itself, or linked to the LPU's output. The routes are set by activating a subset of the crossbar's cross points. A valid configuration creates a path from the input to the output of the LPU. The path may include an arbitrary number of PEs in an arbitrary order, but should not contain any loops.

[2]An overview of all architectural parameters is provided in Table 5.2.

To stack the processing elements in an arbitrary order, their interfaces need to match, as depicted in Figure 5.3. The horizontal ports relate to the pipeline. The valid-ready handshake signals control the data flow between the processing elements; and also between the modules in the data pipeline (parser – LPU – serializer). As the modules and processing elements process the data at different rates, the pipeline may suffer from backpressure. The handshake signals ensure that none of the data gets lost. An element is considered transmitted whenever both signals are set at the rising edge of the clock.

The `next_frame` and `next_quarter` signals handle frame and quarter frame splitting, as previously described in Section 5.2. Finally, the data ports carry the elements, each described by up to $m = 7$ features of $f = 16$ bits. The size and format of an element representation on the crossbar are thus fixed.

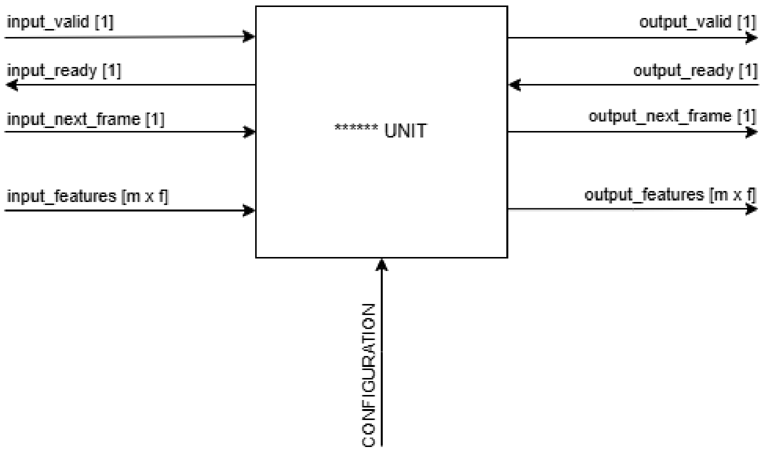


Figure 5.3: **Interfaces of an Arbitrary Processing Element**. The horizontal ports relate to the data pipeline and are equal for all types. The format of the configuration (bottom input) is type-dependent.

Maximum Number of Features per Element	m	7
Feature Width	f	16
Number of FPEs		3
Cells per FPE	$\lvert\mathbf{F}\rvert$	6
Number of ArPEs		3
Cells per ArPE	$\lvert\mathbf{A}\rvert$	3
Number of AggPEs		1
Maximum Number of Group Indices per AggPE	$\lvert\mathbf{I}\rvert$	3
Number of Memory Blocks per AggPE		256

Table 5.2: **Architectural Parameters of PALICUS' LPU**. The number of processing elements and their size were chosen heuristically based on the findings in our survey. For instance, we need at least two ArPEs with three cells to compute voxel indices. The third ArPE may then be used to manipulate the aggregated representation (*e.g.,* height $h := \max(z) - \min(z)$).

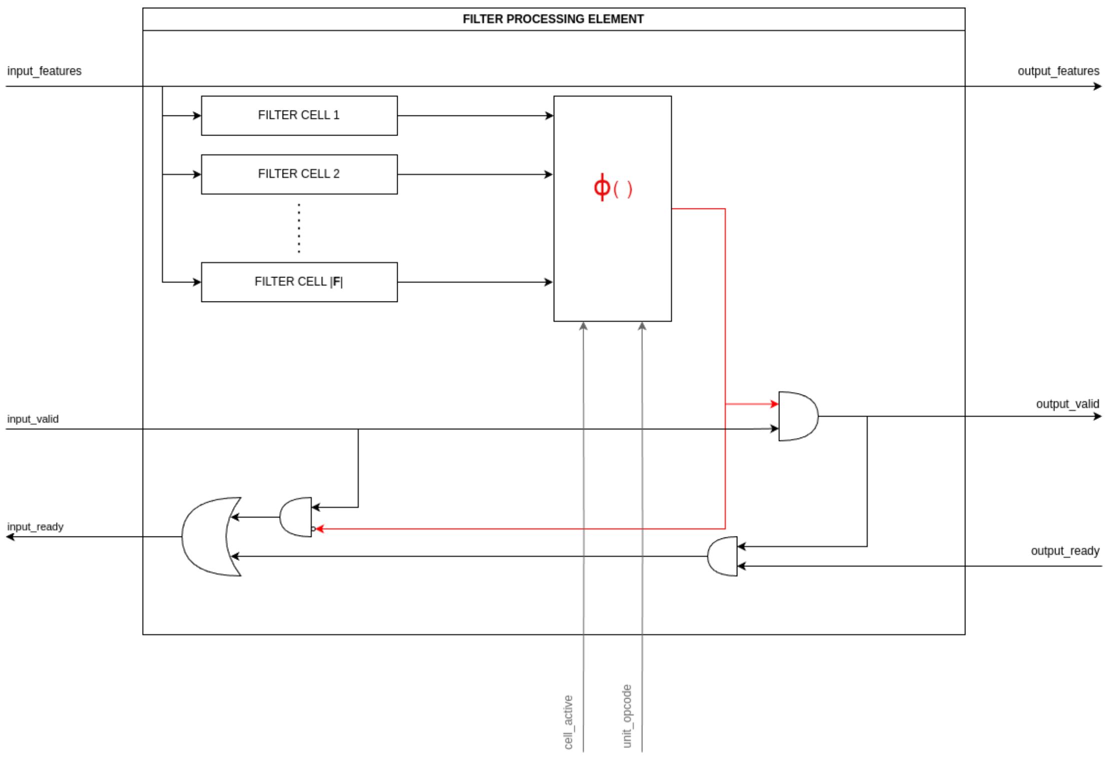


Figure 5.4: **Simplified Schematics of a Filter Processing Element (FPE)**. We omit the cell configuration and some of the flow control signals for simplicity. Filtering mainly operates on the ready-valid handshake signals. The FPE forwards an element if and only if $\phi()$ outputs TRUE. Otherwise, it drops the element.

Filtering Processing Element (FPE)

A filter processing element (FPE) implements a filter operator $\mathcal{F}_\phi$. The predicate $\phi = t_1 \circ t_2 \circ \ldots \circ t_k$ combines k terms t_j with a single operator $\circ \in \{OR, AND, NOR, NAND\}$. Each term $t_j := [F_j] \bullet_j R_j$ restricts the attribute range of a feature F_j by a constant R_j where $\bullet_j \in \{<, \leq, =, \neq, \geq, >\}$.

Figure 5.4 depicts a simplified scheme of an FPE. It holds $|\mathbf{F}|$ filter cells that each evaluate one term t_j. $|\mathbf{F}|$ denotes the size of the FPE and restricts the number of terms it can evaluate. In our implementation, $|\mathbf{F}| = 6$; however, this architectural parameter can be adjusted before synthesizing the design.

The cell configuration comprises an activity flag and the term, defined by the feature index (F_j), a cell opcode $(\bullet_j)$, and a domain constant (R_j). Only active cells contribute to the evaluation of ϕ. The FPE configuration comprises all cell configurations plus the global opcode specifying $\circ$.

Filtering removes all elements from the point cloud representation that do not meet the predicate ϕ. This is implemented by controlling the handshake signals. If the current input element e satisfies the predicate ϕ, *i.e.*, $\phi(e) = \top$, the FPU sets the `output_valid` signal and waits for the successful transmission of e. Otherwise, it drops the element by setting `input_ready` without forwarding it.

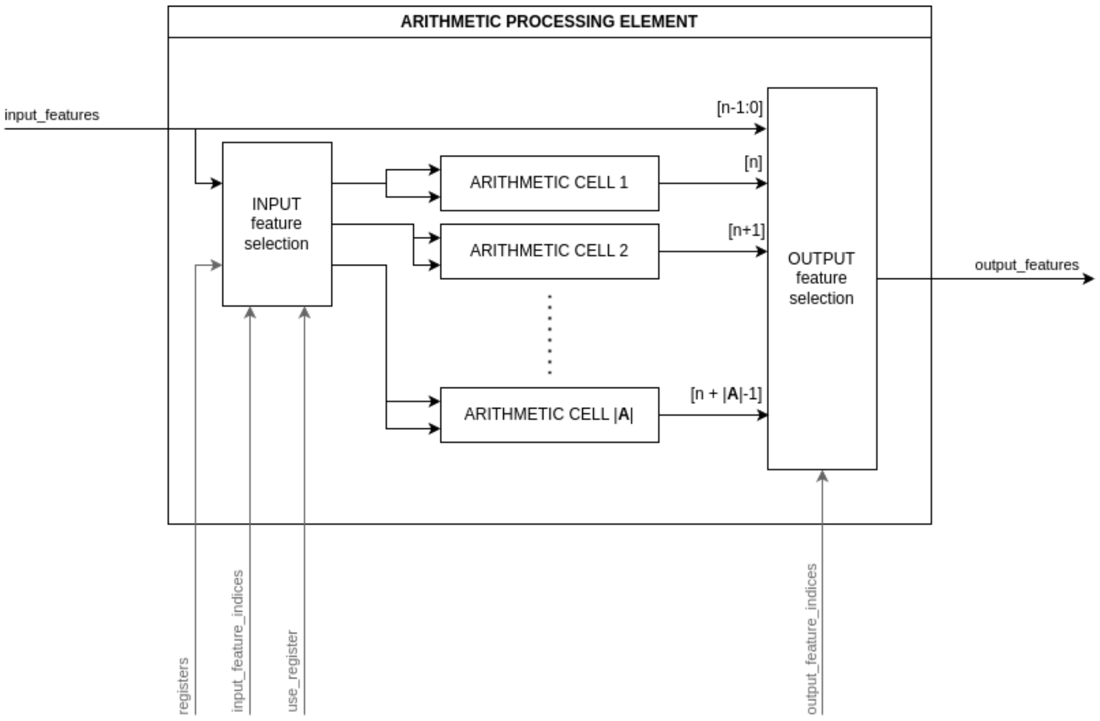


Figure 5.5: **Simplified Schematics of an Arithmetic Processing Element (ArPE)**. We omit the flow control signals and the cell configuration for simplicity. Each arithmetic cell computes a new feature by applying a binary function b to its inputs. The ArPE then selects n features to output.

Arithmetic Processing Element (ArPE)

An arithmetic processing unit (ArPE) implements a DAMOCLES expression of the form

$$\mathcal{P}_{[F_1,F_2,\ldots,F_7]}(C \parallel B_1 \parallel B_2 \parallel \ldots \parallel B_{|\mathbf{A}|}),$$

where B_j is the output of a binary operator $\mathcal{B}$ applied to C. $|\mathbf{A}|$ denotes the size of the ArPE and restricts the number of new features it can compute in parallel. In our implementation, $|\mathbf{A}| = 3$; but as for the size of the FPE, this is an architectural parameter.

Figure 5.5 depicts a simplified scheme of an ArPE. Each binary operator is mapped to an *arithmetic cell*. Each cell takes two input features and applies a binary function $b_j \in \{\text{add}, \text{sub}, \text{div}\}$ to them. ArPEs support both types of binary operators ($\mathcal{B}_{ftr}$ and $\mathcal{B}_{const}$). For that purpose, the cell-level configuration includes two feature indices (F_{j1}, F_{j2}), a register holding a constant R_j, and a flag (`use_register`) to switch between the two types of binary operators. The ArPE selects the input features for each cell accordingly. If `use_register` is set, it connects the input feature indexed by F_{j1} and R_j to the cell. Otherwise, it forwards the two input features indexed by F_{j1} and F_{j2}. Each cell is further configured by an opcode specifying the binary function b_j.

Besides the individual cell configurations, the run-time configuration for an arithmetic unit comprises $n = 7$ output features. We use the indices $\{0, 1, \ldots, n-1\}$ to select on of the ArPE's input features and $n + j - 1, j \in \{1, 2, \ldots, |A|\}$ for the output of the $j - th$ cell.

Aggregation Processing Element (AggPE)

An aggregation processing element (AggPE) implements a DAMOCLES expression of the form

$$
\begin{aligned}
\mathcal{P}_{[1,2,\ldots,|\mathbf{I}|,F_1,\ldots,F_{(m-|\mathbf{I}|)}]}(A_1 &\bowtie_{([1,1],\ldots,[|\mathbf{I}|,|\mathbf{I}|])} \\
A_2 &\bowtie_{([1,1],\ldots,[|\mathbf{I}|,|\mathbf{I}|])} \\
\ldots &\bowtie_{([1,1],\ldots,[|\mathbf{I}|,|\mathbf{I}|])} \\
A_{m-k} &\bowtie_{([1,1],\ldots,[|\mathbf{I}|,|\mathbf{I}|])} \\
COUNT&),
\end{aligned}
$$

where

$$
\begin{aligned}
A_j &:= \mathcal{A}(C, [G_1, G_2, \ldots, G_{|\mathbf{I}|}], \mathrm{agg_j}, G_{x_j}), \\
COUNT &:= \mathcal{A}(C, [G_1, G_2, \ldots, G_{|\mathbf{I}|}], \mathrm{count}, *), \\
G_{x_j}, G_c &\in \{1, 2, \ldots, m\} \\
F_j &\in \{1, 2, \ldots, m - |\mathbf{I}|, m - |\mathbf{I}| + 1\} \\
agg_j &\in \{\max, \min, \mathrm{mean}\}
\end{aligned}
$$

The AggPE groups the elements in a point cloud representation C based on (up to) $|\mathbf{I}|$ features $[G_1, G_2, \ldots, G_{|\mathbf{I}|}]$, computes $m - |\mathbf{I}|$ user-defined aggregated features per group $(A_1, A_2, \ldots, A_{(m-|\mathbf{I}|)})$, and counts the elements per group ($COUNT$). It then outputs the group indices and $(m - |\mathbf{I}|)$ features as defined by $F_1, F_2, \ldots F_{(m-|\mathbf{I}|)}$. If a user defines fewer than $|\mathbf{I}|$ group indices, the corresponding internal indices and outputs are set to 0.

Figure 5.6 depicts a simplified scheme of an AggPE. The result of the aggregation functions is evaluated continuously, such that the PE can receive a new element every few cycles. When an input element is valid, the PE first extracts its metadata, consisting of a frame number, quarter number, and a group index. If there is already an intermediate result for that metadata, the AggPE loads the previous result from memory, re-evaluates the aggregation function, and writes the new result back to the same address. Otherwise, the AggPE assigns an unoccupied address to the group.

On the output side, an arbiter controls the transmission of the results. It only outputs an element if it is sure that the results are definitive. We use the frames and quarter frames for that purpose. For example, the arbiter only outputs the results of frame i, quarter j after receiving an element from frame $i + 1$ or frame i, quarter $j + 1$.

As for the other processing elements, the size and thus the computational power of an AggPE is restricted by architectural parameters. The first parameter is $|\mathbf{I}|$, which restricts the maximum number of features to define a group index. In our implementation, $|\mathbf{I}| = 3$. The second parameter is the number of memory blocks the AggPE can use to store intermediate results, which in our implementation is 256. If all of these memory blocks are occupied and the AggPE cannot find a suitable location to store the next group in, the pipeline gets stuck.

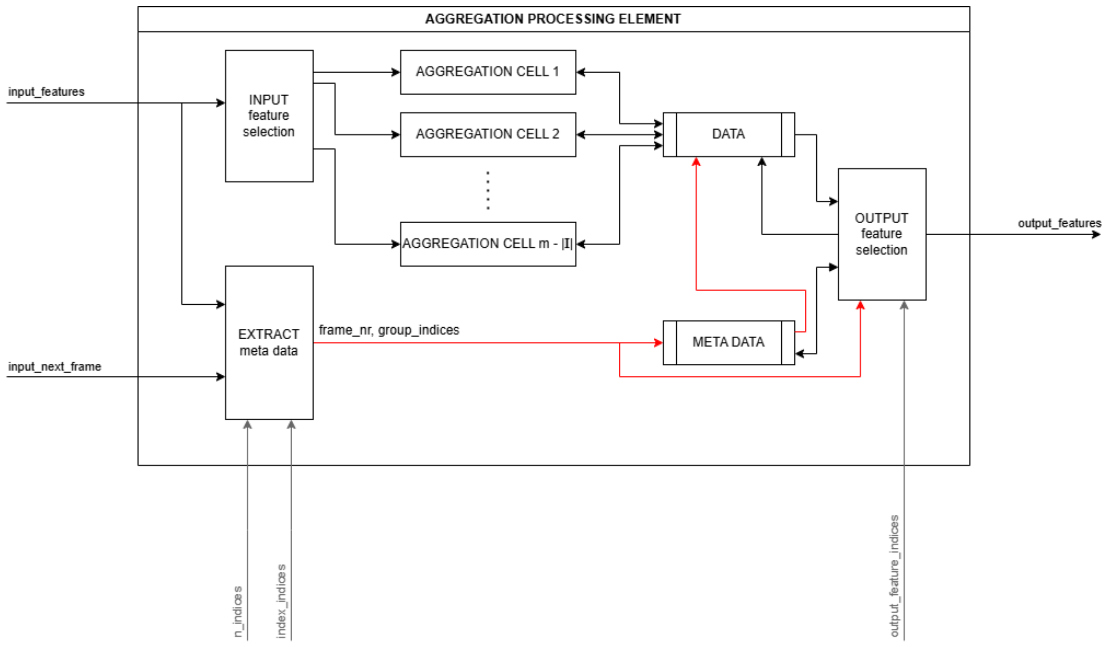
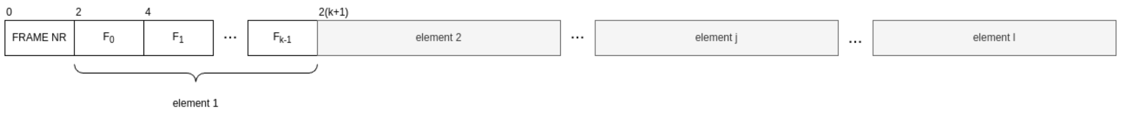
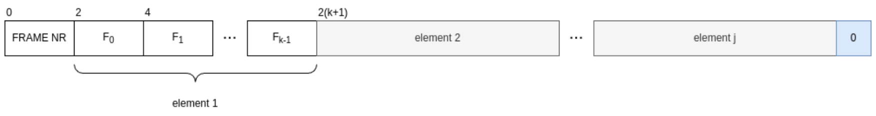


Figure 5.6: **High-Level Schematics of an Aggregation Processing Element (AggPE)**. The elements of each frame are grouped based on a set of $|\mathbf{I}|$ index features. Each aggregation cell extracts a statistical feature per group. The processing element then outputs the index and $(m - |\mathbf{I}|)$ features for each group.



((a)) Full PALICUS Output Packet



((b)) PALICUS Output Packet at the End of a Frame

Figure 5.7: **Payload Format of PALICUS' Output Data Stream**. The first two bytes always hold the number of the current frame. (a) shows a regular packet with l elements. (b) shows a packet that is terminated early due to the end of the current frame. It holds $j < l$ elements and a single zero byte.

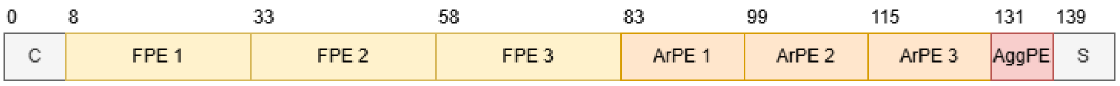
5.4 Serializer Module

The serializer module applies a final projection to the point cloud representation and serializes the selected features into an AXI stream for transmission. It partitions the data into packets, ensuring each packet contains data from only one frame. The serialized module thus completes the frame splitting operator initiated by the parsing module. The payload of each packet starts with two bytes containing a frame number. The remaining payload content is fully configurable by the following parameters:

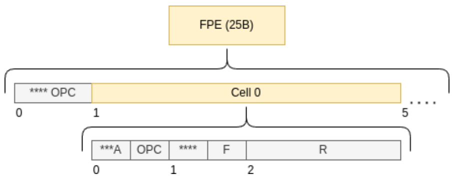
- `items_per_packet`: the maximum number of elements to be transmitted in one packet.
- `features_per_item`: the number of features to describe each element.
- `feature_indices`: up to seven feature indices F_j specifying the features to transmit. If `features_per_item` $= k < 7$, the module ignores the remaining indices.

For the remainder of this section, let `items_per_packet` $= l$ and `features_per_item` $= k$.

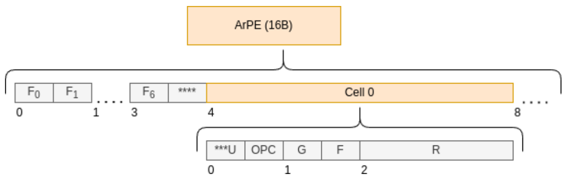
The payload of a packet transmitted by PALICUS always starts with two bytes holding a frame number. It is followed by at most $l \times k \times 2$ bytes describing l elements by the features passed in the configuration, as depicted in Figure 5.7(a). However, if the frame terminates before the packet full, the serializer appends a single 0-byte to the payload. Figure 5.7(b) illustrates this scenario. All values are represented by two bytes in big-endian byte order.



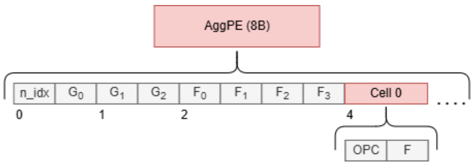
((a)) Payload Structure to Transmit a PALICUS Program



((b)) FPE Configuration



((c)) ArPE Configuration



((d)) AggPE Configuration

Figure 5.8: **PALICUS Program**. (a) shows the whole program, while (b) - (d) describe the configuration for each processing element type.

5.5 PALICUS Programming

PALICUS provides register files to store the entire configurations of all processing elements, the crossbar, and the serialization module, as well as a parser to fill these registers from the payload of a UDP packet, as depicted in Figure 5.8(a). For this reason, we refer to the register file as a Palicus micro-program, or simply program henceforth. Fields spanning more than one byte are encoded in big-endian byte order. Note that indices started at 1 in the formal definitions of the previous chapters and sections for the sake of simplicity. In the PALICUS program, however, indices start from 0.

Crossbar (8 Bytes): Each byte in the crossbar configuration configures one row of the crossbar, starting from the bottom in Figure 5.2. Thus, byte `0` configures the input to FPE 1, byte `1` configures it to FPE 2, and so on. A crossbar configuration is valid if it creates a path from the input to the output.

FPE (25 Bytes per PE): The FPE configuration starts with one byte where the two least significant bits `[1:0]` declare the unit opcode as specified at the top of table 5.3(a). It is followed by configuration for the six filter cells.
Filter Cells (4 Bytes per Cell): The first byte defines the operation of the cell. Bit `[4]` holds a flag indicating whether the cell is active. Bits `[3:0]` hold the cell's opcode as specified at the bottom of Table 5.3(a). The second byte indexes the feature to be compared. The third and fourth bytes hold the constant that limits the attribute domain.

Level	Operation	BIN
Unit	OR	00
	AND	01
	NOR	10
	NAND	11
Cell	$<$	S^*000
	$=$	S^*001
	$>$	S^*010
	$\geq$	S^*100
	$\neq$	S^*101
	$\leq$	S^*110

((a)) **FPE OpCodes**.

PE	Operation	BIN
ArPE	add	S^*00
	sub	S^*01
	div	S^*11
AggPE	mean	S^*01
	min	S^*10
	max	S^*11

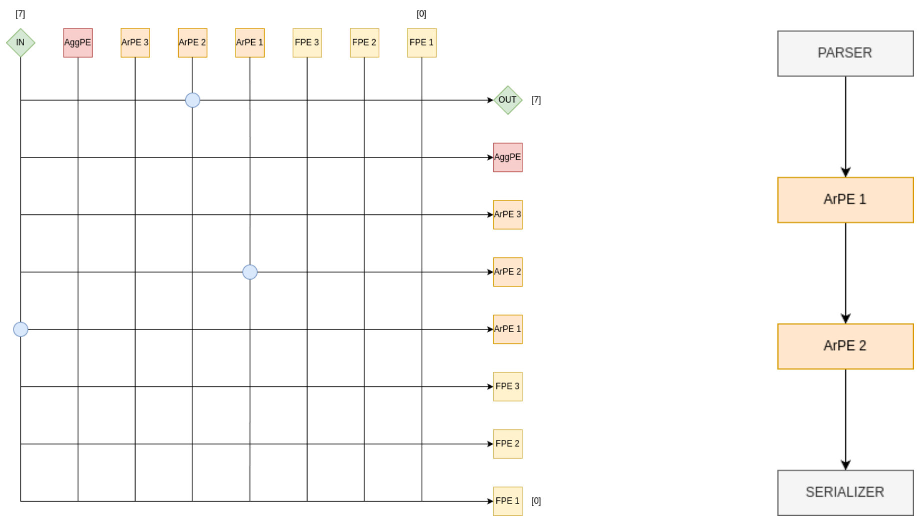
((b)) **Arithmetic and Aggregation Cell OpCodes**.

Table 5.3: **OpCodes for Various Processing Elements**. S^* denotes a flag set to 1 if the operation is signed.

ArPE (16 Bytes per PE): The first two bytes index the seven features that the element outputs. Indices in $\{0, 1, \ldots, 6\}$ refer to the features input to the ArPE, while indices in $\{7, 8, 9\}$ select outputs from the arithmetic cells. Each index spans four bits. We thus transmit two indices per byte. Bits `[3:0]` of the last byte are ignored. The remaining bytes configure the three arithmetic cells.
Arithmetic Cells (4 Bytes per Cell): The first byte specifies the operation to be performed by the cell. Bit `[4]` specifies whether to apply $\mathcal{B}_{const}$ (1) or $\mathcal{B}_{ftr}$ (0) to the incoming elements. Bits `[2:0]` specify the operator as specified in the top section of Table 5.3(b). The second byte holds the indices specifying the input features. Bits `[7:4]` hold the index of the second feature which is only forwarded to a cell performing $\mathcal{B}_{ftr}$ whereas the feature indexed by bits `[3:0]` is forwarded to the first input of the arithmetic cell in any case. The last two bytes hold the constant which is forwarded to the second input of the cell when performing $\mathcal{B}_{const}$.

AggPE (8 Bytes per PE): The first two bytes define the index features used to determine the groups. Bits `[7:4]` in the first byte hold the number of group indices. We can thus specify up to three indices, each specified by four bits and starting with bits `[3:0]` in the first byte. The following two bytes hold the indices specifying what features to output, formatted like the output features of an ArPE. Indices in $\{0, 1, \ldots, 3\}$ refer to the output of the aggregation cells, whereas any index > 4 refers to $\mathrm{count}(*)$. The configuration is completed by *1 Byte per aggregation cell*; bits `[6:4]` describing the operator (see Table 5.3(b), bottom) and `[3:0]` indexing the feature input to the cell.

Serialization (7 Bytes): The first two bytes of the serialization configuration denote the maximal number of elements to be output per packet. The third byte holds the number of features per item, and the remaining bytes list the features to be transmitted.



((a)) Crossbar ((b)) Pipeline

Figure 5.9: **Crossbar Configuration for a Depth Map**.

Example 1: Depth Map

To provide a concrete PALICUS program to build a depth map, we first need to discuss the setup. As mentioned in Section 5.2, our PALICUS implementation supports velodyne's VLP-16 [64]. It captures a vertical field of view bounded by $[-15°, 15°]$ and with a resolution of $2°$. We further assume that it covers the whole $360°$ horizontal field of view with a resolution of $0.2°$. Thus $\omega_{min} = -15$, $\omega_{RES} = 2$, and $\phi_{RES} = 20$.

Table 5.4 shows the full PALICUS program to build a depth map. As depicted in Figure 5.1, we only need two ArPEs to compute the pixel indices p_x, p_y. ArPE 1 replaces ω by $\omega' := \omega - \omega_{min}$. It thus implements the following DAMOCLES expression:

$$C_1 = \mathcal{P}_{[0,1,2,3,4,7,6]}\Big(C_{in} \| \mathcal{B}_{const}(C_{in}, [5], -15, \text{sub})\Big)$$

ArPE 2 computes the pixel coordinates p_x, p_y and applies

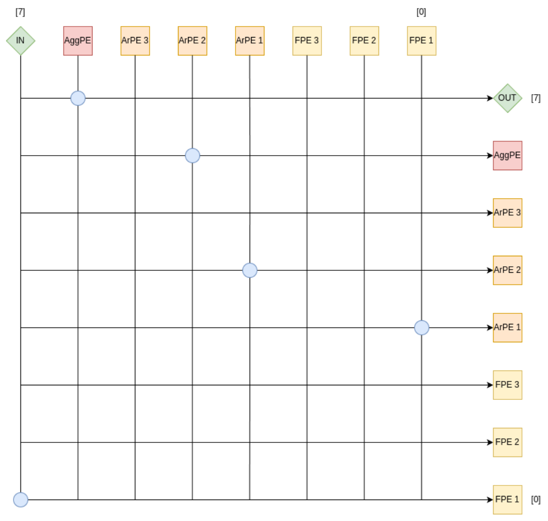
$$C_2 = \mathcal{P}_{[0,1,2,3,7,8,6]}\Big(C_1 \| \mathcal{B}_{const}(C_1, [4], 20, \text{div}) \| \mathcal{B}_{const}(C_1, [5], 2, \text{div})\Big)$$

to the point cloud. It replaces ϕ by $p_x := \frac{\phi}{\phi_{res}}$ and ω' by $p_y := \frac{\omega'}{\omega_{res}}$. Finally, we configure the serializer to select the features (p_x, p_y, r), thus

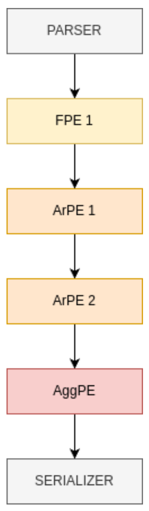
$$C_{out} = \mathcal{P}_{[4,5,3]}(C_2)$$

Byte Index	Attribute	Value (Hex)
`[0]`	Crossbar (see Figure 5.9)	`#00`
`[1]`		`#00`
`[2]`		`#00`
`[3]`		`#80`
`[4]`		`#08`
`[5]`		`#00`
`[6]`		`#00`
`[7]`		`#10`
`[8:82]`	FPE 1-3	`#0000...`
`[83]`	ArPE 1 - Output Feature Indices:	`#01`
`[84]`	replace feature [5] (ω) with the output of the first	`#23`
`[85]`	cell ($\omega' := \omega - (-15°)$)	`#47`
`[86]`		`#60`
`[87]`	ArPE 1 - Cell 1 OpCode: $\mathcal{B}_{const}$; signed sub	`#15`
`[88]`	ArPE 1 - Cell 1: Input Feature: [5] (ω)	`#05`
`[89:90]`	ArPE 1 - Cell 1: Register (-15)	`#fff1`
`[91:98]`	ArPE 1 - Cells 2 & 3	`#0000...`
`[99]`	ArPE 2 - Output Feature Indices:	`#01`
`[100]`	replace feature [4] (ϕ) with the output of the first	`#23`
`[101]`	cell ($p_x := \phi \div 20$) and feature [5] (ω') with the	`#78`
`[102]`	output of the second cell ($p_y := \omega' \div 2$)	`#60`
`[103]`	ArPE 2 - Cell 1 OpCode: $\mathcal{B}_{const}$; div	`#13`
`[104]`	ArPE 2 - Cell 1 Input Feature: [4] (ϕ)	`#04`
`[105:106]`	ArPE 2 - Cell 1 Register (20)	`#0014`
`[107]`	ArPE 2 - Cell 2 OpCode: $\mathcal{B}_{const}$; div	`#13`
`[108]`	ArPE 2 - Cell 2 Input Feature: [5] (ω')	`#05`
`[109:110]`	ArPE 2 - Cell 2 Register (2)	`#0002`
`[111:114]`	ArPE 2 - Cell 3	`#0000...`
`[115:130]`	ArPE 3	`#0000...`
`[131:138]`	AggPE	`#0000...`
`[139:140]`	Serializer - Items per Packet (128)	`#0080`
`[141]`	Serializer - Features per Item (3)	`#03`
`[142:146]`	Serializer - Feature Indices: (p_x, p_y, r)	`#45300000`

Table 5.4: **PALICUS Program to Build a Depth Map.** Bits in gray are either not part of the pipeline or ignored due to the configuration. They may hold arbitrary value, but here, we set them to zero.



((a)) Crossbar



((b)) Pipeline

Figure 5.10: **Crossbar Configuration for a BEV Image**: (a) shows the corresponding crossbar configuration and (b) the resulting pipeline.

Example 2: Height Map

In this example, we want to build the height map for a $6m \times 6m$ square centered at the LiDAR with a resolution of $20cm$. We will thus fill $r_{max} = 750$ and $\Delta = 50$ into the expressions in Section 4.5.

Table 5.5 describes the full PALICUS program to build a height map. As depicted in Figure 5.10 the pipeline consists of one FPE, followed by 2 ArPEs, and completed by an AggPE. The FPE removes all points that do not lie in our region of interest:

$$C_1 = \mathcal{F}_{[0]\geq -750 \wedge [1] \geq -750 \wedge [0] \leq 750 \wedge [1] \leq 750}(C)$$

The ArPEs compute the pixel coordinates p_x, p_y in two steps. ArPE 1 shifts the remaining points to the positive section of the x, y axes while ArPE 2 assigns each point to a $20cm \times 20cm$ cell:

$$C_2 = \mathcal{P}_{[7,8,2,3,4,5,6]}(C_1 \| \mathcal{B}_{const}(C_1, [0], 750, \text{add}) \| \mathcal{B}_{const}(C_1, [1], 750, \text{add})$$
$$C_3 = \mathcal{P}_{[7,8,2,3,4,5,6]}(C_2 \| \mathcal{B}_{const}(C_2, [0], 50, \text{div}) \| \mathcal{B}_{const}(C_2, [1], 50, \text{div})$$

The cell indices are now stored at positions [0, 1] of the outgoing feature vector.

Then we use the AggPE to compute the maximal height ($\max(z)$) for each cell:

$$C_4 = \mathcal{P}_{[0,1,2,0,0,0,0]}(\mathcal{A}(C_3, [0,1], \max, [2]))$$

Finally, we use the serializer to transmit the p_x-coordinate, p_y-coordinate, and height value per pixel:

$$C_{out} = \mathcal{P}_{[0,1,2]}(C_4)$$

Byte Index	Attribute	Value (Hex)
[0]	Crossbar (see Figure 5.10)	#80
[1]		#00
[2]		#00
[3]		#01
[4]		#08
[5]		#00
[6]		#10
[7]		#40
[8]	FPE 1 - Unit OpCode (AND)	#01
[9]	FPE 1 - Cell 1 OpCode: active, signed $\geq$	#1C
[10]	FPE 1 - Cell 1 Input Feature: [0] (x)	#00
[11:12]	FPE 1 - Cell 1 Register: -3m (-750)	#FD12
[13]	FPE 1 - Cell 2 OpCode: active, signed $\geq$	#1C
[14]	FPE 1 - Cell 2 Input Feature: [1] (y)	#01
[15:16]	FPE 1 - Cell 2 Register: -3m (-750)	#FD12
[17]	FPE 1 - Cell 3 OpCode: active, signed $\leq$	#1E
[18]	FPE 1 - Cell 3 Input Feature: [0] (x)	#00
[19:20]	FPE 1 - Cell 3 Register: 3m (750)	#02EE
[21]	FPE 1 - Cell 4 OpCode: active, signed $\leq$	#1E
[22]	FPE 1 - Cell 4 Input Feature: [0] (x)	#00
[23:24]	FPE 1 - Cell 4 Register: 3m (750)	#02EE
[25:32]	FPE 1 - Cells 5 - 6: inactive	#0000000000...
[33:82]	FPE 2 - 3	#0000...
[83]	ArPE 1 - Output Feature Indices:	#78
[84]	replace feature [0] (x) with the output of the first	#23
[85]	cell ($x' := x + 25m$) and feature [1] (y) with the	#45
[86]	output of the second cell ($y' = y + 25m$)	#60
[87]	ArPE 1 - Cell 1 OpCode: $\mathcal{B}_{const}$; signed add	#14
[88]	ArPE 1 - Cell 1: Input Feature: [0] (x)	#00
[89:90]	ArPE 1 - Cell 1: Register (750)	#02EE
[91]	ArPE 1 - Cell 2 OpCode: $\mathcal{B}_{const}$; signed add	#14
[92]	ArPE 1 - Cell 2: Input Feature: [1] (y)	#0y
[93:94]	ArPE 1 - Cell 2: Register (750)	#02EE
[95:98]	ArPE 1 - Cell 3	#0000...
[99]	ArPE 2 - Output Feature Indices:	#78
[100]	replace feature [0] (x') with the output of the first	#23
[101]	cell ($p_x := x' \div 10cm$) and feature [1] (y') with the	#45
[102]	output of the second cell ($p_y := y' \div 10cm$)	#60
[103]	ArPE 2 - Cell 1 OpCode: $\mathcal{B}_{const}$; div	#13
[104]	ArPE 2 - Cell 1 Input Feature: [0] (x')	#00
[105:106]	ArPE 2 - Cell 1 Register: 10cm (25)	#0019
[107]	ArPE 2 - Cell 2 OpCode: $\mathcal{B}_{const}$; div	#13
[108]	ArPE 2 - Cell 2 Input Feature: [1] (y')	#01
[109:110]	ArPE 2 - Cell 2 Register 10cm (25)	#0019
[111:114]	ArPE 2 - Cell 3	#0000...
[115:130]	ArPE 3	#0000...
[131]	AggPE Group Definition:	#20
[132]	2 indices located at position [0, 1]	#10
[133:134]	AggPE Output Feature Indices: first aggregation cell	#0000
[135]	AggPE Cell 1 Definition: signed_max(z)	#72
[136:138]	AggPE - Cells 2 - 4	#000000
[139:140]	Serializer - Items per Packet (128)	#0080
[141]	Serializer - Features per Item (3)	#03
[142:146]	Serializer - Feature Indices: ($p_x, p_y, \max(z)$)	#01300000

Table 5.5: **PALICUS Program to Build a Height Map.**

6 Experiments

This chapter evaluates our hardware prototype PALICUS regarding its capability to (a) process LiDAR data in real-time, and (b) serve as a building block in an edge system that performs more complex target tasks such as semantic segmentation or object detection. We are primarily interested in two aspects: latency and accuracy. The *latency* of a pipeline depends on the number and type of processing elements (PEs) included. In the spirit of an ablation study, we aim to determine how each PE type affects processing time. In Section 6.2, we thus study the latency of different pipelines applied to real-world LiDAR data.

However, low latency is not enough. To perform a complex task with high quality, the input representations must be accurate. PALICUS mainly operates on integers, and sometimes fixed-point numbers, and thus represents features with a lower precision than an all-purpose hardware like a CPU. In Section 6.3, we thus compare the representations computed by PALICUS to a ground truth.

As previously discussed, the input to PALICUS' LPU is a point stream. We thus assume that the processing time per point for a given pipeline is relatively constant. Different types of LiDAR devices, however, output points at different rates. High-resolution sensors, for example, output up to 5.2×10^6 points per second (*e.g.,* [30]). In Section 6.4, we thus aim to find out to what extent PALICUS can handle a higher throughput. This third set of experiments further allows us to explore how the LiDAR's frame rate affects the processing time. But before diving into the results, Section 6.1 describes the setup of our experiments.

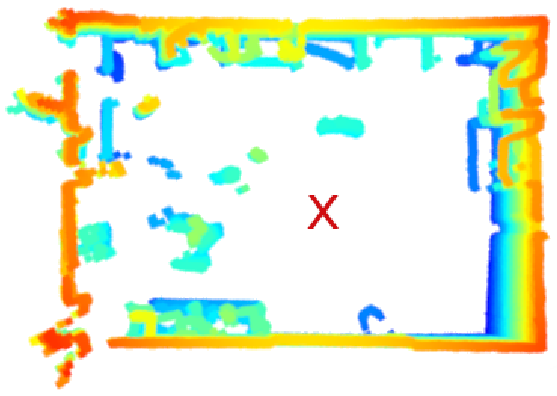


Figure 6.1: **Data Collection Setup**. We can see a single frame of the point cloud in our experiments data set. The red cross marks the approximate position of the LiDAR, approximately in the center of the office.

6.1 Setup

Data Set

We used a velodyne VLP-16 surround LiDAR sensor [64] to collect real-world data in an indoor setting. The sensor was installed approximately in the center of an office, as shown in Figure 6.1. The sensor has a fixed vertical field of view $\omega \in [-15°, 15°]$ captured with an angular resolution $\omega_{RES} = 2°$ and outputs about $288k$ points per second. We further configured the rotational speed of the LiDAR to 300 RPM such that the scene was scanned with $\phi_{RES} = 0.1°$. This setting produces five frames per second, where each frame holds approximately $57,600$ points. We captured 525 complete frames using Wireshark [23] and stored the data in a pcap file. The incomplete frames at the data stream's beginning and end are excluded from our analysis. The data is unprocessed except for filtering out packets that do not carry point cloud data, *i.e.,* the sensor's location packets.

The data set thus consists of 79,321 UDP packets, each holding 24 firing sequences, grouped into 12 data blocks.[1] Each data block starts with a flag (`#ffee`) followed by the azimuth ϕ. It then holds 2×16 measurements belonging to two distinct firing sequences. Each measurement consists of two bytes holding the radius r and one byte indicating the reflection intensity. Note that ϕ refers to the first firing sequence in the data block. We can approximate the azimuth of the second sequence by $\phi + \phi_{RES}$.

[1]2 sequences per block

Pipeline	FPE	ArPE 1	ArPE 2	AggPE
CONV	-	-	-	-
RoI	$r > 0m$	-	-	-
DM(1)	$r > 0m$	$p_x \leftarrow \frac{\phi}{\phi_{RES}}$ $p_y \leftarrow \frac{\omega}{\omega_{RES}}$	-	-
DM(2)	$r > 0m$	$\omega' \leftarrow \omega + 15°$	$p_x \leftarrow \frac{\phi}{\phi_{RES}}$ $p_y \leftarrow \frac{\omega'}{\omega_{RES}}$	-
BEV	$x \in [-3m, 3m] \wedge$ $y \in [-3m, 3m] \wedge$ $r > 0m$	$x' \leftarrow x + 3m$ $y' \leftarrow y + 3m$	$p_x \leftarrow \frac{x'}{20cm}$ $p_y \leftarrow \frac{y'}{20cm}$	`SELECT` $p_x, p_y, \max(z)$ `GROUP BY` p_x, p_y

Table 6.1: **Pipeline Configuration.** Each pipeline appends another processing element to the pipeline.

Pipelines

We evaluated the latency of five data processing pipelines, each building on top of each other:

- The **CONV** pipeline replaces the LiDAR coordinates with Cartesian coordinates. It does not contain any processing elements from the LPU as the parsing module handles coordinate conversion.
- The **RoI** pipeline adds a single FPE to the pipeline, filtering out invalid measurements ($r = 0$).
- The **DM (1)** and **DM (2)** pipelines each build a depth map of the scene. The DM (1) pipeline allows for negative pixel coordinates p_y and thus only requires one ArPE. The DM (2) pipeline comprises one FPE and two ArPEs.
- The **BEV** pipeline combines all three types of processing elements to build a height map of the scene.

Table 6.1 describes the point cloud manipulation performed at each pipeline stage. We configured the deparsing module such that each packet contains at most 128 points or pixels and describes each element by three features: (a) the Cartesian coordinates (x, y, z) for the conversion and RoI pipelines, (b) (p_x, p_y, r) for the depth maps, and (3) $(p_x, p_y, \max(z))$ for the height map (BEV).

System

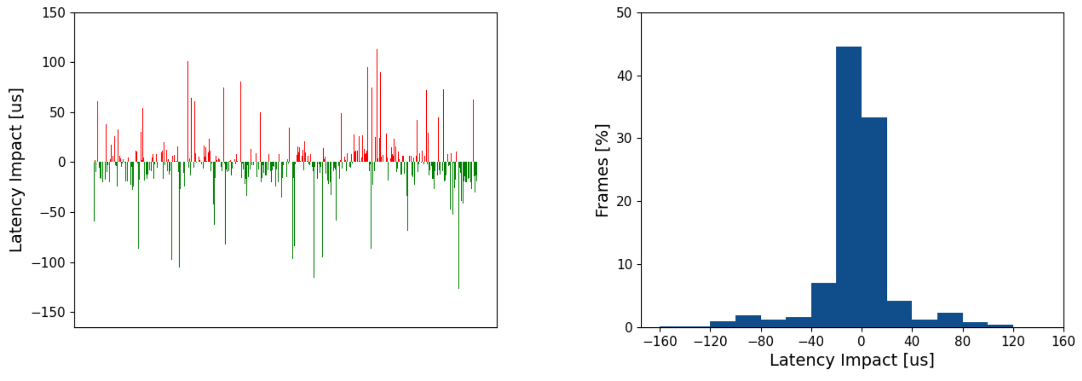
We implemented PALICUS on the FPGA of a KRIA kr260 SoM [3], an edge-sized MPSoC. Table 6.2 summarizes the resource utilization. As we can see, PALICUS is far from exceeding the board's resources. We connected our host system to the prototype through a 1 Gbps LAN. After programming the FPGA, we used ALPHA's [55] configuration software to configure the various pipelines. We then replayed the collected VLP-16 data using Bit-Twist [77] and captured the network traffic using Wireshark [23].

Resource	Utilization	
LUT	44,041	37.6 %
FF	39,288	16.7 %
BRAM	3.5	2.4 %
DSP	34	2.7 %

Table 6.2: **Summary of PALICUS' Resource Utilization**. The central column holds the absolute numbers per resource. The right column holds the relative utilization with respect to the FPGA's available resources.

	CONV	RoI	DM (1)	DM (2)	BEV
mean	390 μs	392 μs	394 μs	394 μs	46.143 ms
median	299 μs	296 μs	299 μs	299 μs	47.705 ms

Table 6.3: **Overall Latency of the Five Pipelines**.



((a)) Per Frame ((b)) Summary

Figure 6.2: **Impact of a Filtering Processing Element on the Processing Time**. (a) shows the impact for each frame. Green bars represent an improved processing time, whereas red bars represent an increased latency. (b) summarizes the results. We can see that the distribution is slightly left skewed, meaning that the processing time is reduced for a majority of frames.

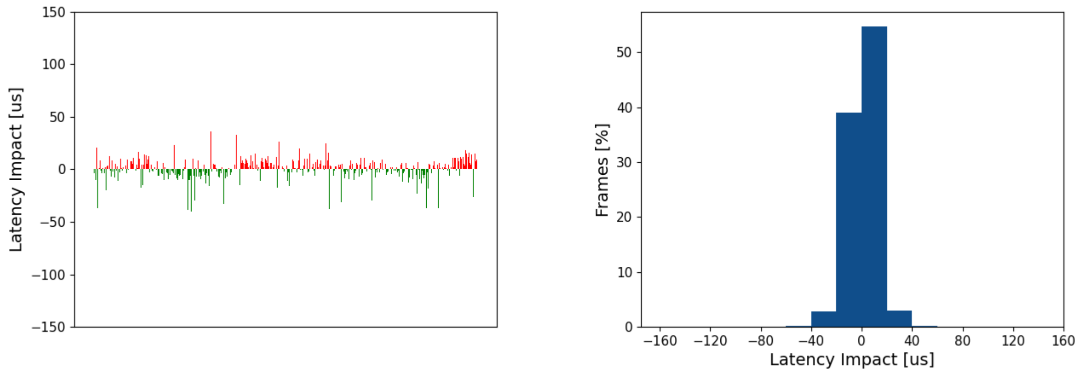
6.2 Latency

For the first set of experiments, we replayed the data set at 8 $Mbps$, which approximately equals the throughput of the LiDAR sensor's data stream. For each pipeline, we repeated this procedure three times, resetting the pipeline after each run.

In the resulting pcap file, containing both outgoing and incoming packets, we annotated each packet with its timestamp and one or more frame numbers.[2] We then computed the latency for each frame as the time difference between the last outgoing packet holding measurements associated with that frame and the last incoming packet labeled with the corresponding frame number. The measured latency thus comprises the processing time plus some latency imposed by the network, causing some variance. However, as we ran the experiments several times, we can identify some trends.

Baseline Latency: The CONV pipeline does not include any processing elements from the LPU. Each point is thus directly forwarded from the parsing to the serialization module. This allows the system to process most frames in less than $0.4\ ms$. However, some outliers took up to $1.8\ ms$ to process. These numbers serve as a baseline for our analysis. In the further course of this section, we analyze how the different processing elements impact the system's overall latency.

[2]Packets output by the LiDAR may hold points from multiple frames.



((a)) Per Frame ((b)) Summary

Figure 6.3: **Impact of an Arithmetic Processing Element on the Processing Time**. (a) shows the impact for each frame. Green bars represent an improved processing time, whereas red bars represent an increased latency. (b) summarizes the results. We can see that the distribution is slightly right skewed, meaning that the processing time is increased for a majority of frames.

Filtering Processing Element: In order to determine the impact of an FPE, we compared the CONV and the RoI pipelines. For both pipelines, we can find frames that take significantly longer to process than the others. As the set of outliers is not the same, we excluded all of them from our analysis. The results for the remaining frames are illustrated in Figure 6.2.

An FPE takes a few cycles to forward an element, thus increasing the time the element spends in the pipeline. In return, it can drop elements, reducing the amount of data processed later in the pipeline. As dropping takes fewer cycles and can be performed independently from the downstream module, this can reduce the frame-level processing time by reducing the backpressure in the pipeline. The impact of an FPE on the processing time thus depends on the number of dumped elements. In our scenario, the RoI pipeline removes approximately 4.4% of the data measurements. This ratio is relatively constant for all frames. As Figure 6.2(a) shows, this dropout reduces the processing time for most frames, on average by $4.5\ \mu s$ ($-1.1\ \%$). Thus, filtering out a small ratio of elements can already improve the processing time.

Arithmetic Processing Element: In order to evaluate the impact of an ArPE, we conducted two comparisons: RoI – DM (1) and DM (1) – DM (2). We first compared the pipeline pairwise and then computed the mean. The results are illustrated in Figure 6.3.

As all processing elements, an ArPE takes a few cycles to compute new features and forward the manipulated element. Unlike FPEs, however, it cannot drop elements. We thus would expect ArPEs to increase the latency of the pipeline. Looking at Figure 6.3, we can see the tendency that the majority of frames takes longer to be processed ($\text{median} = +0.83\ \mu\text{s}$). However, there are also many frames where the processing time decreased. If we look at the comparisons individually, we can see more drastic time differences, *i.e.,* the whole range between $-132\ \mu s$ and $+134\ \mu s$ is covered. Having two data series helps us to get a better idea of an ArPE's impact on pipeline latency. We assume that each ArPE increases the latency by approximately $1\ \mu s$.

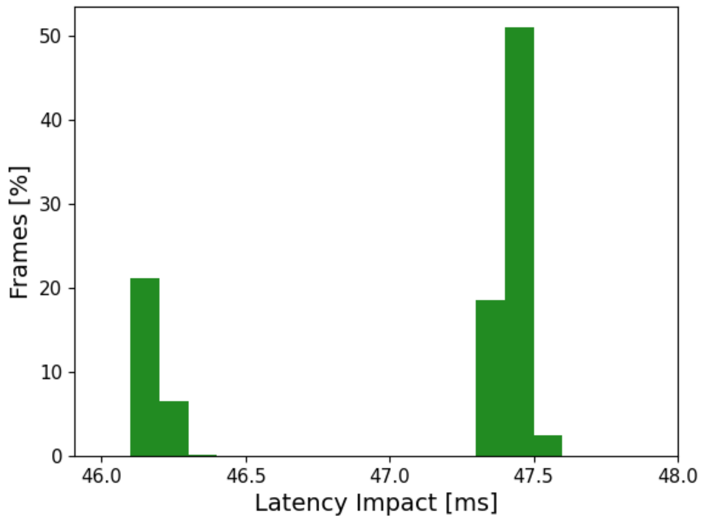
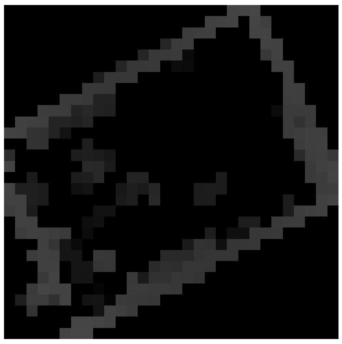
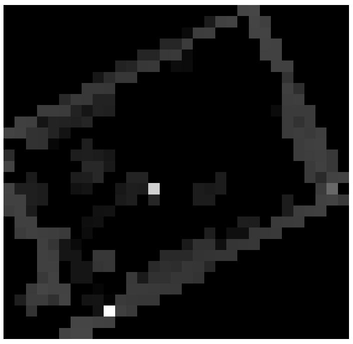


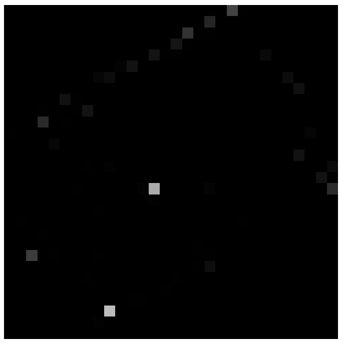
Figure 6.4: **Impact of an Aggregation Processing Element on the Processing Time**. The latency is increased by approximately $47.5\ ms$ for most frames.

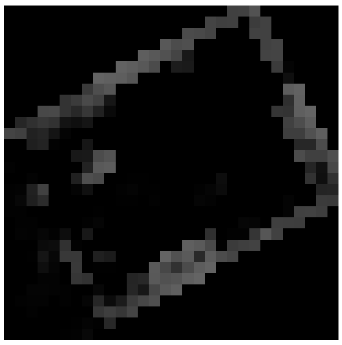
Aggregation Processing Element: In order to evaluate the impact of an AggPE on the processing time, we compared the latency between BEV and the DM (2) pipeline. While a few frames take longer in the DM (2) pipeline, there is also a small subset of frames that take much less time in the BEV pipeline. We excluded all outliers from our analysis.

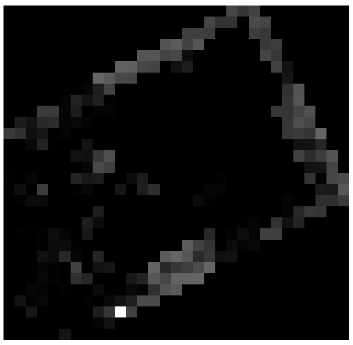
While the impact of the previous processing elements is hardly noticeable, the contribution of an AggPE is more severe, on average $47.08\ ms$ per frame. Compared to that, the impact of FPEs and ArPEs seems negligible. Like other processing elements, AggPEs can receive and output elements every few cycles. While the other PEs operate on a point stream, AggPEs have to summarize data over a certain interval, which is one quarter of a frame in our implementation. In the meantime, the intermediate results are buffered, even if they are not updated any further. On average, the LiDAR outputs a new frame every $200\ ms$, thus starting a new quarter every $50\ ms$. The increased latency for the BEV pipeline is thus mostly related to that buffering time. We will verify this hypothesis in the last section (6.4) of this chapter.


((a)) Height - Ground Truth


((b)) Height - PALICUS


((c)) Height - Delta


((d)) Intensity - Ground Truth


((e)) Intensity - PALICUS

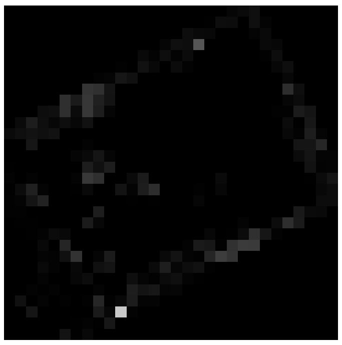

((f)) Intensity - Delta

Figure 6.5: **Comparing BEV Representations**. The top row illustrates the height map $\max(z) \in [-0.7\ m, 5.7\ m]$. The bottom row illustrates the mean reflection intensity $\text{mean}(i) \in [0, 255]$. The left column shows the ground truth, the center the representations computed by PALICUS, and the right column the difference. The lighter the shade, the larger the value.

6.3 Accuracy

For the second set of experiments, we analyzed the LiDAR and the PALICUS traffic separately. We examined the RoI pipeline to study the accuracy of coordinate conversion, the DM (2) pipeline to evaluate the correctness of the computed pixel indices, and the BEV pipeline to verify data aggregation. For the latter, we computed a second channel holding the mean reflection intensity $(\text{mean}(i))$ per pillar in a separate run. This feature is interesting regarding precision, as all the intermediate results are approximations. Thus, the deviation from the ground truth may increase with each received element.

For the ground truth, we extracted the data from the input pcap file and used pandas [54] to build the target representations of each pipeline. We compared it to the representation extracted from the corresponding PALICUS traffic. Both the ground truth and the PALICUS representations were stored frame-wise in CSV files. For the first two pipelines, we associated the elements using their order. We thus only considered frames where the ground truth and the computed representation had the same size. For the point cloud built by the CONV pipeline, we computed the Euclidean distance for each point pair. For the DM (2) pipeline, we computed the L_1 distance of the pixel coordinates. For BEV images, we joined the ground truth and the computed representation based on the pixel coordinates and computed the distance pixel- and channel-wise.

Coordinate Conversion: PALICUS computes the Cartesian coordinates quite accurately. The maximum distance between the evaluated point pairs is less than $1\ cm$. This slight deviation relates to the fixed-point representation of real numbers in PALICUS, *e.g.,* $\cos\omega$ or $\sin\phi$, but also the lower precision when representing Cartesian coordinates.

Computing Pixel Indices: The DM (2) pipeline computes up to $6.6\ \%$ of the pixel indices in a frame incorrectly. Looking at the affected points, we can see that PALICUS computes the p_y coordinate correctly in all cases. This is no surprise as its computation only involves integer arithmetic. The result of $p_x = \frac{\phi}{\phi_{res}}$, however, may be a real number. We can observe that some points are assigned to the right neighbor of the ground truth pixel, meaning that the computed p'_x is equal to the actual pixel index p_x plus one. We assume that, instead of rounding it down, the DSP rounds up the division result. in those cases, *i.e.,* applies a "half-up" rounding strategy.

Aggregation: Figure 6.5 illustrates the ground truth and the computed channels of a BEV image, as well as the difference. The top row depicts the height maps. As we can see in 6.5(c), the difference between PALICUS and the ground truth is rather small. Most pixels are black, meaning the deviation relates to the imprecise representation of the z coordinate in PALICUS. There are approximately 25 gray pixels where the computed height deviates more significantly from the ground truth (ca. $10\ cm - 1.6\ m$). We identified two error sources: (1) the previously discussed inaccuracy when computing the pixel coordinates of a given point, and (2) a *pillar overflow*. The first source may affect the computed representation in three ways. It may change the result of two non-empty pillars, *e.g.,* when the highest point of (p_x, p_y) gets assigned to (p_{x+1}, p_y) and thereby overwrites the actual result of that pillar. However, if the assigned pillar (p_{x+1}, p_y) is empty in the ground truth, a new non-black pixel appears on the height map. Similarly, a pillar may disappear if the only point in (p_x, p_y) is assigned to the wrong pillar. Furthermore, inaccurate pixel index computation may lead to a pixel being listed twice in the PALICUS BEV image. For example, a point located in quarter q_i but near the border to quarter q_{i+1} may be assigned to a pillar located in the latter quarter. As aggregation is performed per quarter frame, that pillar is output twice, once for each quarter.

The second error source – a pillar overflow – refers to the situation when more than 255 points are assigned to the same pillar. In that case, the counter overflows, *i.e.,* restarts from zero, and the intermediate result is reset with the next point. However, there are several strategies to mitigate this problem. For example, we could implement a *hard pillarzation* strategy that ignores all points once a pillar is full.

The second row in Figure 6.5 visualizes the intensity channel of the BEV image. We can see that the difference, visualized in 6.5(f), between PALICUS and the ground truth is much larger as there are more non-black pixels. Of course, computing the mean suffers from the same issues as evaluating the maximal height per pillar, *i.e.,* inaccurate pixel indices and overflowing pillars. Additionally, PALICUS represents the reflection intensity as an integer $i \in [0, 255]$ and thus with a relatively low precision. We could increase the precision by using a different strategy to compute the mean, *e.g.,* summing up the values and counting the elements when receiving the quarter, and performing division when transmitting the final result. However, this would increase (a) the memory consumption per aggregation cell by $50\ \%$, and (b) routing congestion between the AggPE and the board's BRAM block as we need 24 bits (instead of 16) to store the intermediate results. Most likely, this would necessitate reducing the number of memory cells, hence limiting the design's scalability.

For both channels, we can see a small number of almost white pixels in the delta image (two in the height map and one in the intensity map). In both cases, these refer to pixels with unrealistic values, *e.g.,* a maximum height of almost $6\ m$ or an intensity value much larger than 255. There are several frames with such unrealistic values. These errors are hard to fix, as there is no explanation for this behavior. However, such outliers are easily detected and corrected when post-processing the BEV image, *e.g.,* by capping the pixel values.

Replay Speed	**CONV**	**RoI**	**DM (1)**	**DM (2)**	**BEV**
Ground Truth	30,381,904		29,031,584		133,378
8 Mbps (5 fps)	$\pm$ 0		-11		$+5,819$
16 Mbps (10 fps)	✓	✓	✓	✓	✓
32 Mbps (20 fps)	✓	✓	✓	✓	✓
64 Mbps (40 fps)	✓	✓	✓	✓	✓
128 Mbps (80 fps)	✓	-886	-384	✓	-1

Table 6.4: **Results of the Throughput Experiments**. The ground truth row denotes the number of points/pixels extracted from the LiDAR data stream. The 8 $Mbps$ row holds the number of elements extracted from the PALICUS traffic at that replay speed and represents the expected values. The remaining rows list the results for the accelerated runs. A check mark indicates that all expected elements have been transmitted. Otherwise, we report the difference.

6.4 Throughput

For the last set of experiments, we replayed the data several times with different speed. Starting from 8 $Mbps$, we doubled the speed at each iteration up to 128 $Mbps$ (ca. 4.6×10^6 points per second). The latter simulates the throughput of a high-resolution LiDAR sensor with 128 channels, such as Ouster's OS0 [30]. As for the previous experiments, we reset the pipeline after each run.

For the analysis, we used the ground truths from the previous experiments to compute the number of elements per frame. Similarly, we extracted the elements from the PALICUS traffic and computed the frame sizes. Table 6.4 summarizes these results by providing the total number of elements for each run.

The first row reports the total number of elements per pipeline as extracted from the LiDAR traffic. The second row holds the difference between the ground truth and the computed representations at 8 $Mbps$. This is the number of elements we expect to be transmitted by PALICUS. As you can see, the outputs of the RoI, DM (1), and DM (2) pipelines hold eleven elements less than the ground truth. A frame-level analysis shows that only nine frames are affected, each holding one or two fewer elements. The reason for that behavior is the total number of elements in these frames. The payload of the last packet would only comprise 8 or 14 bytes. The TX stack, however, only transmits packets of at least 64 B and thus drops these packets. However, we could easily pad the packets to the minimum length.

We can further see that PALICUS transmits almost $6,000$ more pixels than the ground truth suggests. Unlike in the previous case, the difference spreads evenly across all frames. As discussed in the previous section (6.3), pillar indices are sometimes computed inaccurately, leading to PALICUS' output representations comprising the same pixel multiple times or holding pixels that are not present in the ground truth.

The remaining rows hold the results for the accelerated replays. A check mark (✓) indicates that the number of elements equals the expected result. Otherwise, we report the difference. We can see that the number of elements match for $16,\ 32$, and 64 $Mbps$. However, the output size of the RoI, the DM (1), and the BEV pipelines deviates at 128 $Mbps$.

As previously discussed, data packets are either dropped or processed as a whole. Thus, we would expect the missing data to be associated with a FIFO overflow at the RX or TX path. Indeed, we can find indices for several packets being dropped on the RX path. For instance, the missing pixels in the DM (1) pipeline belong to the same frame (#109) and form a sequence. 384 pixels are missing, which is equal to the number of data measurements per LiDAR packet. We thus assume that PALICUS dropped one packet on the RX path.

The same is true for the points missing in frames $\#29$ and $\#357$ produced by the RoI pipeline; both missing a sequence of 384 points compared to the result produced at $8\ Mbps$. Frame $\#42$, however, misses a sequence of 118 points in its last quarter. As it is not the last packet of the frame, we cannot associate the loss with the TX path; otherwise, 128 points would be missing. However, as all incoming packets hold more than 118 valid measurements, we cannot associate it with the RX path either.

Finally, for the BEV pipeline, a single pixel is missing in frame $\#421$. However, as that pipeline aggregates the points pillar-wise, a pixel gets transmitted if a single point is inside its area. We thus cannot determine whether all transmitted points have been processed solely based on the number of pixels in the BEV images. Indeed, examining frame $\#421$, we can see that, in addition to the missing pixel, five pixels hold a different height value in the image generated at $128\ Mbps$. We scanned all output files and found a few deviating pixels at each replay speed, usually at most three pixels per frame.

Correlation between Frame Rate and Aggregation Latency: In section 6.2, we found that applying aggregation to the point cloud adds a delay of almost $50\ ms$ to the processing time of a frame. We further assumed that this delay relates to the frame rate, which is $5\ Hz$ when the data is replayed at $8\ Mbps$. As expected, the processing time per frame is decreased by almost $50\ \%$ when the replay is doubled. Interestingly, the processing time of the outliers, which are processed significantly faster than the other frames, is more stable between $250\ -\ 300\ \mu s$. This is probably near the lower bound of PALICUS' latency.

7 Discussion

At the beginning of this thesis stands a dissatisfaction with how past research approached the topic of LiDAR point cloud processing. We noticed a huge dominance of machine learning papers that primarily care about accuracy. Especially in traffic-related scenarios, however, accurate results are not enough if we do not get them within a reasonable time. We further realized that systems claiming to process the data in real time often neglect part of the system's latency, use unrealistic hardware, or operate on pre-processed data.

In this thesis, we pursued a classical data management approach to design a domain-specific accelerator (DSA) to pre-process LiDAR-generated point clouds. Our survey of traffic-related applications showed that state-of-the-art systems typically leverage machine learning models to solve a given target task with satisfying accuracy. We further noticed that none of these models operate on the raw LiDAR data. Instead, the systems pre-process the data and build a more suitable representation. We identified four dominant representations – Point Clouds, Voxels, FV, and BEV images. The associated pre-processing techniques reduce the representation's dimensionality or the amount of data to be processed (downsampling).

In the second step, we used these findings to derive a data model – DAMOCLES – that formalizes these processing techniques. In this model, we represent a point cloud by a real matrix where each row describes an element of the representation (*e.g.,* a point, a voxel, or a pixel) and each column represents a semantic feature. We further defined a set of operators that manipulate this representation.

Finally, we mapped parts of our data model to hardware. Our prototype – PALICUS – is a DSA for LiDAR point cloud pre-processing that builds on a programmable LiDAR Processing Unit (LPU). An LPU comprises three types of processing elements, each able to apply a family of DAMOCLES expressions, *i.e.,* filtering, arithmetic operations, and aggregation. The processing elements can be programmed independently. Furthermore, a crossbar scheduling approach allows us to assemble them arbitrarily into a pipeline. The LPU operates on a point stream, meaning it can receive one point at a time. It outputs an element stream, which, depending on the specified pipeline, carries points, voxels, or pixels.

Besides the LPU, PALICUS comprises networking and (de-)parsing modules. On the RX path, these modules parse the incoming UDP traffic into a point stream, annotate it, and split it into frames and quarter frames. On the TX path, the output element stream is de-serialized and wrapped back into UDP packets for transmission. We further provide a configuration parser that extracts a PALICUS program from the payload of a UDP packet.

The prototype's evaluation showed that it could build a wide range of point cloud representations in real time. Filtering elements and computing new features using arithmetic operations hardly add any delay to a pipeline's end-to-end latency. Thus, PALICUS can build point clouds and FV pseudo-images very efficiently ($< 2\ ms/frame$) and also with reasonable accuracy. Aggregation takes more time as information has to be summarized, and the incoming elements can thus not be forwarded immediately. The current implementation processes the data in quarter frames and thus imposes a delay equal to the amount of time the LiDAR takes to capture a quarter frame. For example, if the LiDAR that generates 5 frames per second, building the BEV image of a frame takes approximately $50\ ms$. We also would like to mention that the accuracy of aggregated features is lower. The reasons are, among others, incorrect set membership indices due to the DSP's rounding strategy and rounded intermediate results.

We further evaluated how PALICUS reacts when being exposed to a higher throughput on the RX path. We showed that most pipelines process the point cloud correctly when processing approximately 2×10^6 points per second (pps) ($64\ Mbps$), but start throwing some data at $128\ Mbps$ (4.2×10^6 pps), meaning that the output representation is incomplete or incorrect.

While we clearly showed that PALICUS can build point cloud representations in real time, it is not necessarily clear whether its acceleration is sufficient to perform complex tasks such as object detection or semantic segmentation on the edge. Future research may thus develop more lightweight machine learning models that can run real-time inference on an edge device. Nevertheless, we think that our results are auspicious for systems relying on a frontal view (FV) representation. For example, if PALICUS accelerates the encoding phase in RangeSeg [13], it can decrease the average computation time by almost $4.5\ ms$. As the system is close to meeting real-time requirements, these few milliseconds are precious.

For pipelines applying aggregation to the point cloud, this question is a bit more difficult to answer. Although there are models (*e.g.,* BEVDetNet [52]) that can detect objects in a BEV image with low latency, even on an embedded platforms, PALICUS would most likely fail to produce the input representation due to the lack of resources to store intermediate results in the AggPE. The current implementation can only build BEV images with at most 256 non-empty pixels per quarter frame. While this was sufficient in our experimental setup, it will most probably fail in an outdoor scenario or only allow the generation of the BEV image with a very low resolution. There are several algorithms to improve the memory footprint; however, these reduce the application domain of the AggPE, *i.e.,* by restricting it to cubic voxelization and pillarization.

Another question we did not answer in our experiments, is the relation between frame size and latency. High-resolution sensors hold up to 128 channels, resulting in approximately 0.5 M points per frame (at $5\ Hz$). Our second set of experiments already suggests that PALICUS may be unable to cope with such a high point rate. As PALICUS operates on a point stream, we do not expect the latency to increase significantly; and even if this was the case, we could diminish this effect ed by focusing on some region of interest, *i.e.,* droping most of the points at the beginning of the pipeline.

This thesis further opens several conceptual questions. First of all, the scope of the thesis is intentionally narrowed such that we were able to explore the chosen domain in depth. Future research may extend the domain of DAMOCLES and PALICUS to more use cases, *i.e.,* by defining additional operators and mapping them to hardware. A closely related topic is the optimization of PALICUS' architecture, *e.g.,* regarding the number of processing elements per type and their size. The current architectural parameters were chosen heuristically, and, although PALICUS is far from exceeding the FPGA's resources, we may need to free some of them, *i.e.,* by removing some processing elements or reducing their size.

Another interesting topic is making the hardware accessible from higher levels of the system stack. In our previous work [55], we provided a JSON schema and a parsing software that allows us to express a pipeline in a human-friendly manner and transform it into a PALICUS program. Using this software, however, still requires a deep understanding of the data model and the hardware architecture. Future work may define a programming model or provide a compiler to make the hardware accessible from higher levels of the system stack. Furthermore, such a compiler could include optimization for DAMOCLES expressions, for example, building the pipeline with as few building blocks as possible to optimize latency, and ordering them such that the size of the intermediate representations is minimized.

In summary, this thesis presented a domain-specific accelerator to pre-process LiDAR-generated point clouds for 3D object detection. It supports the four most common representations in real-time and with satisfying accuracy. We thus showed that hardware accelerators do not need to be application-specific to meet low latency requirements.

On a more abstract level, this thesis demonstrated how to combine knowledge and tools from different layers of the system stack to develop a time- and energy-efficient system. We believe this is only possible if we thoroughly understand the data we process. Thus, we hope this thesis will contribute to a more systematic and holistic exploration of the data processing domain in general.

Bibliography

[1] Waleed Ali, Sherif Abdelkarim, Mahmoud Zidan, Mohamed Zahran, and Ahmad El Sallab. Yolo3d: End-to-end real-time 3d oriented object bounding box detection from lidar point cloud. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 0–0, 2018.

[2] Yara Ali Alnaggar, Mohamed Afifi, Karim Amer, and Mohamed ElHelw. Multi projection fusion for real-time semantic segmentation of 3d lidar point clouds. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pages 1800–1809, 2021.

[3] AMD. Kria k26 som – optimized for vision ai and robotics applications. `https://www.amd.com/en/products/system-on-modules/kria/k26.html`. [Accessed April 08, 2025].

[4] Iro Armeni, Ozan Sener, Amir R Zamir, Helen Jiang, Ioannis Brilakis, Martin Fischer, and Silvio Savarese. 3d semantic parsing of large-scale indoor spaces. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1534–1543, 2016.

[5] Alireza Asvadi, Luis Garrote, Cristiano Premebida, Paulo Peixoto, and Urbano J Nunes. Multimodal vehicle detection: fusing 3d-lidar and color camera data. *Pattern Recognition Letters*, 115:20–29, 2018.

[6] Alejandro Barrera, Carlos Guindel, Jorge Beltrán, and Fernando García. Birdnet+: End-to-end 3d object detection in lidar bird's eye view. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2020.

[7] Jorge Beltrán, Carlos Guindel, Francisco Miguel Moreno, Daniel Cruzado, Fernando Garcia, and Arturo De La Escalera. Birdnet: a 3d object detection framework from lidar information. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 3517–3523. IEEE, 2018.

[8] Nicholas Brown, Johan Rojas, Nicholas Goberville, Hamzeh Alzubi, Qusay AlRousan, Chieh (Ross) Wang, Shean Huff, Jackeline Rios-Torres, Ali Ekti, Tim Laclair, Richard Meyer, and Zachary Asher. Development of an energy efficient and cost effective autonomous vehicle research platform. *Sensors (Basel, Switzerland)*, 22, 08 2022.

[9] Cornelius Buerkle, Fabian Oboril, Omar Zayed, and Kay-Ulrich Scholl. Histogrid: Robust lidar-based traffic monitoring. In *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*, pages 209–214. IEEE, 2023.

[10] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[11] Chao Cao, Marius Preda, and Titus Zaharia. 3d point cloud compression: A survey. In *Proceedings of the 24th International Conference on 3D Web Technology*, pages 1–9, 2019.

[12] Zonghan Cao, Ting Wang, Ping Sun, Fengkui Cao, Shiliang Shao, and Shaocong Wang. Scorepillar: A real-time small object detection method based on pillar scoring of lidar measurement. *IEEE Transactions on Instrumentation and Measurement*, 2024.

[13] Tzu-Hsuan Chen and Tian Sheuan Chang. Rangeseg: Range-aware real time segmentation of 3d lidar point clouds. *IEEE Transactions on Intelligent Vehicles*, 7(1):93–101, 2022.

[14] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1907–1915, 2017.

[15] Baya Cherif, Hakim Ghazzai, Ahmad Alsharoa, Hichem Besbes, and Yehia Massoud. Aerial lidar-based 3d object detection and tracking for traffic monitoring. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2023.

[16] Pong P. Chu. *FPGA prototyping by Verilog examples : Xilinx Spartan -3 version*. J. Wiley Sons, Hoboken, N.J, 1st ed. edition, 2008.

[17] William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. `https://cacm.acm.org/research/domain-specific-hardware-accelerators/`, Jul 2020. [Accessed March 10, 2025].

[18] Jean-Pierre Deschamps, Gery JA Bioul, and Gustavo D Sutter. *Synthesis of arithmetic circuits: FPGA, ASIC and embedded systems*. John Wiley & Sons, 2006.

[19] Martin Engelcke, Dushyant Rao, Dominic Zeng Wang, Chi Hay Tong, and Ingmar Posner. Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1355–1361. IEEE, 2017.

[20] Lue Fan, Xuan Xiong, Feng Wang, Naiyan Wang, and Zhaoxiang Zhang. Rangedet: In defense of range view for lidar-based 3d object detection. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 2918–2927, 2021.

[21] Hamidreza Fazlali, Yixuan Xu, Yuan Ren, and Bingbing Liu. A versatile multi-view framework for lidar-based 3d object detection with guidance from panoptic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17192–17201, 2022.

[22] John Alexander Forencich. *System-Level Considerations for Optical Switching in Data Center Networks*. PhD thesis, UC San Diego, 2020.

[23] Wireshark Foundation. Wireshark. the world's most popular network protocol analyzer. `https://www.wireshark.org/`. [Accessed April 17, 2025].

[24] Hongzhi Gao, Zheng Chen, Zehui Chen, Lin Chen, Jiaming Liu, Shanghang Zhang, and Feng Zhao. Leveraging imagery data with spatial point prior for weakly semi-supervised 3d object detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 1797–1805, 2024.

[25] Yacong Gao, Chenjing Zhou, Jian Rong, and Yi Wang. High-accurate vehicle trajectory extraction and denoising from roadside lidar sensors. *Infrared Physics & Technology*, 134:104896, 2023.

[26] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[27] Yulan Guo, Hanyun Wang, Qingyong Hu, Hao Liu, Li Liu, and Mohammed Bennamoun. Deep learning for 3d point clouds: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 43(12):4338–4364, 2020.

[28] Jordan SK Hu, Tianshu Kuai, and Steven L Waslander. Point density-aware voxels for lidar 3d object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8469–8478, 2022.

[29] Yuanxian Huang, Jian Zhou, Xicheng Li, Zhen Dong, Jinsheng Xiao, Shurui Wang, and Hongjuan Zhang. Menet: Map-enhanced 3d object detection in bird's-eye view for lidar point clouds. *International Journal of Applied Earth Observation and Geoinformation*, 120:103337, 2023.

[30] Ouster Inc. Ouster os0: High-precision ultra-wide short-range lidar sensor. `https://ouster.com/products/hardware/os0-lidar-sensor`. [Accessed May 09, 2025].

[31] Kiyosumi Kidono, Takeo Miyasaka, Akihiro Watanabe, Takashi Naito, and Jun Miura. Pedestrian recognition using high-definition lidar. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 405–410. IEEE, 2011.

[32] Yecheol Kim, Jaekyum Kim, Junho Koh, and Jun Won Choi. Enhanced object detection in bird's eye view using 3d global context inferred from lidar point data. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2516–2521. IEEE, 2019.

[33] Lingdong Kong, Youquan Liu, Runnan Chen, Yuexin Ma, Xinge Zhu, Yikang Li, Yuenan Hou, Yu Qiao, and Ziwei Liu. Rethinking range view representation for lidar segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 228–240, 2023.

[34] Jason Ku, Melissa Mozifian, Jungwook Lee, Ali Harakeh, and Steven L Waslander. Joint 3d proposal generation and object detection from view aggregation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–8. IEEE, 2018.

[35] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12697–12705, 2019.

[36] Cecilia Latotzke, Amarin Kloeker, Simon Schoening, Fabian Kemper, Mazen Slimi, Lutz Eckstein, and Tobias Gemmeke. Fpga-based acceleration of lidar point cloud processing and detection on the edge. In *2023 IEEE Intelligent Vehicles Symposium (IV)*, pages 1–8. IEEE, 2023.

[37] Enxu Li, Ryan Razani, Yixuan Xu, and Bingbing Liu. Cpseg: Cluster-free panoptic segmentation of 3d lidar point clouds. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8239–8245. IEEE, 2023.

[38] Yanwei Li, Xiaojuan Qi, Yukang Chen, Liwei Wang, Zeming Li, Jian Sun, and Jiaya Jia. Voxel field fusion for 3d object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1120–1129, 2022.

[39] Ming Liang, Bin Yang, Yun Chen, Rui Hu, and Raquel Urtasun. Multi-task multi-sensor fusion for 3d object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7345–7353, 2019.

[40] Se-Min Lim, Esmerald Aliaj, and Sang-Woo Jun. Durin: Cpu-fpga heterogeneous platform for scalable low-dimensional data clustering. In *2024 IEEE International Conference on Big Data (BigData)*, pages 344–351. IEEE, 2024.

[41] Zhenyu Lin, Masafumi Hashimoto, Kenta Takigawa, and Kazuhiko Takahashi. Vehicle and pedestrian recognition using multilayer lidar based on support vector machine. In *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pages 1–6. IEEE, 2018.

[42] Meng Liu and Jianwei Niu. Bev-net: A bird's eye view object detection network for lidar point cloud. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5973–5980. IEEE, 2021.

[43] Songbin Liu, Min Zhang, Pranav Kadam, and C.-C. Jay Kuo. *3D point cloud analysis : traditional, deep learning, and explainable machine learning methods*. Springer, 2021.

[44] Haihua Lu, Xuesong Chen, Guiying Zhang, Qiuhao Zhou, Yanbo Ma, and Yong Zhao. Scanet: Spatial-channel attention network for 3d object detection. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1992–1996. IEEE, 2019.

[45] Wenjie Luo, Bin Yang, and Raquel Urtasun. Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 3569–3577, 2018.

[46] Yecheng Lyu, Lin Bai, and Xinming Huang. Chipnet: Real-time lidar processing for drivable region segmentation on an fpga. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(5):1769–1779, 2019.

[47] Anas Mahmoud, Jordan SK Hu, and Steven L Waslander. Dense voxel fusion for 3d object detection. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pages 663–672, 2023.

[48] Daniel Maturana and Sebastian Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 922–928. IEEE, 2015.

[49] Fraser McLean, Leyang Xue, Chris Xiaoxuan Lu, and Mahesh Marina. Towards edge-assisted real-time 3d segmentation of large scale lidar point clouds. In *Proceedings of the 6th International Workshop on Embedded and Mobile Deep Learning*, pages 1–6, 2022.

[50] Russell Merrick. *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. No Starch Press, 2023.

[51] Gregory P Meyer, Ankit Laddha, Eric Kee, Carlos Vallespi-Gonzalez, and Carl K Wellington. Lasernet: An efficient probabilistic 3d object detector for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12677–12686, 2019.

[52] Sambit Mohapatra, Senthil Yogamani, Heinrich Gotzig, Stefan Milz, and Patrick Mader. Bevdetnet: bird's eye view lidar point cloud based real-time 3d object detection for autonomous driving. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 2809–2815. IEEE, 2021.

[53] Balázs Nagy and Csaba Benedek. 3d cnn-based semantic labeling approach for mobile laser scanning data. *IEEE Sensors Journal*, 19(21):10034–10045, 2019.

[54] Inc. NumFOCUS. pandas. python data analysis library. `https://pandas.pydata.org/`. [Accessed May 6, 2025].

[55] Sophie Pfister, Alberto Lerner, Abishek Ramdas, and Philippe Cudré-Mauroux. Alpha demo: A hardware-accelerated data model for ad-hoc manipulation of point clouds. In *Proceedings of the 2025 International Conference on Management of Data*, (SIGMOD'25), 2025.

[56] Hazem Rashed, Mohamed Ramzy, Victor Vaquero, Ahmad El Sallab, Ganesh Sistu, and Senthil Yogamani. Fusemodnet: Real-time camera and lidar based moving object detection for robust low-light autonomous driving. In *Proceedings of the IEEE/CVF international conference on computer vision workshops*, pages 0–0, 2019.

[57] Markus Roth, Dominik Jargot, and Dariu M Gavrila. Deep end-to-end 3d person detection from camera and lidar. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 521–527. IEEE, 2019.

[58] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 770–779, 2019.

[59] Kiwoo Shin, Youngwook Paul Kwon, and Masayoshi Tomizuka. Roarnet: A robust 3d object detection based on region approximation refinement. In *2019 IEEE intelligent vehicles symposium (IV)*, pages 2510–2515. IEEE, 2019.

[60] Martin Simon, Stefan Milzy, Karl Amendey, and Horst-Michael Gross. Complex-yolo: An euler-region-proposal for real-time 3d object detection on point clouds. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 0–0, 2018.

[61] Dennis Sprute, Florian Hufen, Tim Westerhold, and Holger Flatt. 3d-lidar-based pedestrian detection for demand-oriented traffic light control. In *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, pages 1–7. IEEE, 2023.

[62] Joanna Stanisz, Konrad Lis, Tomasz Kryjak, and Marek Gorgon. Hardware-software implementation of the pointpillars network for 3d object detection in point clouds. In *Workshop on Design and Architectures for Signal and Image Processing (14th edition)*, pages 44–51, 2021.

[63] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, et al. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2446–2454, 2020.

[64] Velodyne LiDAR, Inc., 5521 Hellyer Ave, San Jose, CA 95138. *VLP-16 User Manual*, 63-9243 rev. e edition, 02 2019.

[65] Cheng Wang, Xuebo Yang, Xiaohuan Xi, Sheng Nie, and Pinliang Dong. *Introduction to LiDAR remote sensing*. CRC Press, 2024.

[66] Fei Wang, Zhao Wu, Yujie Yang, Wanyu Li, Yisha Liu, and Yan Zhuang. Real-time semantic segmentation of lidar point clouds on edge devices for unmanned systems. *IEEE Transactions on Instrumentation and Measurement*, 72:1–11, 2023.

[67] Zhixin Wang and Kui Jia. Frustum convnet: Sliding frustums to aggregate local point-wise features for amodal 3d object detection. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1742–1749. IEEE, 2019.

[68] Wikipedia. Point cloud. `https://en.wikipedia.org/wiki/Point_cloud`, Dec 2024. [Accessed February 10, 2025].

[69] Sascha Wirges, Tom Fischer, Christoph Stiller, and Jesus Balado Frias. Object detection and classification in occupancy grid maps using deep convolutional networks. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 3530–3535. IEEE, 2018.

[70] Bichen Wu, Alvin Wan, Xiangyu Yue, and Kurt Keutzer. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 1887–1893. IEEE, 2018.

[71] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[72] Wen Xiao, Bruno Vallet, Konrad Schindler, and Nicolas Paparoditis. Simultaneous detection and tracking of pedestrian from velodyne laser scanning data. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 3:295–302, 2016.

[73] Danfei Xu, Dragomir Anguelov, and Ashesh Jain. Pointfusion: Deep sensor fusion for 3d bounding box estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 244–253, 2018.

[74] Bin Yang, Wenjie Luo, and Raquel Urtasun. Pixor: Real-time 3d object detection from point clouds. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7652–7660, 2018.

[75] Zetong Yang, Yanan Sun, Shu Liu, Xiaoyong Shen, and Jiaya Jia. Std: Sparse-to-dense 3d object detector for point cloud. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1951–1960, 2019.

[76] Maosheng Ye, Shuangjie Xu, and Tongyi Cao. Hvnet: Hybrid voxel network for lidar based 3d object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1631–1640, 2020.

[77] Addy Yeow. Bit-twist. packet generator/editor. `https://bittwist.sourceforge.io/`. [Accessed April 08, 2025].

[78] Jin Hyeok Yoo, Yecheol Kim, Jisong Kim, and Jun Won Choi. 3d-cvf: Generating joint camera and lidar features using cross-view spatial feature fusion for 3d object detection. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXVII 16*, pages 720–736. Springer, 2020.

[79] Yiming Zeng, Yu Hu, Shice Liu, Jing Ye, Yinhe Han, Xiaowei Li, and Ninghui Sun. Rt3d: Real-time 3-d vehicle detection in lidar point cloud for autonomous driving. *IEEE Robotics and Automation Letters*, 3(4):3434–3440, 2018.

[80] Jiaxing Zhang, Wen Xiao, Benjamin Coifman, and Jon P Mills. Vehicle tracking and speed estimation from roadside lidar. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 13:5597–5608, 2020.

[81] Shaoming Zhang, Tangjun Yao, Jianmei Wang, Tiantian Feng, and Zhong Wang. Dynamic object classification of low-resolution point clouds: An lstm-based ensemble learning approach. *IEEE Robotics and Automation Letters*, 2023.

[82] Xiao Zhang and Xinming Huang. Real-time fast channel clustering for lidar point cloud. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(10):4103–4107, 2022.

[83] Junxuan Zhao, Hao Xu, Hongchao Liu, Jianqing Wu, Yichen Zheng, and Dayong Wu. Detection and tracking of pedestrians and vehicles using roadside lidar sensors. *Transportation research part C: emerging technologies*, 100:68–87, 2019.

[84] Xin Zhao, Zhe Liu, Ruolan Hu, and Kaiqi Huang. 3d object detection using scale invariant and feature reweighting networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9267–9274, 2019.

[85] Yin Zhou, Pei Sun, Yu Zhang, Dragomir Anguelov, Jiyang Gao, Tom Ouyang, James Guo, Jiquan Ngiam, and Vijay Vasudevan. End-to-end multi-view fusion for 3d object detection in lidar point clouds. In *Conference on Robot Learning*, pages 923–932. PMLR, 2020.

[86] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4490–4499, 2018.

[87] Xinge Zhu, Hui Zhou, Tai Wang, Fangzhou Hong, Yuexin Ma, Wei Li, Hongsheng Li, and Dahua Lin. Cylindrical and asymmetrical 3d convolution networks for lidar segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9939–9948, 2021.

Appendix

A - Survey on Traffic-Related Utilization of LiDAR Sensors

Classification Scheme

- **Target Task**: The survey targeted semantic segmentation and object detection. However, there are a few subcategory within those classes, as well as closely related tasks, *i.e.,*
  - *Tracking* builds on top of object detection as it aims to correlate objects across a sequence of frames.
  - *Foreground segmentation* is a less complex segmentation task where the system aims to eliminate points referring to static background objects, such as the ground, buildings, vegetation, *etc.*
  - Similarly, *road segmentation* aims to identify the "driveable region" in the point cloud.
  - *Object classification* refers to the second stage of a two-stage object detection pipeline where object candidates should be assigned to one out of k classes.
  - *<CLASS> detection* refers to an object detection pipeline where only a certain class of objects should be detected.
- **Input Representation**: We found that four representations are overwhelmingly present in the surveyed works. We name them here and discuss them in detail in the Chapter 3: point clouds (P), voxels (V), frontal view (F), bird's eye view (B). The remaining representations are summarized as other (O), and we added some more details as side notes.
- **Fusion**: We recorded whether a system fused LiDAR point clouds with some other data stream, *i.e.,* camera images or GIS data.

Year	**System/Paper**	**Task**	**Representations**		**Fusion**
2011	Kidono *et al.* [31]	Pedestrian Detection	O	Cluster	
2015	VoxNet [48]	Object Classification	V		
2016	Xiao *et al.* [72]	Pedestrian Detection, Tracking	O	Project to (x, y, t)	
2017	MV3D [14]	Object Detection	F, B		x
	Vote3Deep [19]	Object Detection	V		
2018	SqueezeSeg [70]	Foreground Segmentation	F		
	MMVD [5]	Object Detection	F		x
	VoxelNet [86]	Object Detection	V		
	PointFusion [73]	Object Detection	V		x
	Yolo3D [1]	Object Detection	B		
	BirdNet [7]	Object Detection	B		
	Complex-YOLO [60]	Object Detection	B		
	PIXOR [74]	Object Detection	B		
	RT3D [79]	Object Detection	B		
	Wirges *et al.* [69]	Object Detection	B		
	AVOD [34]	Object Detection	B		x
	Lin *et al.* [41]	Object Detection	O	Cluster	
	FaF [45]	Object Detection, Tracking	B		
2019	3D CNN [53]	Semantic Segmentation	V		
	ChipNet [46]	Road Segmentation	F		
	PointRCNN [58]	Object Detection	P		
	RoarNet [59]	Object Detection	P		x
	F-ConvNet [67]	Object Detection	P		x
	SIFRNet [84]	Object Detection	P		x
	STD [75]	Object Detection	P, V		
	LaserNet [51]	Object Detection	F		
	Roth *et al.* [57]	Pedestrian Detection	V		x
	PointPillars [35]	Object Detection	B		
	SCANet [44]	Object Detection	B		x
	DetNet + ConNet [32]	Object Detection	P, B		
	MMF [39]	Object Detection	F, B		x
	3-MLP [83]	Object Detection, Tracking	O	Cluster	
2020	MVF [85]	Object Detection	P, V		
	HVNet [76]	Object Detection	V		
	BirdNet+ [6]	Object Detection	B		
	3D-CVF [78]	Object Detection	B		x
	Zhang *et al.* [80]	Object Detection, Tracking	O	Cluster	
2021	CA3DC [87]	Semantic Segmentation	V		
	RangeDet [20]	Object Detection	F		
	BEVNet [42]	Object Detection	B		
	BEVDetNet [52]	Object Detection	B		
2022	RangeSeg [13]	Semantic Segmentation	P, F		
	VMFV [21]	Object Detection	F, V		
	PDV [28]	Object Detection	V		
	VFF [38]	Object Detection	V		
2023	PointLE [81]	Semantic Segmentation	P		
	RangeFormer [33]	Semantic Segmentation	F		
	CPSeg [37]	Semantic Segmentation	F		
	DVF [47]	Object Detection	V		
	MENet [29]	Object Detection	B		x
	PV-RCNN [15]	Object Detection, Tracking	P, V		
	HistoGrid [9]	Object Detection	F, B, O	Cluster	
	Sprute *et al.* [61]	Pedestrian Detection	F, O	Cluster	
2024	ScorePillar [12]	Object Detection	B		
	Point-DETR3D [24]	Object Detection	B		x