Improving autonomous driving systems with CPU extensions for point cloud processing

Pedro Henrique Exenberger Becker

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Improving Autonomous Driving Systems with CPU Extensions for Point Cloud Processing

Pedro Henrique Exenberger Becker

Advisors: Dr. José-Maria Arnau
Prof. Antonio González

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

July 2024

Acknowledgements

When I was around twelve - for reasons I do not remember - I thought it would be cool to design computers. And ever since, this has been my plan. Seventeen years later, I am defending my PhD in Computer Architecture. And I am quite happy about it.

Many times since college, however, I worried of failing or not knowing enough. I would sometimes re-read random chapters in the Patterson and Hennessy book just to keep it fresh, a seminal paper with concepts I wish I had more clear, or even some very informative Wikipedia entries. Research changed it all, however. From the time I was introduced to research, still as an undergrad student in Brazil around 2015, until now, I figured out it was impossible to know everything. Even better, I learned knowledge evolves. And it was a relief. As long as I could (re-)learn something, I found, it would be fine. And this is why I am very happy and proud about the thesis that comes in the following pages. Apart from the scientific discoveries we present, which I truly believe are relevant contributions to the field, it is also a proof to myself that I can learn new concepts, tools, and related work to find open problems and potential solutions. It is a indication to my twelve-year-old self that I am going in the right direction.

For the aforementioned reasons I would like to thank my advisors Prof. Antonio González and Dr. José-Maria Arnau, for their guidance throughout these four and a half years of PhD. They many times guided my focus to where it was relevant, while also giving me space to come up with my own ideas. This thesis would not exist without their devoted time for meetings, improvement suggestions, and technical discussions. Also, thank you for trusting in me. In my opinion, the trust of the advisors is the most important thing for a PhD student, and I feel I always had your trust to explore ideas, experiments, tools, related work, and that was the main motivation for my PhD journey.

I would also like to thank the "*lab mates*" with whom I shared the ARCO group at UPC for all these years. Albert, Andreas, Aurora, Bahareh, David, Dennis, Diya, Franyell, Ignacio, Imad, Jorge, Marc, Martí, Mehdi, Mojtaba, Mohammad, Nicolas, Nitesh, Puria, Pratyush, Raúl, Rodrigo. Particular thanks for Franyell, Marc, and Raúl for our almost daily coffee at

10 in the morning, where a surprisingly high amount of our PhD problems would be solved with each others help.

I would also like to thank my parents Raquel and Denis for their lovely support 24/7. Somehow you managed to make me feel safe and cared for even if living overseas. I always felt like you would catch the next plane and come for me if I needed to. That was never necessary, thankfully, but I am sure you would if needed, and that is much much more than I could ever ask you to. Thanks to my brothers as well, Thales and Augusto, for also supporting me and giving advice whenever I needed it.

Thank you to all my friends who did not even know they were helping but whose friendship was heartwarming. Particularly, thank you, Bruno and Gustavo, for being my best friends since the time I first dreamed about study computers, and for coming to Spain to visit. Thank you Brunno for the almost daily conversations about everything, any time. Thank you Arthur and Gustavo for the amazing friendship we built in a bit more than a year here in Barcelona.

Finally, thank you Míriam for your infinite amount of support, care, and love. Meeting you was the best part of these years in Barcelona. I love you.

Funding Acknowledgements

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grants PID2020-113172RB-I00 and TIN2016-75344-R (AEI/FEDER, EU), the Catalan Agency for University and Research (AGAUR) under grant 2021SGR00383, the ICREA Academia program, and the FPI-UPC 2020 grant.

Abstract

Autonomous Driving Systems (ADS) are at the cusp of large-scale adoption, promising accident reduction and market potential. However, the complex software and sensor data pressure for better hardware support in this safety-critical scenario, where high performance is mandatory to meet latency deadlines. Additionally, energy efficiency, cost, and volume must also be first-class for market feasibility, calling computer architects into action.

To enrich hardware support for ADS, we carry out a performance and power characterization of Autoware.ai, a state-of-the-art ADS software stack. We find significant time spent processing Light Imaging Detection and Ranging (LiDAR) sensor data, which are widely used by ADS. LiDAR captures Three Dimensional (3D) point clouds for tasks such as segmentation, localization, and object detection.

Despite its importance, hardware support for LiDAR has only recently gained traction. Further, while most point cloud processing algorithms run on Central Processing Units (CPUs), recent works propose costly hardware accelerators. Instead, we aim to use existing general-purpose hardware and software for point cloud processing with minor CPU augmentations. For that, we introduce a small set of CPU instructions targeting point cloud neighbor search based on k-d trees, a key operation used in various algorithms.

The first technique we propose is K-D Bonsai, which reduces data movement during the neighbor search by compressing k-d tree leaves in execution time, exploiting value similarity. K-D Bonsai further compresses the data using a reduced floating-point representation, exploiting the physically limited range of point cloud values collected with LiDAR. We implement K-D Bonsai through a small set of new CPU instructions to compress, decompress, and operate on points. To maintain baseline accuracy, we carefully craft the instructions to detect precision loss due to compression, allowing re-computation in full precision to take place if necessary. Therefore, K-D Bonsai reduces data movement, improving performance and energy efficiency while guaranteeing baseline accuracy and programmability. K-D Bonsai improves the end-to-end latency of the segmentation task of Autoware.ai by 9.26% on average, 12.19% in tail latency, and reduces energy consumption by 10.84%. Unlike the expensive accelerators proposed in related work, K-D Bonsai improves neighbor search with minimal area increase (0.36%).

In the second technique, we found that consecutive neighbor search queries are often similar, visiting k-d tree nodes with considerable resemblance. We leverage this observation to cheaply speed up neighbor search with the available CPU Vector Processing Unit (VPU). We propose a hardware/software co-design called Caravan. At the software level, Caravan-SW exploits search similarity, gathering consecutive queries to search for their neighbors in parallel with Single Instruction Multiple Data (SIMD) instructions. Yet, when the navigation of queries diverges, particularly in the deeper levels of the k-d tree, Caravan-SW faces sparsity and the VPU lanes are underutilized. We tackle this with Caravan-HW, adding two new instructions that re-index valid vector elements and allow fast operand shuffling and dense SIMD operations to take place, suppressing the hard-to-predict runtime sparsity of Caravan-SW. With AVX512, Caravan-SW speeds up neighbor search by 4.05× (1.85× end-to-end) in Autoware.ai point cloud segmentation. With the additional Caravan-HW support, the leaf processing part of neighbor search can be further optimized, boosting gains to 5.19× (1.97× end-to-end), with minimal area costs. Our programmable and minimally intrusive solution has end-to-end benefits comparable to accelerators.

Table of Contents

List of Figures

List of Tables

Acronyms

kNN	K-Nearest Neighbors.
2D	Two Dimensional.
3D	Three Dimensional.
AD	Autonomous Driving.
ADS	Autonomous Driving Systems.
ASIC	Application Specific Integrated Circuit.
AV	Autonomous Vehicle.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
DARPA	Defense Advanced Research Projects Agency.
DNN	Deep Neural Network.
EN	Euclidean Neighbors.
FoV	Field of View.
FP	Floating-Point.
FPGA	Field-Programmable Gate Array.
FPS	Frames per Second.
FU	Functional Unit.

GNSS	Global Navigation Satellite System.
GPU	Graphics Processing Unit.
HBM	High Bandwidth Memory.
HD	High-Definition.
IMU	Inertial Measurement Unit.
IP	Intellectual Property.
ISA	Instruction Set Architecture.
LiDAR	Light Imaging Detection and Ranging.
LSB	Least Significant Bit.
ML	Machine Learning.
NHTSA	National Highway Traffic Safety Administration (US).
OoO	Out-of-Order.
PAPI	Performance Application Programming Interface.
PCL	Point-Cloud Library.
R&D	Research & Development.
RAPL	Running Average Power Limit.
ROS	Robot Operating System.
SAE	Society of Automotive Engineers.
SIMD	Single Instruction Multiple Data.
SoC	Systems on Chip.
TPU	Tensor Processing Unit.
UKF	Unscented Kalman Filter.

VL Vector Length.

VPU Vector Processing Unit.

VR Virtual Reality.

WHO World Health Organization.

1 Introduction

“The future of this new technology is so full of promise. It’s a future where vehicles increasingly help drivers avoid crashes. It’s a future where the time spent commuting is dramatically reduced, and where millions more—including the elderly and people with disabilities–gain access to the freedom of the open road. And, especially important, it’s a future where highway fatalities and injuries are significantly reduced.”

National Highway Traffic Safety Administration
US Department of Transportation [103]

1.1 Applications and Benefits of Autonomous Vehicles

The expectation for Autonomous Vehicles (AVs) to take the streets as real-life products took decades, but it is finally starting to happen [14, 17, 42, 98, 106, 108, 144, 159]. While the concept of an automatically driven vehicle – and the associated benefits – has been idealized for long, only recently has the technology started to reach the huge computation capabilities that are required for Autonomous Driving (AD) [86, 125, 156].

The applications are several, ranging from personal use [42, 144] to vehicle-sharing services [16, 42, 159], delivery [106], or smart cities [124], the technology that allows machines to navigate autonomously has the power to revolutionize society [75]. Arguably, its main appeal is the expected, and needed, reduction in fatal traffic accidents. According to the World Health Organization (WHO) [164], traffic accidents account for 1.2 million deaths and from 20 to 50 million injuries, with many incurring a disability, worldwide every year. It is also the leading cause of death for children and young adults aged 5–29 years. Meanwhile, according to the US Department of Transportation, 94% of fatal crashes in the US are due to human error [103]. This puts pressure on finding solutions to vanish traffic accidents with technology help. In this regard, AVs are expected to play a vital role. Equipped with multiple sensors, these vehicles continuously capture massive data from the driving scene, perceiving every detail with a 360° Field of View (FoV). The sensed data is then processed by a combination of algorithms that aim to make the driving decisions as accurate as possible. The final goal is not only to drive as well as humans, but much better than them, removing human error factors, and significantly reducing fatalities and injuries [103].

In fact, the track record of AVs safety is promising. In a recent article [78], Waymo, one of the pioneer companies in the field, compared crashes data between their AV solution against human drivers. They consider more than 7 million miles driven by Waymo vehicles, and declared an 85% reduction in crashes with some injury reported, and a 57% reduction in crashes that needed to be reported to the police. Although their study is limited i) to the cities where they operate; ii) to scenarios with a maximum speed of 50 mph (around 80 km/h); and iii) not including severe weather conditions (e.g., thick fog or heavy rain), it shows the potential AVs have to drastically reduce accidents.

While the safety benefits by themselves already incur aggregated value, AVs have other features that also enhance their commercial appeal. Particularly, the absence of a human driver is a key advantage, as the target market enlarges. Anyone can benefit from door-to-door journeys, with no license needed. Night trips can be carried without a concern that the driver will fall asleep. The time to find a parking spot is no longer a passenger's problem. These and other improvements on mobility motivate multiple companies to work on making AVs a reality, including Baidu [17], Cruise [42], Daimler [43], Mobileye [98], NVIDIA [108], Tesla [144], TierIV [147], and Waymo [159], to name a few.

These companies focus on applying AVs technology to improve existing services and people's lives. For example, different companies such as Alphabet's Waymo [159], Baidu's Apollo [16], and General Motors' Cruise [42] have started to operate *robotaxis*, a neologism to describe ride-hailing services operated with AVs. By using AVs, they can reduce operational costs, optimize resource allocation, and even reduce environmental impact [48, 77]. This in turn can convert into lower fares and standardized user experience, resulting in better service for customers. Removing the driver also brings unexpected but positive side-effects such as higher passenger safety. In 2020, for example, 998 Uber drivers were reported for sexual assault in the United States, according to a recent safety report by Uber [150].

Expanding the examples, the AV technology can also be employed for delivery. The company Nuro [106] targets last-mile distribution and converts the needless passenger seats into storage volume, maximizing the utility of the vehicle's space for their services. For long-distance delivery, Daimler Trucks [43] expects AD systems to improve their logistic performance by enabling trucks to operate continuously. With a less obvious application, a start-up [124] is embedding AV technology to automatize cities' waste collection. They projected waste containers embedded with AD capabilities. Containers that are full self-drive to the waste center, and a new one self-drives to replace the former. This avoids the inconveniences of traditional waste collection with garbage trucks such as full containers waiting to be replaced, noise, and traffic jams.

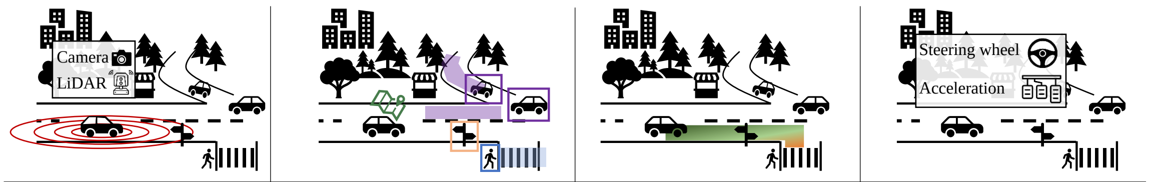
1.2 Computer System for Autonomous Vehicles

AVs depend on computers to perform the driving. Following, we provide a brief overview of the workloads they need to execute, followed by a number of computer architecture challenges associated with it. We then briefly narrow the discussion to Light Imaging Detection and Ranging (LiDAR) sensors, a sensor used by AVs, and how processing its sensed data is key for AVs, requiring better hardware support.

1.2.1 Functional Requirements

The concept of Autonomous Driving Systems (ADS) evolved since its early years of research (1980's). At that time, some researchers pursued the development of automated highway systems, in which vehicles depend significantly on the highway infrastructure to guide them [2]. In the 2000s, however, there was a convergence towards enabling the vehicles to individually make their driving decisions, revoking external infrastructure support. This allows an organic adoption of AVs, which can incrementally share the road with human drivers. It also lowers market risks as adapting vehicles to the current cities is more likely to be accepted than adapting the cities for them. When this was shown to be a possibility, particularly after the proof of concept in the Defense Advanced Research Projects Agency (DARPA) challenges [24, 151], the AV technology faced a new horizon whose chase led to the scenario we have today: we are in the cusp of AD adoption [2, 75].

In this paradigm, we need embedded computer systems in the vehicle to perform AD. Just like a human driver, therefore, the system needs to ***sense*** the environment, ***understand***



① The vehicle's sensors collect data from the environment.

② Scene recognition process sensed data to identify and track traffic elements, and localize the vehicle.

③ Predictions, destination, and traffic rules guide planning algorithms to calculate upcoming route and speed.

④ The computer maneuvers the vehicle, adjusting the steering wheel and throttle position so it drives according to the computed decisions.

Figure 1.1 The computation pipeline for the AD task. AV typically performs four major steps: sensing, scene recognition, planning, and control.

it, ***plan***, and ***control*** the vehicle. Figure 1.1 illustrates these steps. All we have to do, therefore, is to transpose this human task to a computer task, hereby referred to as the *AD computing pipeline*. Easier said than done. We nevertheless try to break down this transposition following. In time, we indicate that a broader and deeper explanation of an AV system can be found in Chapter 2, with a dissected real-life example. For now, we provide a high-level explanation to motivate the discussion. While most companies do not share their implementation details, we base ourselves on a structure typically reported by the contemporary literature and state-of-the-art open-source projects [4, 68, 86, 88, 168, 169, 172].

Getting started with our transposition analogy, the ADS needs to ***sense*** the environment (step ① in Figure 1.1). For this, they are equipped with multiple sensors including cameras, LiDAR, radar, Global Navigation Satellite System (GNSS), and Inertial Measurement Unit (IMU). This rich data collection is the starting point for the subsequent step, ***scene recognition***, where a complex chain of algorithms perform, among others, object detection and localization (step ② in Figure 1.1). For object detection, algorithms identify static traffic elements (e.g., roads and traffic lights) and dynamic traffic participants (e.g., a pedestrian, a car, etc), classifying them according to their role in the traffic scene, and adding semantics to the perceived environment. Subsequently, other algorithms are used to track each of these participants, inferring their position and speed, and predicting their route for the near future. Concurrently, the vehicle utilizes sensed data to determine its own position. The outcomes of the different algorithms are then fused into a unified representation of the driving scene. Afterward, the ADS can utilize ***planning*** algorithms (step ③ in Figure 1.1) to perform short-term decisions (e.g., avoid an obstacle) and long-term decisions (e.g., optimize the route to destination). Finally, the planned decisions are converted into actions through ***control*** commands (step ④ in Figure 1.1), such as turning the steering wheel or activating the throttle.

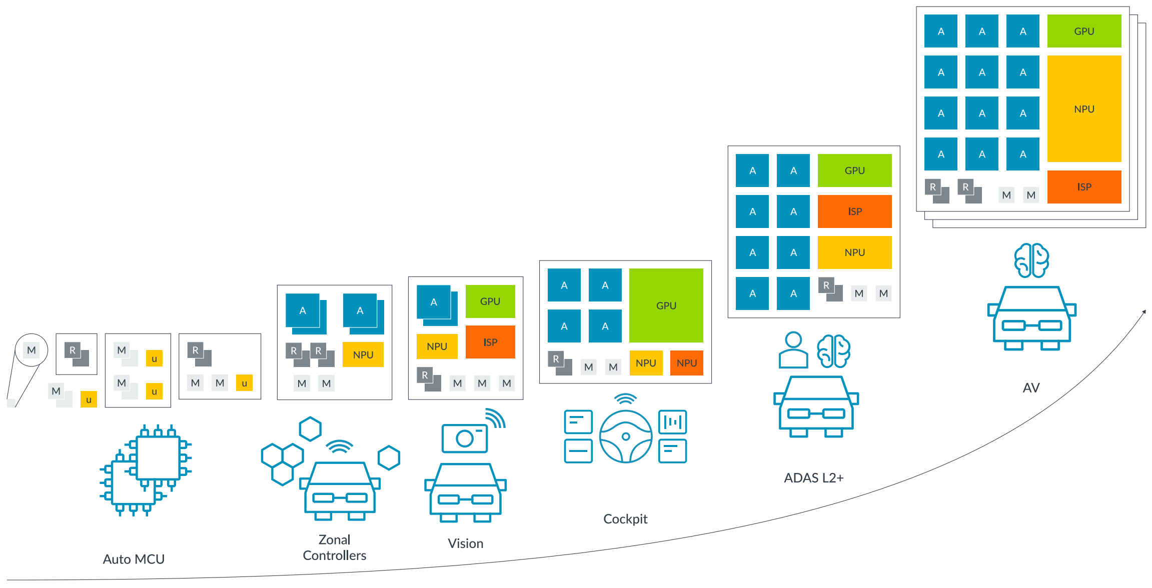


Figure 1.2 Use cases of ARM Processors and IPs for Automotive, adapted from [8]. The higher the processing power the higher the aid from the computer platform in the driving task, from indicators to the drivers, to driving assistance, and finally fully AV. In the figure Central Processing Units (CPUs) can be A-class (high performance), M-class (energy-efficient), or R-time (real-time).

The more advanced the implemented driving features are, the more situations the AV can handle. Figure 1.2 illustrates this concept based on use cases for Arm processors and Intellectual Property (IP) targeting automotive [8]. These *'driving skills'* are referred to as the Level of Autonomy, which is a relevant metric to indicate how mature a given ADS is, and also to compare the proficiency of different AD solutions. A standard taxonomy to categorize ADS is maintained by the Society of Automotive Engineers (SAE) [139], which is adopted by relevant government traffic agents guidelines such as the National Highway Traffic Safety Administration (US) (NHTSA) policy framework [103]. Table 1.1 summarizes the six possible categories for driving automation defined by SAE. It varies from no autonomy (level 0) to full autonomy (level 5). The latter is the target level of automation for AVs, where the vehicles may not even have a steering wheel given that the ADS can completely handle all driving scenarios, at all times and conditions. However, as we discuss next, achieving the highest levels of autonomy (4 or 5) is a defiant goal.

1.2.2 Architectural Challenges

Coping with *all* possible driving scenarios is very challenging as even skilled human drivers are susceptible to accidents. Notwithstanding, AVs need to surpass human driving skills

Table 1.1 Levels of automation based on SAE J3016 [139].

Automation Level	**Description**
0 - No Automation	Human driver is in complete control of all aspects of driving.
1 - Driver Assistance	Vehicle assists with either steering or acceleration, but not both simultaneously, and the driver still performs most of the driving.
2 - Partial Automation	Vehicle can control both steering and acceleration simultaneously under certain conditions, but the driver must remain engaged and monitor the environment.
3 - Conditional Automation	Vehicle can perform all driving tasks under certain conditions, but the driver must be prepared to take over when requested.
4 - High Automation	Vehicle can perform all driving tasks under certain conditions, without any human driver intervention.
5 - Full Automation	Vehicle can perform all driving tasks under all conditions, without any need for human intervention.

to be justified and gain public acceptance [73], which requires a lot of processing power from their embedded computers [63]. For example, a work on the speed of processing in the human visual system [145] had shown that the fastest reaction time of a human is around 150 ms. Consequently, studies generally pose a reaction time constraint of 100 ms for AVs [39, 86], a tight interval to encompass the driving pipeline from sensing to acting. Different studies – including contributions from this thesis – have found high-end platforms to be incapable of meeting such constraints at all times [19, 69, 172]. Hence, reducing latency to compute AD algorithms is paramount.

There is also substantial pressure for higher performance considering ADSs are increasingly adopting sensors to enhance their perception abilities and achieve higher levels of autonomy (e.g., level 5). In 2020, Waymo gave details about their vehicles' sensors [158], reporting the use of 29 cameras, 5 LiDARs, and 6 radars in a single vehicle. This directly impacts the amount of data to process, pressuring the hardware for greater computation power. In the same trend, a white paper by Huawei [156] stated that computer performance is the limiting factor for AD, forecasting a necessary improvement of 10$\times$ to transition from level 4 to level 5 autonomy.

However, despite industry progress, no company or academic study has reported achieving level 5 autonomy yet. Notwithstanding, this should motivate research since previous advancements in autonomy levels were obtained through industry and academia efforts for improving sensors, algorithms, and hardware. Particularly, hardware improvements were important enablers for advances in ADS so far. Indeed, due to the importance of building and optimizing the computing system, some AVs companies reported spending around 50% of their Research & Development (R&D) budget on it [168], placing computer architects and hardware engineers as a mainstay for assuring well-suited hardware for AD.

Waymo, for instance, discussed how a 10× boost in their CPU performance, from 2012 to 2017, allowed them to cut-through levels of autonomy [125]. They also affirm that improving computer performance is required to drive fully autonomously on complex city streets and for a smoother and more robust driving experience. Similarly, Mobileye recently announced their new Systems on Chip (SoC) EyeQ6H, tailored for AD [97]. According to them, the EyeQ6H boosts the computing power against its predecessor EyeQ5H by 3×. Such improvement was one of the pillars for Mobileye to move from level 2 driving assistance to level 4 AD. Consonant, Nuro announced a partnership with Arm, to leverage recent Arm CPUs targeting AD [8, 9]. These CPUs are designed specifically to execute AVs applications, hence to be employed on upcoming, improved versions of the Nuro Driver™.

While increasing the performance of the computing platform is a top priority, other relevant facets must also be accounted for. Energy consumption, for instance, is another critical design concern. If energy consumption is too high, the operational range of the vehicle will be reduced [86, 88, 156]. This is particularly problematic for electric vehicles, which gained traction due to their reduced carbon footprint, but which are also known to struggle with battery limitations [32]. On the other hand, because of the strict performance requirements, current AD solutions rely on power-hungry hardware, including a combination of high-end CPUs and Graphics Processing Units (GPUs) [69, 125, 168]. Thus, not only do they harm battery duration on their own, but also incur power dissipation that adds extra cooling necessities, further increasing energy consumption [86, 156].

Notwithstanding, achieving energy efficiency is not the only challenge that computer architects need to cope with when designing performing hardware for AVs. Simplistic solutions such as adding more hardware increase cost and occupied volume, which are relevant concerns in some cases. Last-mile delivery vehicles [82], for example, are constrained by size as they are generally smaller than those carrying passengers [30]. In general, considering the aforementioned constraints, it is necessary to find ways of improving performance with minimal overheads in other hardware requirements.

One potential approach for this is to utilize hardware accelerators. In this case, one can map software that executes on a general-purpose CPU to a more specific hardware such as GPUs or custom Application Specific Integrated Circuits (ASICs). Let's take the realm of Deep Neural Networks (DNNs) to exemplify. We will see that different hardware accelerators have been proposed in recent years [114], surpassing performance and/or energy-efficiency of CPU and even GPU counterparts, for tasks such as object detection in images. However, adopting application-specific hardware in the computing system platform leads to integration and development challenges.

For example, the latency overheads for off-loading tasks to the accelerator should be less than the reduced computation time, which is not always the case [59, 69, 168]. Moreover, GPU acceleration is only suitable when the target application can benefit from parallelism [1, 80, 113]. On the other hand, ASICs cannot serve any other purpose other than what was taped out in the chip. This reduces their adaptability to new problems, which is still common on AD domain, where algorithms and requirements are subject to change [168]. Alternatively, updating ASIC hardware with a new one hurts time-to-market [96], and also incurs significant expenses since the costs for taping out silicon chips are enormous [115]. Also, due to their lack of generality, hardware accelerators need to be very well justified since they can occupy even more silicon area than a CPU [18], while attending *much* fewer use cases. Therefore, although a good alternative to having up one's sleeve, hardware accelerators are suited for well-defined problems, where they are known to be a long-term solution.

Otherwise, employing general-purpose hardware such as CPUs provides a more flexible solution, covering different uses and reducing development risks. With them, adjusting to a new algorithm is as simple as compiling a new code. This adaptability also allows the CPU to execute different tasks across the AD computing pipeline, one after the other, maximizing silicon utilization. However, as we discuss in Chapter 2 of this thesis, CPUs are sometimes insufficient to execute some AD tasks [148]. Nevertheless, we believe there is potential to refine them for the AD domain, as the industry appears to be doing [8], to take advantage of its virtues as a whole.

1.2.3 Help Wanted: Computer Architects for LiDAR's Point Cloud Processing

The open challenges for ADSs have a root in the need to process a high amount of sensor data with complex algorithms. In the early days, meeting the functional requirements was the main goal. Solutions were tested with big computing prototypes at low speed [12, 15], and the main problem was to make sure the algorithms were computing the right thing, and driving as expected [63]. As functional requirements became more mature, concerns shifted towards meeting the non-functional requirements such as latency, energy, power, area, and cost. For that, the AV problem became more interdisciplinary, requiring attention

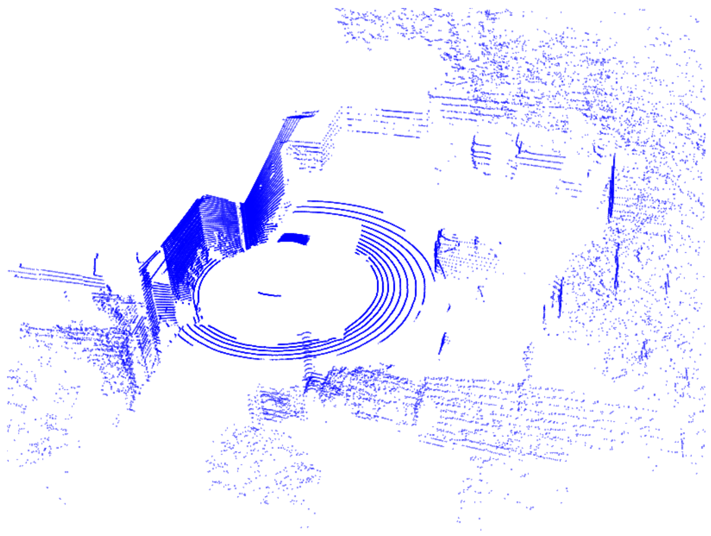


Figure 1.3 An example of a point-cloud obtained with LiDAR. Data from [67].

from computer architects to identify their main bottlenecks and come up with solutions for them.

Contemporary literature – including contributions from this thesis – has found that from all steps in the AD computing pipeline, ***perception*** is the most time and resource-consuming one [19, 172]. To aid perception, the ADS rely on data captured by sensors, processing them to detect and interpret the elements of the traffic scene, such as pedestrians, cars, traffic signs, etc. Although different sensors exist, cameras and LiDAR are the most commonly used as they are powerful and versatile. Given their importance, we briefly explain them following.

Starting with the best-known, cameras are passive sensors that capture images. For such, they have lens to project incident light into a grid of light-sensitive diodes [31]. The diodes measure the intensity of red, green, and blue lights from source light rays to set the resulting color in each image position (pixel). Differently, a LiDAR is an active sensor capable of collecting a Three Dimensional (3D) representation of the environment (see Figure 1.3), featuring accurate depth information that is not featured in its Two Dimensional (2D) counterparts (e.g., images from cameras). Their working principle is to shoot laser beams around and measure the round trip time for the beam to reflect back to the sensor [85]. In the absence of sensing noise, each sensed point belongs to a surface in the real world (of a wall, a car, a tree, etc.) within the sensor range. The set of collected points is referred to as a ***Point Cloud***.

As illustrated in Figure 1.4, each different sensor has its particular advantages for the driving task. For this reason, they are generally combined to cover up each other's

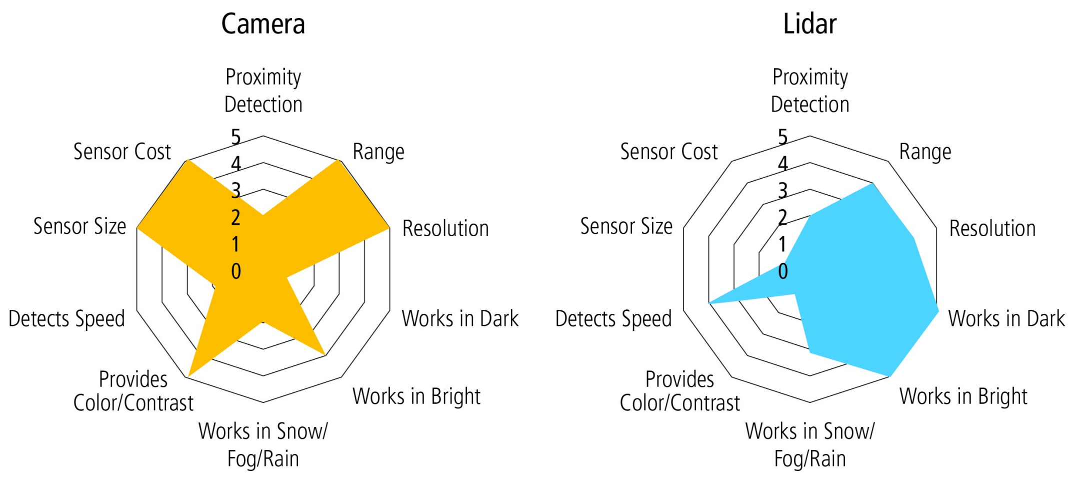


Figure 1.4 Strengths and Weaknesses of Camera and LiDAR for ADS. Adapted from [163].

deficiencies and improve overall perception accuracy [121, 148, 169]. Cameras, for instance, have excellent resolution and range but struggle in very dark or very bright situations. Nevertheless, they are cheap and can be used in generous quantities to capture as much detail as possible from the driving scene [158]. Their low price comes from their popularity since their use is broader than just on AV [137]. Because of the widespread usage of cameras, hardware support for image-based algorithms has been extensively studied and even precedes AVs [18, 33, 36–38, 61, 66, 70, 91, 99, 102]. Therefore, system designers have a plethora of hardware and software options to choose to perform image-based AD tasks (e.g., object detection).

On the other hand, LiDARs afford accurate 3D representation of the environment, wide FoV, high angular resolution, and independence from light conditions, with drawbacks of struggling under heavy rain, snow, or fog, and also being more expensive than cameras [171]. Nevertheless, LiDAR sensors are essential for AD being widely employed for some of its major ***perception*** tasks such as segmentation, 3D object detection, and localization [4, 13, 27, 28, 53, 54, 71, 79, 93, 105, 129]. Given their accuracy and versatility, they play a key role in achieving higher levels of autonomy. This can be observed as AVs equipped with LiDAR report higher levels of autonomy compared to AVs that do not use them. To the best of our knowledge as of the writing of this document, all AVs with level 4 autonomy employ LiDAR sensors. On the contrary, camera-only approaches, are arguably falling behind [76].

Before found to be indispensable for AVs, however, LiDARs had niche applicability. They were far from their current spotlight, with little permeability to the general public usage, instead being used, for example, on aerial mapping or atmospheric aerosols studies [128]. Differently from AD, these LiDAR applications had more relaxed constraints and arguably lower market appeal. Therefore research on improving performance to achieve safety-critical or real-time requirements for LiDAR processing was limited if existent, particularly from the computer architecture community. But this changed with the rise of ADS, where LiDAR processing needs to be first-class regarding performance and energy efficiency. At the same time, recent works have found that modern systems cannot achieve the desired non-functional requirements, especially latency, for processing LiDAR point clouds [19, 69, 172]. As a consequence, better hardware support for LiDAR processing gained importance recently, and some works in computer architecture targeting point clouds started to appear [35, 49, 87, 116, 140, 165, 167].

Notwithstanding, there is still a great imbalance between hardware support for camera versus LiDAR-based applications. Existing related works for the latter mostly propose custom ASICs, with solutions that hurt development costs and programmability. Controversially, the industry favors less experimental approaches to accelerate tasks in their real-life solutions, preferring traditional general-purpose options [114]. Nevertheless, very few works put effort into tailoring general-purpose architectures, such as CPUs, to ameliorate LiDAR's point cloud processing. We believe this is a missing opportunity. CPU's are ubiquitous and versatile, and currently the main hardware platform employed for LiDAR-based AD algorithms [53, 54]. In this sense, finding ways of tuning CPUs for LiDAR's point cloud processing can have a great impact in upcoming AVs technology, especially if the solution is minimally intrusive, performing, energy efficient, and programmable.

1.3 Problem Statement, Objectives, and Contributions

The future of ADS depends on hardware progress. Foremost, performance improvements are necessary to reduce reaction time and to cope with more sophisticated algorithms increasing safety and achieving higher levels of autonomy. The refinements also need to consider other constraints such as energy efficiency, for a longer operation range, low area, for minimal silicon cost and maximal space utilization, and programmability, for adaptability to new algorithms and time-to-market. All these requirements must be contemplated to unleash next-generation AVs technology.

Acknowledging these constraints, this Ph.D. thesis contributes with novel computer architecture findings, helping the development of the AV technology. For this, we first identify relevant research paths for computer architecture in the ADS domain. Through literature review and hands-on characterization of a real-life software stack for AD (Chapter 2), we identify that improving LiDAR's point cloud processing is a key research topic. The point clouds captured by these sensors are the input for a multitude of AD algorithms, some of which account for significant execution time and computing resources, being the

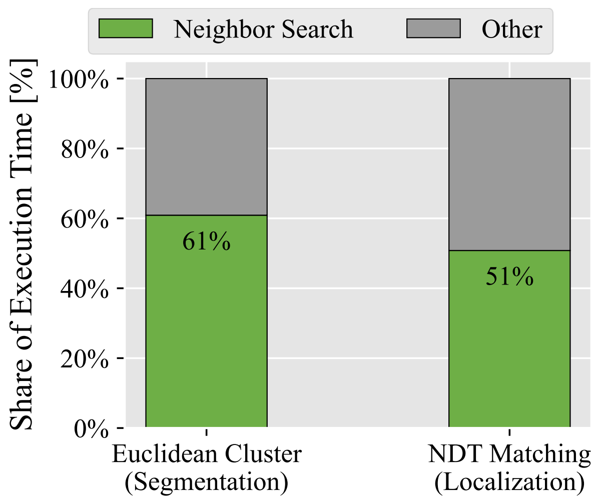


Figure 1.5 The share of execution time devoted to neighbor search in two key LiDAR-based algorithms used in AD. Measured on an Intel®Core™i5-10210U CPU.

main bottleneck of the computing system. Hence, we advocate for improving LiDAR-based applications that process point clouds. By investigating it further, we found point cloud processing to heavily depend on the *neighbor search* sub-task. As we detail in Chapter 3, this is a crucial part of point cloud processing that searches for nearby points with respect to a query point. Cumulatively, *neighbor search* takes over half of the point cloud processing execution time for crucial LiDAR-based algorithms such as segmentation and localization, as shown in Figure 1.5. As we also explain in Chapter 3, the used data structures and characteristics of neighbor search justify why point cloud processing is typically performed on CPUs, as we mentioned in the previous section.

In this context, the main objective of this thesis is to identify areas of improvement and enhance current computer architectures for AD tasks, tailoring hardware platforms for better performance and energy efficiency. Specifically, we narrow our focus to LiDAR's point cloud processing aiming to fill research gaps for such a relevant part of ADS that, in our view, has been under-explored. For such, our objective is to deeply understand the key neighbor search operation that is widely used in LiDAR-based AD applications. We aim to exploit its characteristics to fine-tune general-purpose hardware for point cloud processing. Particularly, we focus on exploring (micro)architectural improvements upon contemporary CPUs to enhance point cloud processing with reduced execution time and energy consumption, hold programmability, and be as minimally intrusive as possible on existing CPUs, making our proposals appealing for next-generation CPUs for ADS.

Following, we detail the contributions of this thesis while pursuing the aforementioned objectives. They were submitted for peer review, in the form of three conference articles and two poster presentations, as indicated in each subsection.

1.3.1 Characterizing a State-of-the-Art Software Stack for Autonomous Vehicles

We start by presenting a thorough characterization of an AD solution, detailing open problems in current software and hardware for future research on AVs. The investigation is performed with a modern and fully open-sourced solution, namely Autoware [10, 14], which is built upon cutting-edge algorithms for AVs. We stimulate Autoware's software stack with real-life sensor data and profile several of its traits. The measurement includes the individual computation latency of different modules, the end-to-end latency from sensor data collection up to complete scene recognition, and the power and energy required for those tasks. It also encompasses data from architectural hardware counters such as instruction mix, branch misprediction, and cache miss ratio that are discussed to understand how the different algorithms stress the computing platform. Some of the contributions are the following.

- We provide a didactic overview of typically employed algorithms for AVs.
- We discover that LiDAR-related components, which are key to driving the car safely, are important contributors to end-to-end latency, showing execution times in the order of tens of ms.
- We find that Autoware cannot guarantee real-time perception on a modern computer with a high-end GPU, as its end-to-end latency frequently exceeds time requirements by more than twofold.
- We observe that camera-based applications such as object detection DNN make significant use of the GPU, which prevents the GPU to be employed in other tasks such as LiDAR-based ones.
- We observe that GPUs consume significantly more power than CPUs, which should make system designers prefer the latter when considering the energy constraints of ADS.

These findings can also be found in the following works.

- Pedro H. E. Becker, José-María Arnau, and Antonio González. Demystifying Power and Performance Bottlenecks in Autonomous Driving Systems. In *Proceedings - 2020 IEEE International Symposium on Workload Characterization, IISWC 2020*, pages 205–215. IEEE, oct 2020. ISBN 9781728176451. doi: 10.1109/IISWC50251.2020.00028. URL https://ieeexplore.ieee.org/document/9251251/

- Pedro H. E. Becker, José-María Arnau, and Antonio González. Characterizing Self-driving Tasks in General-purpose Architectures. In *ACACES 2021 Poster Abstracts*, pages 117–120. HiPEAC, the European Network of Excellence on High Performance Embedded Architecture and Compilation., 2021. ISBN 978-88-905806-8-0

1.3.2 Improving Support for LiDAR's Point Cloud Processing on CPU

Motivated by the findings reported in the previous Section 1.3.1, and the discussion in Section 1.2.3, we narrow our efforts into improving CPUs support for LiDAR's point cloud processing. As we detail in Chapter 3, the point cloud processing heavily depends on the *3D neighbor search* operation [117, 130], which dominates its execution time [22, 165]. In short, 3D neighbor search is the process of finding points (with coordinates x, y, z) in a point cloud, subject to neighborhood criteria w.r.t. a query point. The neighborhood criteria can be, for example, all points within a distance (also known as the Euclidean Neighbors (EN)), or the k closest points (also known as the K-Nearest Neighbors (kNN)).

However, the safety-critical nature of AD [86, 88] using point clouds - of which *neighbor search* is a critical part - imposes low latency requirements. This is challenging to achieve given the tens of thousands of points found on point cloud frames sensed with LiDARs [152]. For this reason, performing *neighbor search* with an exhaustive search is prohibitive, and instead, a search-friendly data structure is necessary. Among different data structures, k-d trees [26, 51] are widely used in practice as they are efficient for pruning the search space on low-dimensional data such as LiDAR-acquired ones. The k-d tree recursively sub-divides the space, separating the points in one coordinate at a time and creating a leaf node whenever the subspace has less than N points. When searching the k-d tree later, a query can reach the sub-spaces (leaves) where potential neighbors may exist, and verify its distance to them.

While deeper background will be provided later, in Chapter 3, we use this brief explanation so the reader can grasp the contributions of this thesis regarding point cloud processing and neighbor search, targeting better ADS. For this, two main contributions are described following.

Reducing Data Movement

We start by reducing pressure in the memory system while performing point cloud processing. For this, we propose *K-D Bonsai*, a technique to compress point clouds to reduce data movement during point cloud's *neighbor search* execution, improving its performance and energy efficiency. The technique is based on a set of observations that allow compression to take place. We first observe that LiDARs have physical limitations on their range of operation, setting an upper-bound value for the coordinates of the collected points. Second, we observe that k-d trees [26, 51], the typical data structure used for efficient point cloud searches, contain points with similar values in their leaves. As a consequence, the sign and exponent fields (in IEEE 754 Floating-Point (FP) representation [138]) are frequently repeated across leaf points and can be merged. Third, we observe that it is also possible to reduce the size of the mantissa field, and still compute a large percentage of *neighbor search* without losing search precision. More importantly, we show how to cheaply identify any precision loss at run-time, and re-issue full-precision computation to keep baseline accuracy with minimal overheads. In summary, *K-D Bonsai* reduces data movement, improving performance and energy efficiency while guaranteeing baseline accuracy and programmability. Some of the contributions are the following.

- We identify redundancy on bit-fields of FP representation in point cloud data stored in k-d trees.
- We verify that k-d tree neighbor search, a critical operation for point cloud-based algorithms in AVs, tolerates reduction in format representation.
- We derive a mathematical equation to verify whether or not the reduction in format representation could harm the accuracy of the neighbor search operation, which will trigger re-computation with baseline precision if necessary.
- We propose K-D Bonsai, a compression technique to exploit data redundancy and reduction in format representation. K-D Bonsai reduces data movement during *neighbor search*, improving performance and energy efficiency.
- We implement K-D Bonsai through new CPU instructions, namely Bonsai-extensions, demonstrating that our scheme could be easily adopted on next-generation processors for AD. We also validate the proposed scheme using a state-of-the-art and open-source software stack for AD.

These findings can also be found in the following work.

- Pedro H. E. Becker, José-María Arnau, and Antonio González. K-D Bonsai: ISA-Extensions to Compress K-D Trees for Autonomous Driving Tasks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, New York, NY, USA, jun 2023. ACM. ISBN 9798400700958. doi: 10.1145/3579371.3589055. URL https://dl.acm.org/doi/10.1145/3579371.3589055

Maximizing Single Instruction Multiple Data (SIMD) Efficiency

Orthogonal, we exploit ways of increasing computation throughput during the neighbor search, leveraging the Vector Processing Unit (VPU), and making use of this formerly unused CPUs part. For this, we identify that consecutive neighbor searches experience query similarity, which leads to similar k-d tree navigation between consecutive neighbor searches. To exploit it, we propose Caravan: a hardware (Caravan-HW) and software (Caravan-SW) co-design. Caravan packs consecutive (and generally similar) queries to traverse the k-d tree together while reducing computing costs. To leverage query similarity, Caravan first actuates at the software level (Caravan-SW) by gathering multiple queries and employing the VPU to perform several k-d tree searches in parallel. However, the VPU faces vector sparsity when the queries diverge, particularly in the deeper levels of the k-d tree, causing VPU lanes to be underutilized. To amortize this, Caravan actuates at the hardware (Caravan-HW) by means of two new instructions that allow fast operand shuffling to suppress the runtime and hard-to-predict sparsity found in Caravan-SW. In summary, Caravan is able to reduce execution time with a programmable solution, while being minimally intrusive to the CPU. Some of the contributions are the following.

- We identify that consecutive neighbor searches experience query similarity, which leads to similar k-d tree navigation between searches.

- We propose to group multiple queries at the software level to search multiple queries at once, making use of the VPU, and reducing instruction count.

- Despite the benefits of a software-only approach, we observe run-time sparsity in the VPU operands, particularly on the k-d tree leaves, due to query divergence caused by the small differences between queries.

- We design a technique to re-arrange elements in the VPU operands to reduce sparsity and maximize useful operations, reducing execution time further.

- We implement such re-arrangements in hardware with new and minimal architectural features, in the form of two instructions that allow for maximal vector lane utilization during leaf processing.

These findings can also be found in the following works.

- Pedro H. E. Becker, Franyell Silfa, José-María Arnau, and Antonio González. Caravan: A Hardware/Software Co-Design for Efficient SIMD Neighbor Search on Point Clouds. In *Under Review*, 2024
- Pedro H. E. Becker, José-María Arnau, and Antonio González. Boosting Point Cloud Search with a Vector Unit. In *RoboARCH @ MICRO*, 2023

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides an explanation of state-of-the-art AD, including typically employed sensors, algorithms, and hardware. Further, it provides a characterization of multiple algorithms and how they perform on existing architectures, identifying research opportunities. For instance, it identifies the need for better support of point cloud applications that handle data acquired from LiDAR sensors. Chapter 3 further details the importance of LiDAR for AD, also describing how its data is generally employed and processed. Moreover, it discusses related work on hardware support for point cloud processing, pointing out their strengths and weaknesses. Chapters 4 and 5 detail novel techniques to improve point cloud processing. They show how to reduce data movement with K-D Bonsai, and how to achieve high-efficient SIMD processing of point clouds with Caravan. These solutions focus on improving performance and energy with minimal impact on programmability and area. Finally, Chapter 6 presents the conclusions and open questions for future research.

2 Measuring the Problem

“First, we have to understand the problem; we have to see clearly what is required.”

How to solve it (1945)
George Pólya

Before proposing novel solutions to improve the current computing platforms for AD, we need a diagnostic of their main problems. Although some works present an analysis on individual AD algorithms [69, 86, 148], the full picture (i.e., multiple tasks executing concurrently) of AV and their most challenging tasks is necessary, so we put effort where it is needed the most. This chapter aims at obtaining this diagnostic, characterizing a state-of-the-art and open-source software stack for AVs, running in contemporary and high-end hardware. We first deepen the explanation of the software tasks, expanding the quick background given in the Introduction. Following, we measure and discuss the latency of relevant AD modules and the required power from the computing platform to execute them, seeking potential points of improvement from the architectural perspective.

2.1 Autonomous Vehicles Architecture Overview

To carry out this investigation we leverage Autoware.ai [14], a representative open-source project for AVs. Hereinafter interchangeably referred to as Autoware, it was first introduced in 2015 [68], and has been constantly improved by the open-source community, with contributions from industry and academia [13]. In 2023, its main maintainer company, Tier IV [147], got certified in level 4 AD [146] in Japan, employing Autoware software. Although Autoware is not the only open-source project available, it has the largest AD community

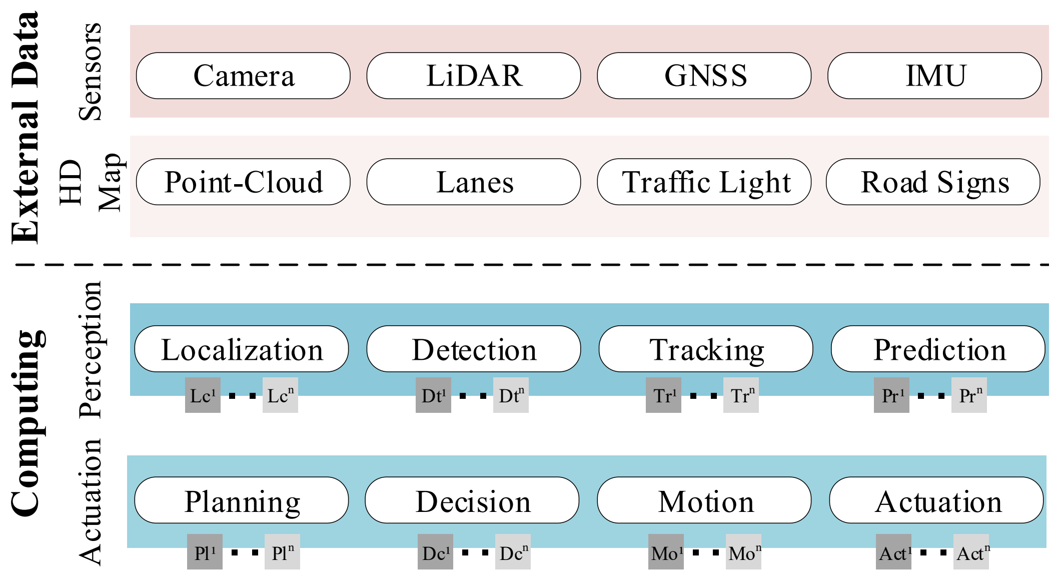


Figure 2.1 A high-level description of the Autoware architecture. Sensor data and pre-existing high-definition maps feed the software stack.

on GitHub, and it does not impose limitations as other projects in the same tier of maturity and complexity. For instance, Baidu's Apollo project [4], which is also open-source, has some vital parts of the code, such as DNN models for visual detection, released as black-box libraries [4]. On the contrary, Autoware software is open-source, and all the used libraries or third-party applications are also open-source. Thus, we can always inspect some of its parts with more detail, without concern that we might end up with proprietary code, which sometimes can lead to research dead ends.

At a high level, Autoware can be divided into a set of layers that interact and collaborate to perform self-driving, as we depict in Figure 2.1. The next subsections detail the modules that compose each of these layers.

2.1.1 External Data

Capturing the real-world is the first concern for any AD solution. Following, we explain the external data that feeds Autoware's computing system, such as sensor data and High-Definition (HD) maps.

Sensing

A mandatory layer in AVs platforms is composed of the vehicle sensors [88], whose data will be further processed to characterize the driving scene. Although different companies use different sets of sensors, a list of common sensors includes i) one or more *cameras*, which capture images for object detection; ii) *LiDAR*, that measures the distance from the car to surrounding objects through laser scan. LiDAR capture point clouds, which are used for fine-grain localization, segmentation, and object detection; iii) *RADAR*, for better robustness across weather conditions compared to LiDAR, but with lower precision; iv) *GNSS*, to provide an approximate initial position to the localization algorithms; v) *IMU*, which provides data such as linear velocity, useful to refine estimations obtained with other sensors.

Although Autoware has modules to interface with all previously mentioned sensors, only LiDAR appears in Autoware.ai's minimum requirements [11]. This is due to the versatility of LiDAR generated point clouds, that can be used for multiple scene recognition algorithms, as we explain soon. Still, adding more sensors enlarges the number of scenarios Autoware can cope with. For instance, cameras are required to recognize traffic lights.

HD Map

Relying on GNSS and commodity map applications (e.g. Google Maps) is not enough for AVs [88]. Instead, they require HD Maps [133, 134] to localize themselves with centimeter-level precision. The creation of an HD map is often performed by specialized companies. They contain a static point cloud of the complete region where the vehicle can drive. The point cloud is combined with cartographic and traffic details such as the disposition of the lanes, the allowed ways to drive, speed limits, 3D position of traffic lights, road signs, and zebra crossings. Later on, this information helps the ADS to improve its surroundings' perception and actuation. In time, we note that the point cloud inside of an HD map is static and collected *a priori*. It differs from the dynamic point clouds collected by the vehicle's LiDAR, although both are necessary for vehicle localization.

2.1.2 Computing

The Autoware software stack is built upon the Robot Operating System (ROS) [120]. ROS is a middleware collection of libraries, tools, and underlying infrastructure that increases productivity on robot systems development. Typically, a ROS project is divided into multiple software modules, named *nodes*, whose purpose is to individually solve a task (e.g., detect an object) while globally collaborating towards a major goal (e.g., successfully self-drive a vehicle). The *nodes* communicate through a publish-subscribe arrangement: *nodes* publish their outputs into a shared memory space, referred to as a *topic* in ROS jargon, to which other *nodes* subscribe, as we depict in Figure 2.2. When new messages are published, all *subscriber*

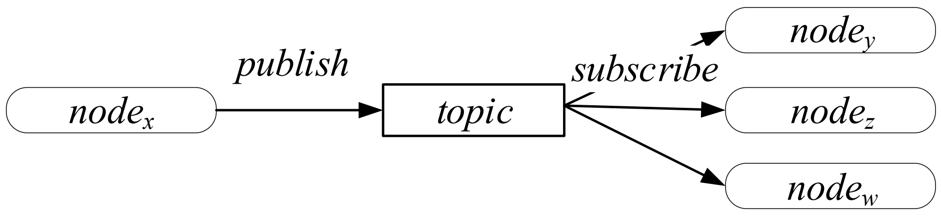


Figure 2.2 The publisher-subscriber scheme for *node* communication in ROS. A *node* publishes its output into a memory space, called *topic*. Multiple *nodes* can subscribe to a *topic*, being notified by ROS when new messages (data) are published on it.

nodes are notified, being able to read and process the data. This simple arrangement allows *nodes* to consume inputs, process them, and assign the output data for upcoming *nodes* to carry further processing.

Additionally, each ROS *topic* specifies an interface for message publication. Therefore, different *nodes* implementations can subscribe or publish to a given topic, as long as the interface is respected. This allows *nodes* to be easily substituted by different ones speeding up the testing of different implementations without major changes in the software structure. For example, Autoware has multiple *nodes* that perform image-based object detection. Some are faster, and lighter in memory consumption, some are slower but more accurate. Users can select the best-suited one from a pool of image detector *nodes*, according to their needs.

Following, we review the most relevant *nodes* from the Autoware software stack. Readers can find a summary in Table 2.1.

Perception

Prior to performing driving actions, self-driving vehicles must first understand the scene they are in. This is done through *perception* algorithms. As we depict in Figure 2.1, the perception layer contains different modules, as we explain as follows.

- *Localization* is one of the major tasks AV perform. In Autoware, the localization starts with a GNSS estimation, and then it is refined via LiDAR data processing. For this, Autoware first down-samples the input point cloud through a node called *voxel_grid_filter*[1] [117], reducing the number of points to speed up further processing. The filtered point cloud is then processed by the *ndt_matching* [28, 93] node. This node tries to maximize the overlap between the sensed (LiDAR) and the reference (HD map) point clouds, in a process also known as registration. Employing Newton's optimization method, the algorithm searches for a transformation matrix that translates and rotates the sensed point cloud so that it optimally aligns with the reference point cloud. The obtained transformation tells where the origin of the sensed data (i.e., the LiDAR sensor) is with respect to the origin of the reference frame (i.e., the origin of the HD map). This relative pose pinpoints the vehicle in the map, typically localizing it with centimeter-level precision. To speed up the optimization, the GNSS is used for an initial guess of the pose. Additionally, the IMU may be used to anticipate where the subsequent positions are likely to be.

- *Detection* must identify objects in the environment, classify them (e.g., a car, a pedestrian), and find their relative pose to the vehicle. In Autoware, object detection can be performed with LiDAR and camera sensors. For LiDAR, the point cloud is first filtered by the *ray_ground_filter* which separates the source point cloud into two groups: the ground points (which we can drive over) and the non-ground points (which we should avoid hitting). Afterward, it provides the non-ground points to the *euclidean_cluster* [129] node, which performs point cloud segmentation, clustering points near each other to perceive them as a single entity, extracting their shape. For that, it repeatedly searches for neighbor points in the sensed point cloud, applying transitivity to cluster them [129, 170]. The algorithm also calculates the clusters' centroids to stipulate how distant the objects are from the vehicle. For camera-based object detection, Autoware provides support for SSD [90] and YOLO [122] DNNs. They rely on trained data to find and label relevant objects captured by the camera. They output the type of each found object and the coordinates where they appear in the image. After detecting and classifying objects in the images, the output is fused with the LiDAR findings. This task is carried by the *range_vision_fusion* node, which uses camera and LiDAR calibration data to fuse coordinates into the same space domain. The fusion provides several benefits to detection. On the one hand, LiDAR obtained data adds 3D features to the image-based detection, giving a sense of volume to the object and also localizing them in the map. On the other hand, image detection adds semantics to the objects, labeling the type of a given entity, e.g., a car, a truck, or a pedestrian.

- *Tracking* aims at keeping a consistent identification of different objects across consecutive frames, to determine how they are moving. In Autoware, the *tracking* task requires the list of detected objects from the *Detection* task, after *fusion*. The *tracking* must cope with several challenges since detected objects may experience occlusion,

[1]There are other down-samplers available. We do not try out an exhaustive combination of all possible implementations, but rather, use the recommended algorithms accordingly with Autoware documentation [10]. This also holds for other tasks, whenever there is a consensus on a given solution. When the best algorithm/implementation is not clear or incurs considerable trade-offs (e.g. object detection DNNs) we experiment with the different possibilities.

dissimilar motion patterns, and clutter [143]. To cope with these problems, Autoware implements the *imm_ukf_pda_tracker* node, which is inspired in previous works [121, 132] that combines different filters algorithms to overcome the challenges (e.g., the Unscented Kalman Filter (UKF) for non-linear estimation [155]). The *tracking* then publishes the list of tracked objects with information such as position, velocity, and associated identification.

- *Prediction* takes place after *tracking* updates the status of the objects. This module utilizes each tracked object's position, velocity, and direction to stipulate the path they are likely to follow in the near future. In the version used in this work (1.12), Autoware.ai only considered the objects to have constant velocity (both when driving straight and when turning), hence the prediction *node* name *naive_motion_predict*. In any case, the velocity is updated on each frame (at a rate of 10Frames per Second (FPS)), making the approximation less harmful. Finally, the tracked objects associated with predicted paths, and the LiDAR ground points, feed the *costmap_generator_obj* node. This *node* determines the drivable areas for the vehicle, i.e., not occupied by objects or to be occupied in the near future, based on the trajectory predictions. This is key for finding the possible trajectories the AV can take.

Actuation

Once the vehicle has a broad comprehension of the current traffic scenario and its participants, it can plan and execute driving actions through the following components:

- *Planning* determines the trajectory for the vehicle. In Autoware, it is divided into two parts: global and local planning [45], respectively implemented by *op_global_planner* and *op_local_planner* nodes [44]. The global planner defines a high-level route to reach the target destination. Meanwhile, the local planner details how the route will be followed depending on the perception outcome. For instance, the local planner decides whether the vehicle should stop for a pedestrian, accelerate in a free lane, or change lanes to avoid an obstacle while considering the high-level route defined by the global planner.
- *Motion* modules generate the control output to maneuver the vehicle in order to follow the planned path. Autoware implements the pure pursuit algorithm [136] (*pure_pursuit node*) to calculate the linear and angular velocity the vehicle should perform. These velocity values are submitted to a low-pass filter (*twist_filter node*),

Table 2.1 Summary of important Autoware nodes.

Node	Description
voxel_grid_filter	Downsample an input point-cloud, reducing the amount of points to simplify further computations.
ndt_matching	Localize the vehicle by matching LiDAR acquired point-cloud with a region of the HD map point-cloud.
euclidean_cluster	Cluster LiDAR acquired points nearby each other, segmenting it into volumes that can be perceived as objects.
YOLO / SSD	DNN-based nodes used to detect and classify objects (e.g., vehicles, pedestrians) from images.
range_vision_fusion	Combine LiDAR and image-based detected objects into the same coordinates, improving objects perception.
ray_ground_filter	Separate an input point-cloud in two: points that compose the ground, and points above the ground level.
imm_ukf_pda_tracker	Track objects by assigning them an identification and keeping it coherent among subsequent frames.
naive_motion_prediction	Extrapolate the current trajectory of different objects to predict where they will be in the future.
costmap_generator	Determine drivable areas in the map, i.e., with no objects at the time or predicted to be in the near feature.
op_planner	Global and local path planning based on the current scene and target location.
pure_pursuit	Calculate the necessary motion (linear and angular acceleration and velocity) to follow the desired path.
twist_filter	A low-pass filter applied over motion control to smooth the vehicle driving.

used to smooth the driving actions. Finally, Autoware interfaces with the vehicle through a drive-by-wire system, sending the control to perform the driving actuation.

2.2 Characterization Methodology

In this section, we present the methodology for the Autoware characterization. We detail tools, configurations, and steps to acquire the data.

2.2.1 Autoware Execution Environment

Prior to characterizing Autoware, we need to prepare its execution environment. While it would be captivating to set up Autoware in an actual vehicle, doing so is costly, arguably dangerous, and, more important to our analysis, hinders the experiments' reproducibility. Instead, we evaluate it in a controlled, simulated setting. For that, we rely on real-life data collected beforehand with sensors during a vehicle drive, and a set of ROS tools to provide trustworthy data input to the Autoware software stack. Therefore, pre-recorded data is input to Autoware, which runs on a high-end computing platform while also being profiled. Figure 2.3 illustrates the arrangement.

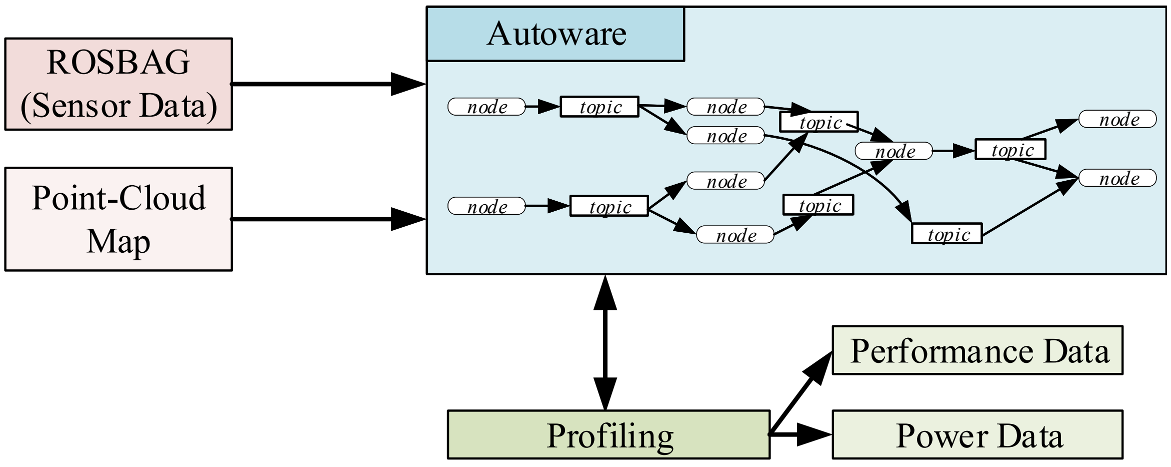


Figure 2.3 The experimental setup. Preexisting files for sensor data and point-cloud map provide real-life inputs. Profiling tools collect characterization data.

The external data (Section 2.1.1) used in our experiments was collected from an 8-minute drive, recorded in the city of Nagoya in Japan, obtained from the Autoware Data [67] repository. The data was released in the form of a *.bag* file, a typical file type used by the ROS community for data logging, allowing users to record datasets, visualize, label, and store them for future use [126]. A *.bag* file may contain multiple *topics*, each with its data and headers. For instance, it has consecutive point cloud frames captured by the LiDAR installed in the vehicle, along with their timestamp with a rate of 10 FPS. Likewise, it also has camera frames with a rate of 15 FPS.

We can use the *rosbag* tool [127], released with ROS, to replay the *.bag* file, publishing messages in the *topics* specified in the file into the ROS subscribe-publish mechanism. When running Autoware within the same ROS environment, it will capture these messages, by subscribing its *nodes* to the *topics* of interest. This means that we can use sensor data as input as if Autoware was receiving the data from the sensors from a vehicle in real time. More importantly, we can re-run the same *.bag* file to stimulate Autoware as many times as needed. Hence, we can experiment with Autoware with multiple combinations of nodes, metrics, and profiling tools, always using the same input, improving reproducibility, and easing further data analysis.

To completely stimulate the Autoware stack we also need an HD map, particularly to perform *localization*. However, the *.bag* does not come with an HD map, nor could we find one for that specific region (HD maps are often proprietary and crafted for small parts of the globe). To overcome this, we use an Autoware utility (*ndt_mapping*), which creates a point cloud map based on the LiDAR data from the *.bag* file. This, however, only generates a point cloud map, useful to stimulate the localization nodes, but does not contain the whole HD map with annotations (speed limit, traffic light poles, etc.). We discuss the limitations of this process later on.

The specific hardware and software are detailed in Table 2.2. We note that most of the software versions presented in the Table are required by Autoware, for compatibility reasons. As for the different knobs to configure the nodes, we use the predetermined configuration released with Autoware. The image-based nodes that perform object detection (which we refer to as *vision_detection*) run in the GPU as set in Autoware's default configuration. We also followed the Autoware guidelines to download the pre-trained weights for the *vision_detection*. We experiment with SSD [90] (SSD 300 and SSD 512, with data from [89]) and YOLO [122] (YOLO v3-416 model and pre-trained weights from [123]) models. We also set the *euclidean_cluster* to run in the available GPU changing the default configuration to run it on the CPU. Execution of it in CPU and GPU was found to be similar regarding execution time by previous works [69], so here we take the chance to see how the GPU behaves when it is used by multiple algorithms (*euclidean_cluster* and *vision_detection*). All other evaluated algorithms are single-thread and run on the CPU, again following the default configuration of Autoware.ai.

Table 2.2 Environment hardware and software platforms.

		CPU	GPU
Hardware	Model	Intel i7-7700K	NVIDIA GeForce GTX 1080
	Architecture	Kaby Lake	Pascal
	Frequency	4.2 GHz	1.6 GHz
	# Cores	4 (8 Threads)	2560
	L1 Cache (I/D)	32 KB (8-Way)	-
	L2 Cache (Unified)	256 KB (4-Way)	4 MB
	L3 Cache (Unified)	8 MB (16-Way)	-
	Main Memory	64 GB DDR4	8 GB GDDR5X
Software	Operating System	Ubuntu 16.04 x86_64	
	Kernel	4.15.0-96-generic	
	ROS	Kinetic	
	Autoware	Autoware.ai 1.12	
	CUDA	9.0	

2.2.2 Characterization Procedures

Latency Measurement. We leverage the C++ *chrono* library to instrument Autoware's code and measure two latency values: *single node latency*, and perception *end-to-end latency*, as illustrated in Figure 2.4. *Single node latency* measures the time a node takes to process the input. Measurement starts immediately after the input is read from the source *topic* until immediately before the output is written to the destination *topic*. This measurement aims to identify latency-hungry nodes, which are potential bottlenecks for AD.

We also assess *end-to-end latency* for perception, encompassing the latency for a chain of multiple nodes to process an input frame, which we call computation paths. Each computation path corresponds to a sequence of nodes that process one after another, transforming the sensor inputs into a precise perception of the surroundings. Therefore the *end-to-end latency* includes the latency of multiple *single nodes* plus the communication costs of the ROS subscribe-publish mechanism. Figure 2.4 depicts this situation. The sequence of nodes in different computation paths used by Autoware can be found in Table 2.3, together with the discussion of results.

Differently from previous works (e.g., [69, 86]), we analyze a broader number of computation paths, hence enlarging the identification of performance bottlenecks. Also, while they simply sum the individual latency of the nodes to measure end-to-end latency, our measurements encompass contention and communication overheads since we profile Autoware with multiple ROS nodes running in parallel. We track the messages since the sensors' data is given to the first Autoware node. We use message timestamps to track

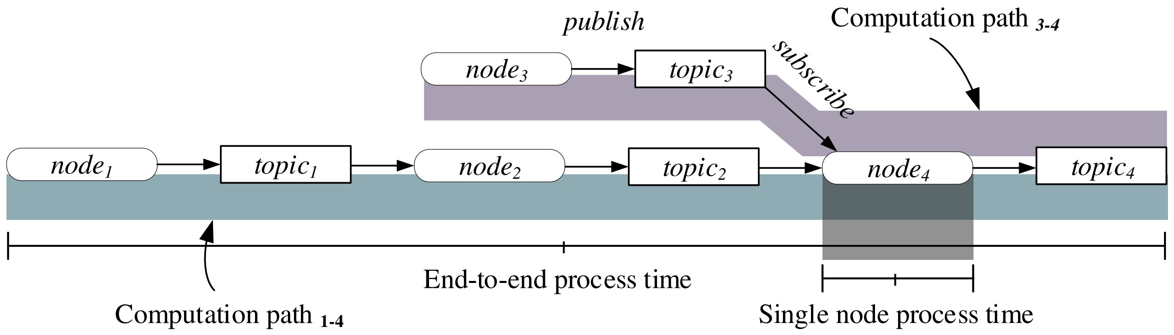


Figure 2.4 The latency measurements accounted for in our experimentation. Single node process time corresponds to the processing inside a node, excluding communication overheads. End-to-end process time measures the latency since a frame (e.g., camera image, LiDAR point-cloud) enters the system until its final contribution to the AVs' perception.

the end-to-end latency. The timestamp stays untouched in the messages' headers and it is passed along from input messages to output messages, serving as an identifier tag that can be used to backtrack their end-to-end processing time.

Power Consumption Complementary to the latency assessment we also gather overall information regarding CPU and GPU power consumption. The goal is to quantify the associated power and energy costs used by each of these two platforms. This could help us understand when each of them is more suitable, depending on the performance and power consumption trade-off. We use Intel's Running Average Power Limit (RAPL) [46, 160] to measure power consumption in the CPU, and the *nvidia-smi* [110] tool to measure power in the GPU.

2.2.3 Methodology Limitations

As aforementioned, the lack of an HD map with rich annotation imposes a few limitations. Since we do not have the annotation for traffic light poles' position, we cannot perform traffic light detection algorithms. Also, since there is no specification of roads, allowed lanes, and speed limits, we cannot stimulate control and motion algorithms, namely *op_planning, pure_pursuit, and twist_filter.* It is possible to use game-engine-based simulators (e.g., the LGSVL [81] or CARLA [47] simulators), which provide this information in a rendered traffic environment. However, such simulators demand high CPU and GPU utilization. Given the considerable overheads introduced to the system, these simulators are more suited to verify functionalities instead of profiling. Finally, the profiling focuses on perception nodes, as they represent the vast majority of the execution time [69, 86].

2.3 Characterization Analysis

2.3.1 Latency Characterization

Single node latency. Figure 2.5 presents the latency distribution for Autoware nodes. Three sub-figures display results according to the use of different image detection algorithms: (a) SSD 512, (b) SSD 300, and (c) YOLO v3. Each position in the x-axis corresponds to an individual node from Autoware (described in Table 2.1). A box plot shows the distribution of the latency values for each node in the y-axis. The distributions comprise multiple individual measurements for processing the input messages, one at a time. In a box plot [149], each box (colored rectangle) contains 50% of the values, from the first quartile (25%) to the third quartile (75%). A horizontal line marks the second quartile (50%), i.e., the median value of the distribution. Whisker lines indicate how the data spreads, marking maximum and minimum measured values up to a limit of $1.5\times$ the height of the box. Values beyond these limits are shown separately, as outliers.

From the figure, when a big DNN model such as the SSD 512 (Figure 2.5a) is used, the image-based object detection is the most time-consuming node, taking around 80 ms to execute. If we consider 100 ms as the target reaction time, as in previous works [39, 86], we are devoting 80% of it to finding the objects in the camera images. This is a high number considering that other tasks need to execute after that. When we consider other models such as SSD 300 (Figure 2.5b) and YOLO v3 (Figure 2.5c), inference time roughly drops by half, being under 40 ms, making them a better choice performance-wise. However, although latency plays a major role in choosing the image detector for AVs, we acknowledge that assessing the most propitious image detector is out of the scope of this work since other metrics that are not evaluated here (e.g. accuracy) also need to be taken into account.

While Autoware appears to have a single bottleneck regarding camera-based processing, it has a handful of LiDAR-based algorithms with significant latency. Not rarely nodes such as *ndt_matching*, *euclidean_cluster*, *ray_ground_filter*, and *costmap_generator_obj* (all LiDAR dependent) take 20-40 ms to execute. This is even more concerning since LiDAR processing generally relies on a chain of nodes, as we explained in Section 2.1.2, thus adding up individual computation times as we discuss in the upcoming subsection. Although needed, to improve them might be difficult. Since these nodes perform different tasks, it is challenging to find a one-fits-all manner of accelerating them.

In time, the range of values is another relevant observation from Figure 2.5. We can see a considerable distance between the maximum and the minimum values in the reported latency values. Particularly, we see outliers appearing in the upper parts of the

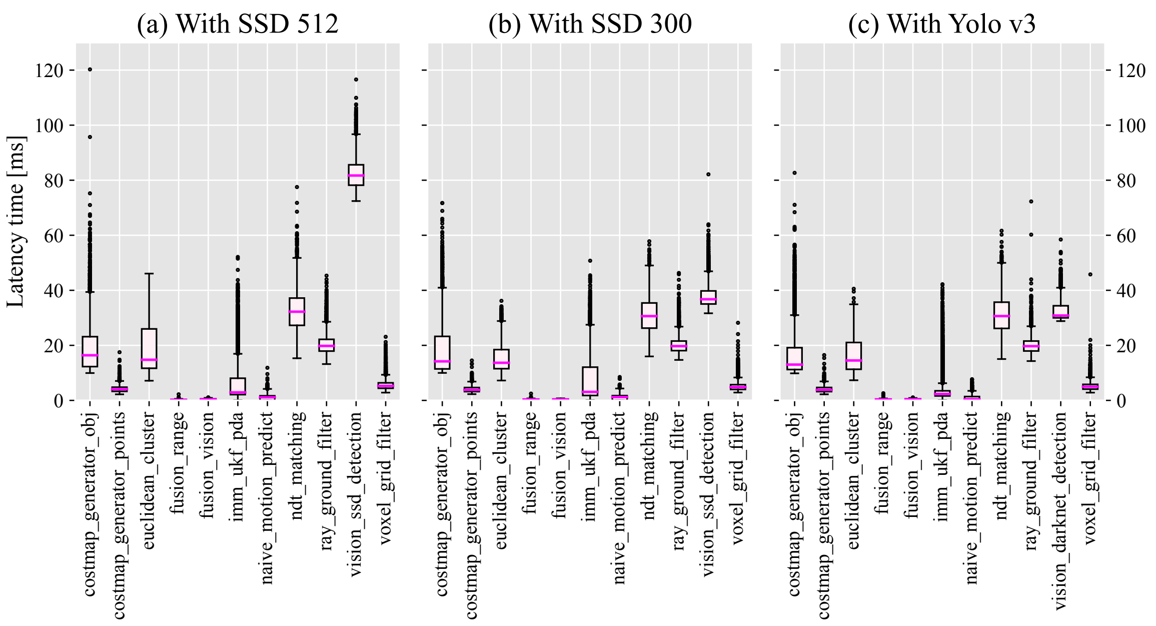


Figure 2.5 Autoware's single node latency for different nodes. Experimentation includes different image detector nodes.

box plots, indicating that some frames take considerably more time to execute than they normally do. These higher measurements belong to the *tail latency* and occur in special-case frames. For example, nodes with many outliers such as *costmap_generator_obj*and *imm_ukf_pda_tracker* highly depend on the number of objects in the scene. This is expected since the more driving players, the higher the time to track each of them (*imm_ukf_pda_tracker*), and project their occupancy in the world (*costmap_generator_obj*). Therefore, when the AV is in a crowded street, the collected frames (e.g., point cloud) will have more data, and their execution time will rise beyond the usual. These outliers cannot be neglected since they can cause the system to miss a time deadline for a frame, hurting reaction time and introducing a safety flaw. Thus, when considering tail latency, the performance improvement concerns become even more evident.

Figure 2.5 also shows another interesting finding: the tail latency of some relevant nodes is highly affected by the other components running in parallel. For example, the *euclidean_cluster* has a tail latency of 36 ms when running alongside the SSD 300, but it increases to 46 ms when using the SSD 512. This increase of 28% in the tail latency can happen since both algorithms compete for resources (e.g., the GPU). The bigger the DNN model used for *vision_detection*, the higher the GPU usage, and the less resource available for the *euclidean_cluster.* Other nodes also see a fluctuation in their tail latency depending on the image detection module employed.

End-to-end latency. The AD task depends on the collaboration of multiple nodes, as introduced in Section 2.1. We now examine different computation paths that Autoware relies on to perceive the environment. We verify the end-to-end time from sensor input until final localization and road occupation. Since road occupation (*costmap_generator_obj*) can be updated by three different computation paths, we measure each of them individually. For example, both camera and LiDAR findings can update the occupation. The set of nodes and topics that compose each computation path are described in Table 2.3. The latency of these four computation paths is presented in Figure 2.6.

Table 2.3 Computation paths description.

Computation Path	**Path description (/topic → node)**
localization	/points_raw → voxel_grid_filter → /filtered_points → ndt_matching
costmap_points	/points_raw → ray_ground_filter → /points_no_ground → costmap_generator
costmap_vision_obj_*	/image_raw → vision_ssd_detection → /detection/image_detector/objects → range_fusion_01 → /detection/fusion_tools/objects → imm_ukf_pda_01 → /detection/object_tracker/objects → ukf_track_relay → /detection/objects → naive_motion_predict → /prediction/motion_predictor/objects → costmap_generator
costmap_cluster_obj	/points_raw → ray_ground_filter → /points_no_ground → lidar_euclidian_cluster_detect → /detection/lidar_detector/objects → range_fusion_01 → /detection/fusion_tools/objects → imm_ukf_pda_01 → /detection/object_tracker/objects → ukf_track_relay → /detection/objects → naive_motion_predict → /prediction/motion_predictor/objects → costmap_generator

As depicted in the figure, some computation paths can take hundreds of milliseconds. Further, regardless of the chosen image detector algorithm, the worst case among all computation paths always surpasses 200 ms to compute when considering tail latency. Thereby, there are moments during the AV driving where the system takes more than twice the time commonly stipulated to drive safely (100 ms). This is an interesting finding considering our measurements are over a mature self-driving project. Notwithstanding, we highlight our computing platform (see Table 2.2) is a high-end computer. Such a big computing arrangement may not even be an option for AVs due to size and energy consumption constraints. We can use our data and extrapolate that smaller and less aggressive hardware would yield even worse results, being impractical in reality, and reinforcing the need for improving computing platforms so they meet the AD constraints.

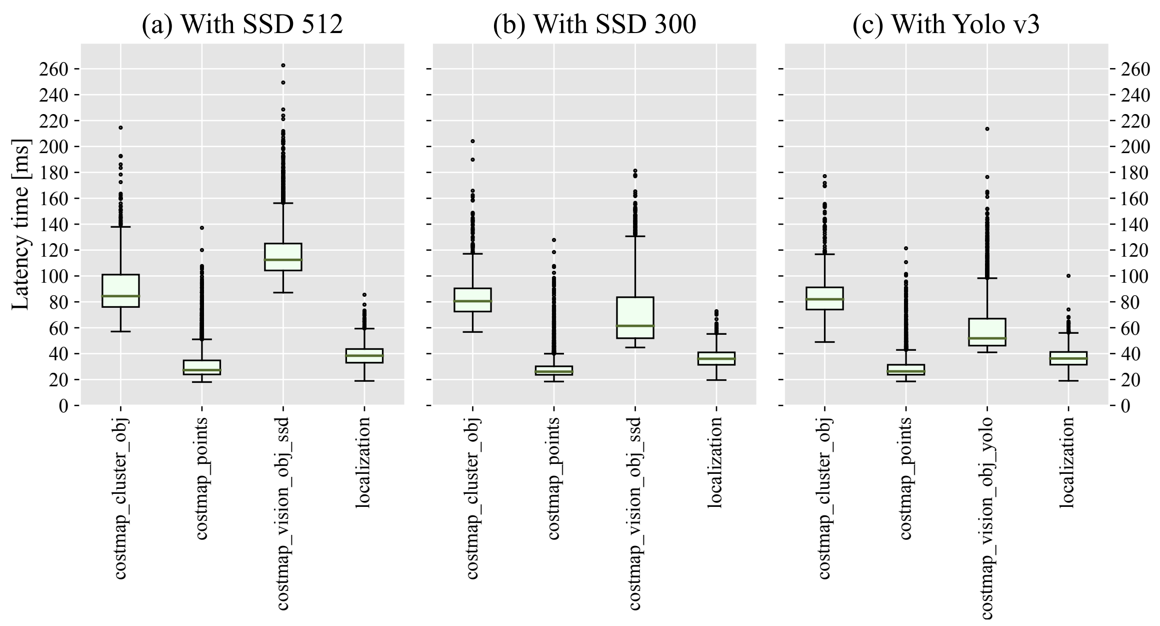


Figure 2.6 Autoware's end-to-end perception latency. Experimentation considers the usage of different image detector nodes.

The most demanding computation paths are those that extract semantics, with image object detection or point cloud clustering followed by tracking and prediction. If SSD 512 is used (Figure 2.6a), then the path with the *vision_detection*, namely *costmap_vision_obj_ssd*, holds the worst average latency. When faster vision detection algorithms are used, e.g., SSD 300 (Figure 2.6b) or YOLO (Figure 2.6c), the worst average latency relies on the computing path with the *euclidean_cluster*, namely *costmap_cluster_obj*. The computation path for localization appears to be the quickest one. Part of this is because it only encompasses two nodes (*voxel_grid_filter* and *ndt_matching*). Still, end-to-end latency sometimes often reaches 60 - 80 ms. Since our dataset is far from stressing all possible corner cases for localization, we cannot discard optimizations in this path. Revisiting the single node latency analysis and combining it with the end-to-end latency analysis, we conclude that *vision_detection*, *euclidean_cluster*, *ray_ground_filter*, and *ndt_matching* are the most important nodes where optimization efforts should focus. These nodes are time-consuming, and on belong to different critical paths of end-to-end latency.

2.3.2 Power Characterization

Following the latency analysis, we briefly present power consumption data during the execution of Autoware. Measurements are reported in Table 2.4. First of all, we see that GPUs consume considerably more power than the CPU, even though they were configured

Table 2.4 CPU and GPU mean power dissipation.

	CPU	**GPU**	**Total**
With SSD512	44.90 W	122.14 W	167.05 W
With SSD300	42.63 W	67.08 W	109.71 W
With YOLO	42.35 W	116.73 W	159.08 W

to run only two nodes (*vision_detection* and *euclidean_cluster*) while the CPUs executed more than ten. When using the SSD 512 model, the power consumption gap between CPU and GPU is $2.7\times$, with the GPU accounting for more than 70% of the total power consumption, which is a high amount. Using simpler models, such as the SSD 300, can decrease GPU power consumption by 45%, greatly reducing its gap to the CPU counterpart to a lower, but still concerning $1.57\times$. Therefore, finding simpler but effective models should be considered when designing novel DNNs for object detection. At the same time, the high power consumption of GPUs also explains the increasing interest in energy-efficient alternatives for image-based DNNs, with accelerators such as Google's Tensor Processing Unit (TPU) [70], and others as we discussed in the Introduction Chapter.

Finally, this energy gap between CPUs and GPUs should also guide the decision on where to execute a given node. For example, related work [69] has found *euclidean_cluster* to perform similarly (performance-wise) in the CPU and the GPU. In these cases, and acknowledging the difference in power consumption between the two platforms, running it in the CPU to achieve better energy efficiency is most suitable. GPUs, otherwise, should be employed only if they consistently outperform CPU latency times, as for *vision_detection* algorithms.

2.4 Chapter Takeaways

AVs are a timely problem. Because they are incipient, however, they lack a clear list of improvement points. To fill this gap we characterized a state-of-the-art AD solution, Autoware.ai [13], and described it in this Chapter. By inspecting its software architecture, i.e., the employed algorithms and how they collaborate, together with latency and power data, we could take hints for research directions.

First of all, we found that the high end-to-end latency of Autoware does not meet latency constraints, confirming the need for software and hardware improvements. Upon that, multiple individual nodes take considerable time to compute and are combined in different computation paths. Moreover, their objectives and computations are varied. Therefore there is no silver bullet for ADS. The most time-consuming algorithms should be inspected individually. At the same time, we saw that using a GPU is very power-consuming, although necessary to guarantee the performance of some nodes such as *vision_detection*. For algorithms that do not considerably benefit from it like the *euclidean_cluster* [69], using CPUs is a better choice for energy efficiency reasons. Targeting hardware improvements, and acknowledging the drawbacks of specialization (see Section 1.2.2), we can try to find commonalities between different nodes, to encounter generic solutions. Particularly, the dominance of CPU usage and LiDAR point cloud processing in nodes' execution suggests we go deeper into that research line. Accordingly, we narrow our study into improving CPU support for point cloud processing.

3

Point Cloud Processing for Autonomous Vehicles

“By emitting invisible lasers at incredibly fast speeds, LiDAR sensors can paint a detailed 3D picture from the signals that bounce back instantaneously. These signals create ’point clouds’ that represent a three-dimensional view of the environment, allowing LiDAR sensors to provide the visibility, redundancy and diversity that contribute to safe automated and autonomous driving.”

A Sense of Responsibility: Lidar Sensor Makers Build on NVIDIA DRIVE [107]
NVIDIA Blog (2021)

As discussed in previous Chapters, point clouds collected by LiDAR are fundamental for AVs. They are used for several AD algorithms and take considerable resources and execution time. At the same time, their use in safety-critical applications is considerably under-investigated compared to other sensors such as cameras. Due to their importance, the remaining of this thesis narrows the scope into LiDAR-based tasks for AV. Following, we provide background on LiDAR functioning and point cloud generation, introduce the key *neighbor search* operation for point clouds, and then we present state-of-the-art related works that also try to improve LiDAR point clouds processing.

3.1 Background

3.1.1 LiDAR and Point Clouds

Light Imaging Detection and Ranging (LiDAR) sensors are active sensors based on light emission. They beam laser lights and capture their reflection back in the sensor to create point clouds of the environment. Based on the direction of each beamed laser and the elapsed time for it to return, the sensor calculates the 3D position (x, y, z) of each reflected point. Optionally other attributes such as intensity and color can also be collected. LiDAR sensors [25] are commonly employed for collecting point clouds in real-time given their accuracy, relatively small size, and sampling rate [85, 153], being a critical technology for AVs to see the world they are navigating [4, 68, 85, 169].

Among different variants of LiDAR, those based on *Mechanical Spinning* are the most popular ones [85]. Figure 3.1 exemplifies a *Mechanical Spinning* LiDAR produced by the company Velodyne [154]. These sensors are composed of a set of laser emitters and receivers that beam and capture laser pulses at different angles. The emitters and receivers are placed on top of a spinning base, which is rotated with a motor, allowing the LiDAR to scan the complete surroundings. Figure 3.2 depicts how these sensors are typically installed in a vehicle.

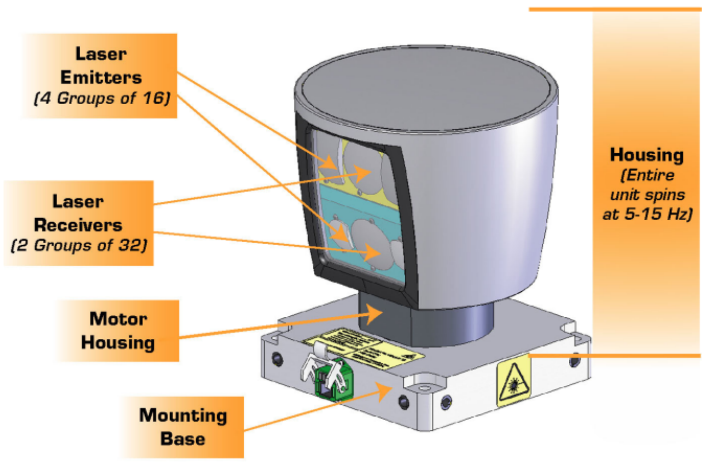


Figure 3.1 The Velodyne HDL-64E is an example of LiDAR sensor used for AD. It has a 360° horizontal FoV, and a 26.8° vertical FoV. Adapted from [52].

Figure 3.3 shows how the horizontal and vertical FoV are covered. At fixed angular steps, the sensor shoots a column of laser beams, thus collecting a column of points, repeating the process for the next angular step. After a complete revolution, the sensor has a point cloud frame ready to be sent to the computing platform of the AV, such as the one depicted

Figure 3.2 A LiDAR mounted on a vehicle. The sensor is typically placed on top of the vehicle to avoid occlusion and exploit its full range of operation. Adapted from [154].

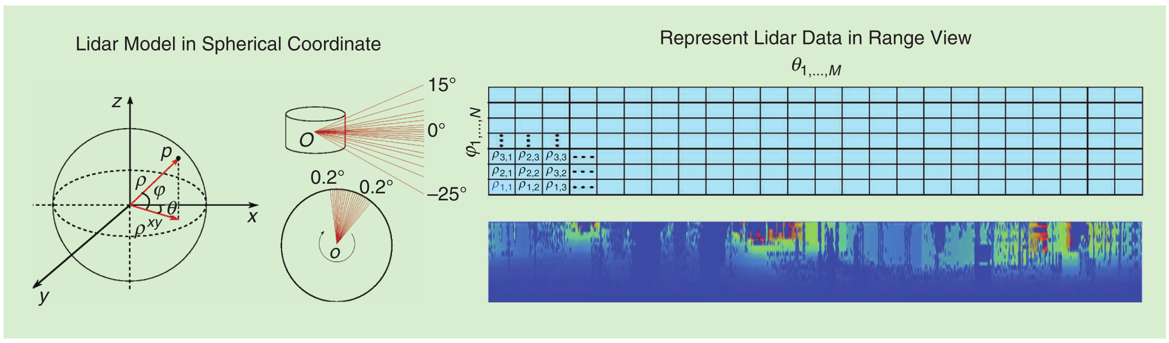

Lidar Model in Spherical Coordinate
Represent Lidar Data in Range View
$\theta_{1,\dots,M}$
$\varphi_{1,\dots,N}$
z
p
ρ
φ
θ
ρxy
x
y
O
15°
0°
−25°
0.2°
0.2°
o
$p_{3,1}$ $p_{2,3}$ $p_{3,3}$
$p_{2,1}$ $p_{2,2}$ $p_{3,2}$
$p_{1,1}$ $p_{1,2}$ $p_{1,3}$

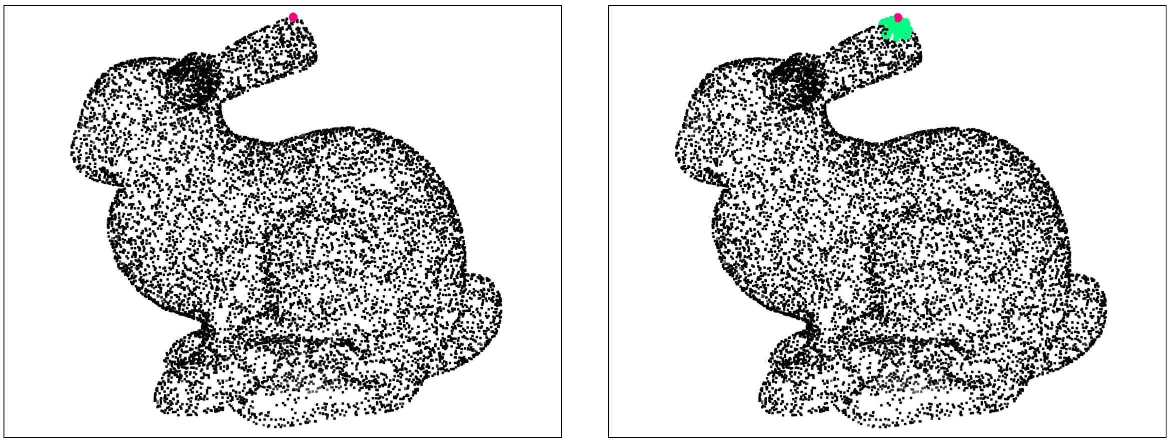
Figure 3.3 The range view of the Velodyne Ultra Puck. At each angle θ (horizontal FoV), multiple laser beams cover an aperture φ (vertical FoV) capturing the distance ρ to a reflective surface. The collected point cloud can be seen as a range image (right). Reproduction from [85].

in Figure 1.3. Depending on the resolution, these point clouds can have up to hundreds of thousands of points [152].

Therefore, each point cloud framed sensed by the LiDAR contains a collection of points in the 3D space with coordinates (x, y, z), with the origin at the LiDAR sensor. With proper calibration, these coordinates can be translated with respect to some other origin (e.g., of a camera, or the center of the vehicle) so that multiple sensors collaborate in the same coordinate system. Therefore, each point in the 3D set belongs to a surface of the real world, providing a discrete (yet detailed) representation of it, that can be used for AD perception.

3.1.2 Searching Neighbors in Point Clouds

After the LiDAR collects a point cloud frame, the CPU takes place to process it, as we explained in Section 2.1. Now, although different AD algorithms have different high-level



(a) The *query* point submitted for neighbor search, highlighted in magenta.

(b) The found neighbor points, highlighted in green.

Figure 3.4 A visual example of the *neighbor search* operation on a point cloud.

goals, they sometimes share common processing building blocks. We inspected nodes in Autoware with tools such as *valgrind* [104] and Performance Application Programming Interface (PAPI) [162] to search for such important building blocks. We found that point cloud processing heavily depends on the *neighbor search* operation. In Autoware.ai, for example, *neighbor search* accounts for more than half of the execution time of segmentation and localization, as we reported in Figure 1.5. Out of the Autoware scope, *neighbor search* is used for other point cloud tasks such as normal estimation and 3D DNNs.

The *neighbor search* operation obtains the neighbor points of a *requested* coordinate in the point cloud, also known as the *query* point $q = (x_q, y_q, z_q)$. Figure 3.4 illustrates a query point (Figure 3.4a, in magenta), and a set of neighbor points (Figure 3.4b, in green). The user specifies a criteria to define which points are *neighbors*, and which are not. Typical implementations (see next subsection) include support for Euclidean Neighbors (EN) or its K-Nearest Neighbors (kNN). The EN considers any point within a distance r to be a neighbor of the *query* point. The kNN adds a restriction upon the EN, sorting the results of EN and selecting only the closest k points as neighbors.

Neighbor search allows the program to inspect the point cloud by parts, which happens to be useful for many applications. In fact, support for *neighbor search* is common in well-known libraries for point cloud processing such as the Point-Cloud Library (PCL) [117, 130], and Open3D [173]. But before explaining the mathematical details and data structures used for *neighbor search*, we motivate its usage in some applications below.

Segmentation

Autoware's segmentation module *euclidean_cluster* [129], for example, utilizes *neighbor search* to cluster points. In the beginning, the whole point cloud is to be segmented. One point from the point cloud is chosen and used as *query* for *neighbor search.* The found neighbors and *query* are labeled as part of the same cluster. Additionally, each found neighbor is subsequently used as a *query* for a new *neighbor search.* The found neighbor points are cumulatively included as part of the same cluster. When the *neighbor search* cannot find any new point (i.e., a point that was not already found), the cluster is complete. From that, another unprocessed point from the point cloud is chosen, and the process repeats until the whole point cloud is segmented into different clusters.

Localization

The *neighbor search* is also used for localization. In Autoware's *ndt_matching* [93], for example, the point cloud map (also known as the reference point cloud) is broken down in a grid of 3D cells (also known as voxels). The points contained by each cell in the map are used to build a normal distribution, and the centroids (spatial average) of the points in each cell are used to build a new, filtered, reference point cloud. each centroid point has a reference to the voxel information. Later, the sensed LiDAR points perform *neighbor search* in the reference point cloud to find neighbor centroids and retrieve voxel information. The coordinates of the sensed points are compared against the normal distribution of the voxels of interest. This comparison returns a score, being maximum at the peak of the distribution, which intuitively indicates how aligned the point is with the expected distribution of points in that region. The scores of the points feed an optimization algorithm that tries to maximize the alignment between the sensed point cloud and the reference one. For this, it rotates and translates the point cloud, improving its alignment. The steps repeat until the alignment error is below a threshold.

Normal Estimation

Normal Estimation is another typical algorithm for point clouds in applications such as surface reconstruction and computer graphics [111]. The goal is to infer the direction of the normal vector anywhere in the point cloud. This is useful to, for example, infer supporting planes of the surfaces and apply light reflections if building a 3D model. In this case, the *neighbor search* retrieves the neighbor points of the coordinates where the normal is to be found. The neighbors are used to build a covariance matrix, upon which the eigenvectors and eigenvalues are used to obtain the normal. Mathematical details can be found in [129].

3D DNNs

The success of Machine Learning (ML) for image processing motivated works in studying DNN for 3D data as well. Although the initial implementations such as PointNet [118] and PointNet++ [119] were not very performing, they showed that it was possible to train models to perform classification and segmentation with ML over point cloud data. However, DNNs require regions from the input to assemble a feature matrix that captures local structures. In 2D images, these matrices are filled with neighbor pixels, which are regularly distributed in the image. In LiDAR point clouds, however, neighbor points are not well structured in memory. Thus, it is necessary to perform *neighbor search* to find the neighbors and fill the input matrices before the DNN layers take place.

3.1.3 Efficient Neighbor Search with K-D Trees

Point clouds obtained with LiDAR sensors are not organized in any manner. Instead, points are usually pushed to an unsorted array as they are collected by the sensor. Thus spatial locality does not imply memory locality, and neighbors can be anywhere in the list of points. Given the low latency requirements of AD computations, exhaustively searching within the tens of thousands of points found on point cloud frames is prohibitive. Instead, a search-friendly data structure is necessary. Among different data structures, k-d trees [26, 51] are widely used in practice (e.g., PCL [117, 130] and Open3D [173]) as they are efficient for pruning the search space on low-dimensional data such as LiDAR-acquired point clouds.

A k-d tree [26, 51] is a binary tree used for partitioning k-dimensional spaces. During its building process, the k-d tree recursively splits the data space into two subspaces, separating the data in one coordinate. The splitting coordinate is typically the one with the higher spread, and it takes place in its median value, as exemplified in Figure 3.5 (in 2D for simplicity). The splitting process continues recursively in each subspace until it contains $\leq N$ elements. This terminal subspace is represented by a leaf node in the k-d tree and will point to up to N elements of the indexed data.

Later, a query point q with coordinates (x, y, z) can efficiently search its neighbors using the k-d tree. At each visited node, starting from the root, the query checks the splitting coordinate and its value. Based on it, it calculates which subspace it belongs to and descends into the corresponding sub-tree. This is shown in pink in Figure 3.5. This navigation policy aims to quickly reach the leaf with the best potential neighbors for the query. In the leaf, we calculate the distance between the query q and each point $p_i \mid i \leq N$ held by the leaf. In 3D, the distance is calculated as shown in Equation 3.1.

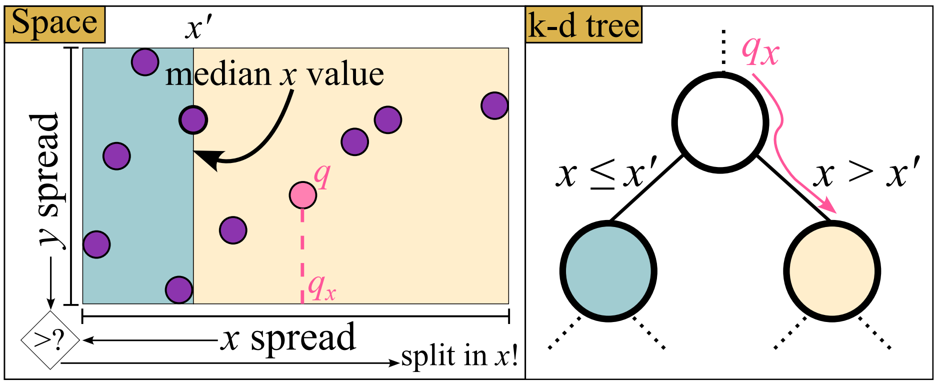


Figure 3.5 The k-d tree splits the data in two, at the median value of the coordinate with the higher spread, which is stored at the node. During the search, a query q uses this value to decide how to descend on the tree.

$$d(q, p_i) = \sqrt{(q_x - p_{ix})^2 + (q_y - p_{iy})^2 + (q_z - p_{iz})^2} \tag{3.1}$$

To avoid performing the square root, a common optimization is to calculate the squared Euclidean distance.

$$d^2(q, p_i) = (q_x - p_{ix})^2 + (q_y - p_{iy})^2 + (q_z - p_{iz})^2 \tag{3.2}$$

With the distance at hand, it is possible to classify a point as a neighbor or not, and if so, add it to the *list of neighbors* for the query q. For the EN, a point p_i is a neighbor if it satisfies $d(q, p_i) \leq r$ (where the distance r is a search parameter), as shown in Equation 3.3. For the kNN, the *list of neighbors* is limited to k elements (where k is also a search parameter), sorted by distance. Therefore, for the kNN, a point p_i must also satisfy $d(q, p_i) < list\_of\_neighbors[k-1]$ to enter the *list of neighbors* of q.

$$classification(q, p_i) = \begin{cases} neighbor, & \text{if } d^2 <= r^2 \\ not\ neighbor, & \text{if } d^2 > r^2 \end{cases} \tag{3.3}$$

When the recursion unwinds, the query decides if it is necessary to visit the other sub-tree or not. It will if the other sub-tree is sufficiently close in the *splitting coordinate* so that q might find neighbors there. Otherwise, it unwinds and repeats the test in the upper levels. Whenever it visits the other sub-tree it descents with the same policy as explained before, until it reaches a leaf and repeats the leaf processing. Therefore, a query might visit multiple leaves to find all of its neighbors.

3.2 Related Work and State-of-the-art Hardware Support for Point Cloud Processing

As we saw in this and previous chapters, there is increasing use of LiDAR in 3D applications such as AD, and also others out of the scope of this thesis such as Virtual Reality (VR). This motivated recent works on improving point cloud processing. We hereby cover state-of-the-art proposals that, like us, saw the need for better point cloud processing.

To improve point cloud *neighbor search* several ASICs were proposed. One of the first accelerators for EN and kNN was proposed by Heinzle et al. [62]. Their key idea is to search slightly more points than necessary (e.g., with a greater k or r), and search in this pool of neighbors for subsequent, spatially closed queries. The accelerator is prototyped in an Field-Programmable Gate Array (FPGA), and achieves 68% of the CPU performance. According to the manuscript, this slowdown is caused by the low frequency of the FPGA. Since their measurements do not include transfer costs between the CPU and the FPGA, end-to-end performance is likely to be even worse. Tigris [165] accelerates the *neighbor search* targeting the localization task. The accelerator has multiple recursion units to search multiple independent queries in parallel. It also has a pool of processing elements to calculate the distance between the query and points during leaf processing. Tigris also has a scheme to search on previously obtained neighbors, employing approximation, losing some accuracy for the sake higher of performance. EdgePC [167] proposes data structure changes to accelerate sampling and searching points for 3D deep learning analytics with a GPU. They structure the point cloud with Morton code and use approximation to speed up the search. They do DNN re-training to minimize the accuracy drop caused by their approximate search to find features.

Another work, QuickNN [116], also proposes an accelerator for *neighbor search.* They implement it in an FPGA platform and focus on optimizations to decrease external memory bandwidth. Optimizations include a write-gather cache, to accumulate similar accesses before accessing external memory, and caches for the k-d tree nodes. A more recent work, ParalellNN [35], also proposes an accelerator for *neighbor search.* They rely on a High Bandwidth Memory (HBM) to reduce the bandwidth burden between the accelerator and the memory system. Differently from the previously mentioned accelerators, they index their point cloud in an octree data structure. The accelerator has ASIC modules to build and search the octree in parallel. To simplify the hardware, they do not perform backtracking (i.e., when a query unwinds the recursion and visits the other sub-tree for potential neighbors), introducing approximation in their result. The Crescent accelerator [50] divides the tree into a top tree and a set of sub-trees. They allow backtracking to happen only inside a sub-tree to avoid conflicts when accessing memory. The approximation incurred by limited backtracking is also amortized during re-training, as they target 3D DNNs applications. In another recent work, the geometric similarity in Convolutional Neural Networks (CNNs)' point cloud analytics was exploited [34]. A completeASIC for 3D CNN is proposed, with optimizations including the *neighbor search*. For that, they *voxelize* the point cloud limiting *neighbor search* to perform only on adjacent voxels of the query. Inside each voxel, however, comparisons are comprehensive. In time, there are accelerators for point cloud DNNs [49, 87, 140] with no specific proposals or techniques to improve *neighbor search*. While they do perform *neighbor search*, improving it is not their main goal.

While accelerators can be very efficient and provide considerable speedup for *neighbor search*, typical point cloud processing algorithms interleave *neighbor search* with other operations. However, both QuickNN [116] and ParalellNN [35] evaluate their accelerators with a workload that *only* performs neighbor searches across multiple points from the KITTI [55] dataset. Therefore, their reported speedups of 19× (QuickNN) and 107.7× (ParalellNN) against the PCL on a CPU may aggressively decrease when used in a real scenario application (Amdahl's law). Out of the mentioned works, two [165, 167] report end-to-end gains and confirm this behavior. Tigris [165] reports a speedup of 392.2× for the search *only*, which converts into 1.86× end-to-end speedup for a localization application running in a Xeon CPU. EdgePC, reports a 3.68× speedup for sample and neighbor search, which converts into a 1.55× speedup in end-to-end inference over a Volta GPU. Also, they can incur significant silicon costs (and thus also money). Let us look at the Tigris accelerator [165] as an example since it only performs *neighbor search* (other accelerators embed support for some complete task, e.g., DNN execution). Tigris requires more than 8 MB for buffers, and a total area of $15.57mm^2$. To put in perspective, the Arm-based Samsung Exynos CPU M6 has 4 MB in their L3 Cache [58].

At the same time, some previous works also tried to improve neighbor search in a CPU. The work in [3] exploits CPU SIMD hardware for neighbor search. However, it targets high dimensional spaces and the corresponding data structure for that. In their case, they use Asymmetric Distance Computation (ADC) to index the searchable data. A US patent from 2018 [57] describes a way to use SIMD operations to update the list of neighbors in kNN, avoiding iterations to find the correct position of the new neighbor in the ordered list. This, however, applies to a small part of the neighbor search.

Finally, some works exploited GPUs to improve point cloud processing. The Buffer k-d tree [56] proposes nearest-neighbor search using a buffer to delay the processing of queries of the same leaf until enough work is gathered. RTNN [174] proposes to formulate neighbor search into a ray tracing problem. It then exploits contemporary ray tracing hardware in

GPUs to improve the search. The work, however, shows to be effective on point clouds orders of magnitude (hundreds of thousands to millions of points) bigger than those generally processed in one LiDAR frame (thousands of points), as data transfer overhead shows to be increasingly relevant as the point cloud size decreases. Nguyen et al. [105] focus on the software perspective, implementing the *euclidean_cluster* task with different data structures and observing their efficiency in the GPU hardware. Nonetheless, evaluation of Autoware.ai algorithms had shown that using the GPU for the *euclidean_cluster* performs similarly to an Out-of-Order (OoO) CPU due to the GPU communication overheads [69]. Indeed, Autoware.ai uses the CPU instead of the GPU to run it by default [13]. The GPU is reserved for image-based object detection DNNs where its benefits are much less debatable [69].

3.3 Chapter Takeaways

In this Chapter, we detailed the functioning of LiDAR sensors and their use for AD. Moreover, we detailed the *neighbor search* operation, a backbone for point cloud processing, exemplifying its application in four different use cases. We also explained how k-d trees work and how it allows efficient search. We also provided a discussion on relevant state-of-the-art, with a focus on computer architecture and hardware proposals. We saw, for example, that most of the recent works focus on ASIC solutions. We highlighted concerns with this approach such as communication overheads, lack of programmability, and hardware costs. In the following two Chapters, we present two new proposals for augmenting CPUs for point cloud processing. Our approach intend to balance performance improvement, while offering programmability and low hardware cost. For that, we will leverage properties from LiDAR and k-d trees that we covered in this Chapter.

4 K-D Bonsai: Reducing Data Movement in Point Cloud Search

“The information content of a message is the measure of how much the recipient is surprised.”

Claude Shannon

In the previous Chapters, we saw the importance of point cloud processing and the neighbor search operation for AD. We advocated for better CPU support for it. In this Chapter, we propose *K-D Bonsai*, a technique to compress point clouds stored in k-d trees used in Euclidean Neighbors (EN) search (also known as *radius search*, or *ball query search*). The compression reduces data movement between the CPU and the memory hierarchy when inspecting the leaves of the k-d tree, improving performance and energy efficiency of EN search.

To perform the compression, we first observe that LiDARs have a physically limited range of operation, defining an upper-bound value for the distance of collected points, and hence, an upper-bound for each of its coordinates (x, y, z). For example, the Velodyne HDL-64E [152] - a typically employed LiDAR sensor - has a maximum operation range in the order of 120 m. By consequence, the FP exponent fields of the collected points have a roof. This allows some bits of the exponent to be ignored without information loss. Second, we observe that tree leaves of k-d trees [26, 51], intrinsically hold points with similar values. As a consequence, the sign and exponent fields (in IEEE 754 FP representation [138]) are frequently repeated across different points in the same leaf. The repeated values can be merged, again with no information loss. Third, we observe that it is also possible to reduce the bits of the mantissa field, losing information in a bounded manner, and still computing a large percentage of EN without losing *correctness*. More importantly, we show how to cheaply identify any correctness loss at run-time, and re-issue full-precision computation to keep baseline accuracy with minimal overheads.

The mechanism is implemented through Instruction Set Architecture (ISA) extensions, which we named Bonsai-extensions, requiring minimal extra hardware in a traditional CPU. We use the Bonsai-extensions to compress the k-d tree during its construction phase and to read and operate over compressed data during the search phase. The Bonsai-extensions reduce the number of necessary bytes to find the EN, decreasing the number of memory accesses, energy consumption, and execution time. The CPU modifications are punctual, incrementing over existing hardware. Moreover, our solution provides flexibility, since improvements are exposed as new CPU instructions. Thus, with minimal software changes, existing applications can immediately leverage the benefits of our proposal. In contrast with expensive and hard-to-program out-of-core accelerators, K-D Bonsai is a much less intrusive option to enhance next-generation ADS.

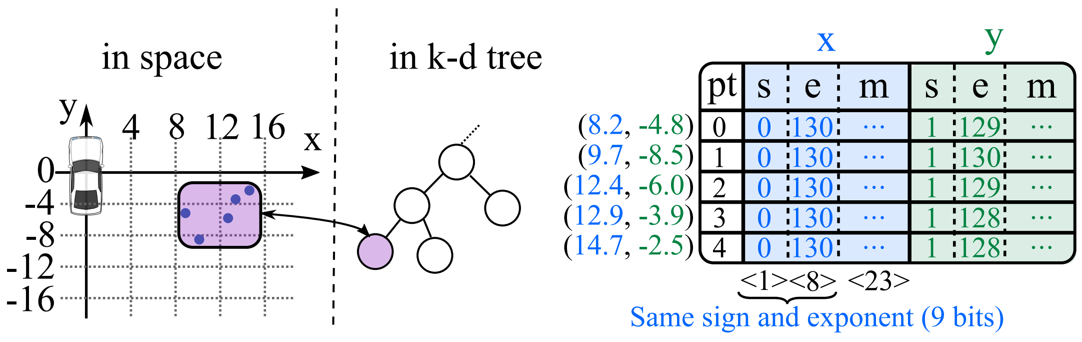
4.1 Compressing Point Clouds on K-D Trees for Radius Search

In this section, we explain how k-d-trees can be compressed when used for EN search in AD tasks, reducing the number of bytes needed to fetch the points during leaf inspection. We discuss a twofold compression approach that uses both value similarity and a smaller representation. Finally, we discuss the errors introduced (by a smaller representation; value similarity does not introduce any error) and our approach to detecting and correcting them, guaranteeing the baseline accuracy.

4.1.1 Compression Based on Value Similarity

When the k-d tree is built (as explained in section 3.1.3), the point cloud space is subdivided in a way that nearby points end up together in the leaf nodes. Hence, the coordinates of the points are similar to each other. This scenario is illustrated in Figure 4.1, in two dimensions for simplicity.

Figure 4.1a exemplifies a situation where spatially close points are held by the same k-d tree leaf node. The origin of the coordinate system is in the vehicle (where the LiDAR sensor is), and the distance to the points is given in meters. Figure 4.1b lists the coordinates of the points (x and y in this example), exposing their internal FP representation (in 32-bit IEEE 754 [138]). We depict the sign (s), exponent (e), and mantissa (m) fields of FP representation



	pt	x s	x e	x m	y s	y e	y m
(8.2, -4.8)	0	0	130	···	1	129	···
(9.7, -8.5)	1	0	130	···	1	130	···
(12.4, -6.0)	2	0	130	···	1	129	···
(12.9, -3.9)	3	0	130	···	1	128	···
(14.7, -2.5)	4	0	130	···	1	128	···

(a) Nearby points mapped to the same k-d tree leaf node.

(b) Floating-point fields for each point.

Figure 4.1 Nearby points in space are often held by the same k-d tree leaf, creating opportunity to compress data due to value similarity. Particularly, the sign and exponent fields frequently repeat within each point's coordinate.

separately. Following the IEEE 754 standard, the stored value is given by the following equation.

$$value = -1^{sign} \times 1.mantissa \times 2^{exponent-bias} \tag{4.1}$$

When points are close in space, their coordinates are likely to have the same sign (i.e., they all belong to the same quadrant in the coordinate system), and exponent (i.e., values are within the same power of 2). For example, all points in Figure 4.1 have their x coordinate between 8.0 and 16.0, hence yielding the *same* exponent field value of 130[1].

To check the applicability of this observation, we verified how often *sign and exponent* fields are the same for a given coordinate across all points in a leaf node (as it is the case for coordinate x in Figure 4.1a). We inspected a set of point clouds spanning more than 37 million points that feed the *euclidean cluster* node in Autoware.ai [13, 68] (details about data-set can be found in Section 4.3). We identified that 78% of leaf nodes have the same *exponent* and *sign* for the x coordinate, and 83% for the y coordinate.

Therefore, value similarity in internal fields of FP representation of point clouds is *very* common and a suitable compression source for k-d tree data. If the sign and exponent are the same in a coordinate across all points in a leaf, we can store them only once, and reconstruct the values inside the CPU, only when computation takes place (details in Section 4.2).

[1]For 32-bit FP (single precision), the bias is 127, resulting in a final exponent of $130 - 127 = 3$.

Table 4.1 Classification error of radius search for euclidean clustering using smaller floating-point representations.

	# of bits			Misclassified points
	Sign	Exponent	Mantissa	
IEEE-754 32-bits	1	8	23	0% (baseline)
IEEE-754 16-bits	1	5	10	0.076%
bfloat 16	1	8	7	0.61%
Custom float 24	1	5	18	0.0003%

4.1.2 Compression Via a Smaller Representation

Compressing the *sign* and *exponent* of FP representation fields (Section 4.1.1) yields a maximum compression ratio of 9 out of 32 bits per coordinate when 32-bit is used - the default in Autoware.ai and PCL, and the *baseline* considered in this work. To improve the compression ratio further, we need to work over the remaining 23 bits of the FP representation which belongs to the *mantissa*.

The problem here is that the *mantissa* field hardly repeats across the points in a leaf. Therefore, compression due to value similarity will not be fruitful for the *mantissa* bits. We can, however, reduce the size of the FP representation at the cost of precision. Table 4.1 depicts the error in classification (Equation 3.3) using different FP formats with less than 32-bits. We use the same set of point clouds as in Section 4.1.1. We experimented with two common 16-bit FP representations: *IEEE-754 16-bit* (IEEE half-precision format [138]), the *bfloat 16* (used for machine learning applications, and e.g., supported by CUDA [109]); and also a custom 24-bit representation, for a midway reference in our comparison.

Overall, we found that both 16-bit and 24-bit FP representations yield less than 1% classification error. This is a good indication that reducing the representation can be effective for compression, introducing few mistakes. Notice that for *IEEE-754 16-bit* and the Custom float (24 bits) representations the *exponent* field size is also reduced, affecting the *range* of representable numbers. However, point cloud data obtained from sensors such as LiDAR have limited range. For example, the Velodyne HDL-64E [152] (a typically employed LiDAR sensor) has a maximum cover range of 120 m. Indeed, none of the errors depicted in Table 4.1 are due to the lack of range to represent numbers. Hence, reducing *exponent* bits in our case is not a problem, but something to take advantage of[2].

[2]Lack of range representation due to fewer exponent bits could be a problem when the coordinate system of the point cloud does not have the origin on the sensor itself and is, otherwise, far away. For example, when point cloud maps [133, 134] are created, several point clouds are combined to represent a region. Hence,

Going further, we evaluate the involved trade-offs of the different representations to select a good fit for our compression scheme. We noticed that *IEEE-754 16-bit* has the same size as *bfloat*, but balances better the use of exponent bits (for range) and mantissa bits (for precision), being more accurate by an order of magnitude. Also, the 8 extra bits in our Custom (24 bits) float for increased precision do not pay off since the 16-bit formats already hold decent (<1% error) accuracy. Finally, the *IEEE-754 16-bit* is already partially supported by nowadays CPUs (e.g., for storage on Arm [7]) hence being less intrusive on existing architectures than a new custom format. For these reasons, we choose the *IEEE-754 16-bit* to represent the points of k-d tree leaves, and over that apply compression due to value similarity (Section 4.1.1).

Our main conclusions about using a smaller representation in k-d tree radius search are two-fold: i) the *mantissa* bits can be reduced with low accuracy loss; ii) AD algorithms consume points that are near the vehicle, hence the *exponent* bits can be reduced and still represent the point cloud values.

4.1.3 How to Keep Accuracy Despite a Smaller Representation

So far, we have discussed two different ways to reduce the size of points searched by k-d trees, with the side effect of introducing classification errors. However, since AD systems are safety-critical, introducing mistakes is not desirable [63] and pose consequences which are hard to test [74]. Hence, we propose an approach to detect possible mistakes in classification, and re-compute them with baseline accuracy. For this, we assume to have access to both the original points and the compressed points. The idea is to use the compressed points, alleviating memory usage, and exceptionally lookup for the original 32-bit values if a possible misclassification is detected.

Let B be a number in *32-bits IEEE-754* format that we want to represent in the *16-bit IEEE-754* format, at the cost of an error δB associated with the loss of precision. Let B' denote the resulting value of B in 16-bit representation.

$$B' = B + \delta B \tag{4.2}$$

For the default rounding mode in the IEEE-754 Standard, the Least Significant Bits (LSBs) of the mantissa are dropped, and the resulting number is rounded up or down, towards the nearest number. For values whose exponent can be stored equally in both representations (our case, see Section 4.1.2), the rounding in the mantissa is the single source of error. In

points can be more distant to the origin than the sensor range. A possible solution for this case is to translate the origin to a more convenient position. This could be done offline or when the map of the region is loaded.

this case, the 11^{th} to 23^{rd} *mantissa* bits will be used to round the number to its nearest value, adjusting the 10^{th} bit of the 16-bit resultant number.

Since we can round up or down to the nearest number, the *maximum mantissa error* will be half the value of the $10th$ bit, while the *maximum value error* will also depend on the exponent, since $2^{exponent-bias}$ multiplies the mantissa to form the FP number (Equation 4.1). In these conditions, the maximum error δ for rounding a number B when converting it from 32-bit to 16-bit IEEE-754 FP is given by:

$$max(\delta B) = 2^{exponent-bias} \times \frac{2^{-10}}{2} = 2^{exponent-bias} \times 2^{-11} \tag{4.3}$$

The takeaway here is that using only the exponent one can infer the maximum rounding error. Thus, with B' at hand, there is no need to lookup B, as the exponent value is representable in both B' and B according to our assumptions.

Now, let's proceed to find the error in the squared difference between a value A, in 32-bit, and a value B', in 16-bit. We start looking at the subtraction, applying Equation 4.2.

$$A - B' \;=\; (A) - (B + \delta B) \;=\; (A - B) - \delta B \tag{4.4}$$

Where $-\delta B$ is the associated error. We can proceed and evaluate the error for the square operation $(A - B')^2$ applying Equation 4.2, Equation 4.4, and Newton's binomial theorem.

$$\begin{aligned}
(A - B')^2 &= [(A - B) - \delta B]^2 \\
&= (A - B)^2 - 2(A - B)\delta B + \delta B^2 \\
&= (A - B)^2 - 2[A - (B' - \delta B)]\delta B + \delta B^2 \\
&= (A - B)^2 - 2(A - B' + \delta B)\delta B + \delta B^2 \\
&= (A - B)^2 - 2[(A - B')\delta B + \delta B^2] + \delta B^2 \\
&= (A - B)^2 - 2(A - B')\delta B - 2\delta B^2 + \delta B^2 \\
&= (A - B)^2 - 2(A - B')\delta B - \delta B^2
\end{aligned} \tag{4.5}$$

Where $-2(A-B')\delta B-\delta B^2$ is the associated error of the square of the differences operation (ϵ_{sd}). Notice that δB can be either positive or negative, depending if the number was rounded up or down. At run-time, however, we will not know which case it was because that would require fetching and inspecting the LSBs of the original value, which we are trying to avoid. Instead, we can be pessimistic and calculate the *worst case* magnitude of ϵ_{sd}, using the $max(\delta B)$ (Equation 4.3) instead of δB.

$$max(\epsilon_{sd}) \;=\; 2 \cdot |A - B'| \cdot |max(\delta B)| + max(\delta B)^2 \tag{4.6}$$

Again, notice that the $max(\delta B)$ and $max(\delta B)^2$ can be directly obtained with the exponent of B'. Finally, we can compute the approximate square differences of form $(A - B')^2$ for each coordinate, and sum to get the approximate euclidean distance squared d'^2.

$$d'^2(q, p_i') = (q_x - p_i'x)^2 + (q_y - p_i'y)^2 + (q_z - p_i'z)^2 \tag{4.7}$$

Likewise, we can sum the maximum error of the squared differences in each coordinate and get a total error $T\epsilon_{sd}$

$$T\epsilon_{sd} = max(\epsilon_{sd})_x + max(\epsilon_{sd})_y + max(\epsilon_{sd})_z \tag{4.8}$$

We can finally use Eqs. 4.7 and 4.8 to perform the classification (with p_i' instead of p_i).

$$classification'^2(q, p_i') = \begin{cases} neighbor, & \text{if } d'^2 <= r^2 - T\epsilon_{sd} \\ not\ neighbor, & \text{if } d'^2 > r^2 + T\epsilon_{sd} \\ use\ Eq.\ 3.3, & \text{otherwise} \end{cases} \tag{4.9}$$

In other words, we can use the worst-case error $T\epsilon_{sd}$ to confirm the correctness of the classification with p_i'. We do so by defining a shell around r^2 with values $r^2 - T\epsilon_{sd}$ and $r^2 + T\epsilon_{sd}$, as depicted in Figure 4.2. Whenever d'^2 falls outside the shell, the classification is the same as the baseline, computed by Equation 3.3. For instance, a point inside the radius but outside the shell cannot be outside the radius even if we add $T\epsilon_{sd}$ to d'^2. On the other hand, when d'^2 falls inside the shell, the error could be large enough to change the classification, and cannot be guaranteed to be the same as the baseline. In this case, we propose to fetch the original point p_i, and re-do the classification with the full-precision, using Equation 3.3.

4.2 Proposed Design

In this section, we motivate and explain the design decisions of *K-D Bonsai*. We explain the hardware structures and how to use them through new instructions, the *Bonsai-extensions*.

4.2.1 Hardware Support for K-D Tree Compression

After deriving a compression scheme (Section 4.1), hereby referred to as *K-D Bonsai*, it is of our interest to use it in tasks that perform *radius search*. A naive approach would be to (de)compress points with a software-only solution. However, iteratively inspecting and re-ordering bits in software slows down *radius search* in the order of 7× (data-set

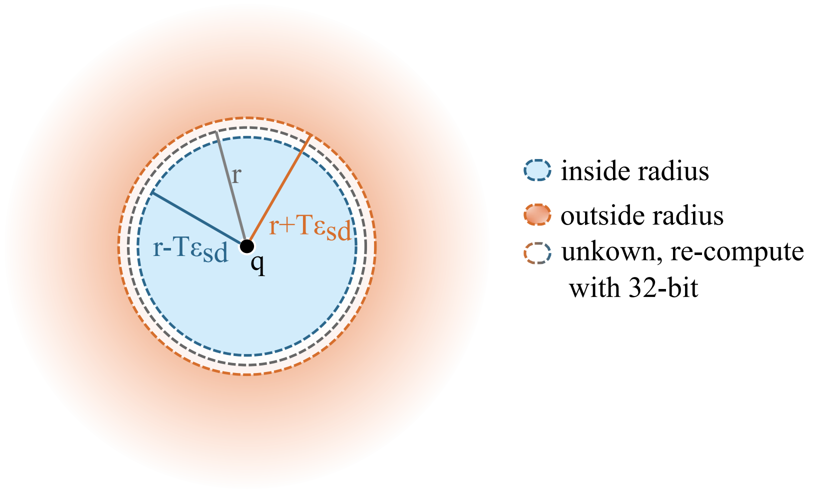


Figure 4.2 Visual representation of Equation 4.9.

and experimentation platform in Section 4.3.1), undermining the compression benefits. Alternatively, it is possible to add hardware to support *K-D Bonsai* effectively.

Two main options arise to implement *K-D Bonsai* in hardware: i) with an out-of-core accelerator; or ii) in the CPU through ISA-extensions. In this work, we stand with the latter as we justify next. First, the CPU would have to transfer data in and out to communicate with the accelerator. However, the leaf processing done by K-D Bonsai is a fine grain task, requiring only a handful of cycles to complete (implementation details in Section 4.2.2). Thus, using proper hardware inside the CPU to perform (de)compression and classify points avoids communication costs [135]. Alternatively, (de)compression operations could be coalesced to amortize communication costs. However, accelerators are likely to be more expensive (see Section 3.2). At the same time, leaf processing is only a fraction of the point cloud handling, limiting the maximum performance improvement (Amdahl's law), and jeopardizing accelerator adoption. Nevertheless, industry favors less experimental approaches to accelerate tasks in their real-life solutions, rarely employing accelerators [114].

On the other hand, while new instructions yield more conservative performance gains, they are a much simpler solution from the hardware standpoint. Additionally, ISA-extensions are easier to integrate and to program, facilitating *K-D Bonsai* implementation in existing platforms. For example, Arm releases new (sometimes optional) ISA-extensions yearly [6]. Also, some Arm processors support to-be-defined custom instructions [40]. Both alternatives exemplify the use of ISA-extensions to specialize CPUs for relevant scenarios, such as AD. Support for custom instructions is also a key feature of the RISC-V ISA [157].

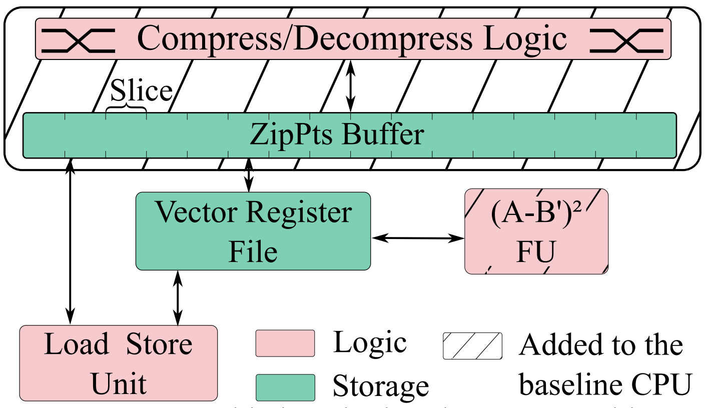


Figure 4.3 The new components added to the baseline CPU and how they interact with pre-existing ones.

This set of reasons motivates us to propose specific instructions in the CPU to implement *K-D Bonsai* effectively.

4.2.2 Changing the CPU

A main advantage of the ideas discussed in Section 4.1 is how easily and cheaply they can be carried out in the hardware. Indeed, the set of new functionalities required is small: i) we need to compress the data; ii) decompress the data; and iii) support computation of the squared differences (and associated error) in the form $(A - B')^2$ (Equation 4.5).

Figure 4.3 depicts the two components that we add to the CPU, and how they interact with the existing hardware. The first added component we discuss is the Compression/Decompression unit, at the top of the figure. The unit is divided into two parts: a buffer, named ZipPts Buffer, and a Compress/Decompress Logic.

ZipPts Buffer. The ZipPts Buffer is designed to hold both compressed and uncompressed 16-bit points, being the source and destination operand for compression and decompression operations. In our implementation, we restrict the ZipPts Buffer size to hold a maximum of 16 points (the number of points per leaf in the PCL is 15 by default). We also reserve space for 3 bits in the buffer, to encode whether x, y, and z coordinates are compressed.

The buffer has two 128-bit ports to interface with the Vector Register File and one 128-bit port to interface with the Load Store Unit. Hence, data is exchanged in chunks of 128-bit, which we refer to as a *ZipPts Buffer slice.* When less than 128-bits must be transferred (e.g., the last chunk of a compressed data), we pad data with zeroes. The width of the ports equals the ones that already exist in our baseline CPU (see Section 4.3), for example in the Vector Register File. Hence, we can load and store data from/to memory directly to the

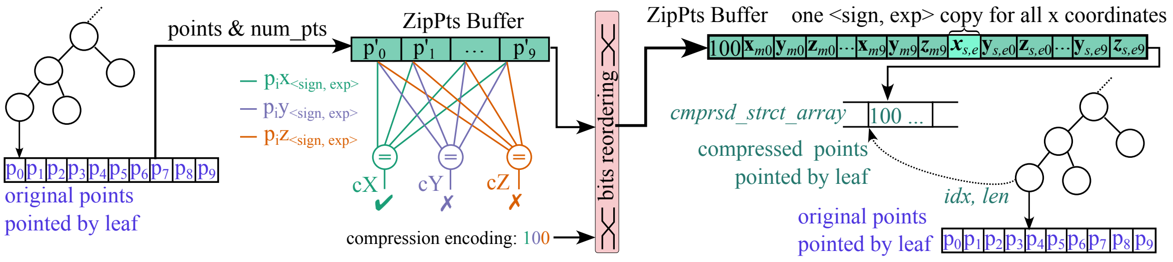

points & num_pts
ZipPts Buffer
p'0 p'1 … p'9
— pix<sign, exp>
— piy<sign, exp>
— piz<sign, exp>
cX ✔
cY ✗
cZ ✗
p0 p1 p2 p3 p4 p5 p6 p7 p8 p9
original points pointed by leaf
compression encoding: 100
bits reordering
ZipPts Buffer
one <sign, exp> copy for all x coordinates
100 xm0 ym0 zm0 … xm9 ym9 zm9 xs,e ys,e0 zs,e0 … ys,e9 zs,e9
cmprsd_strct_array
100 …
compressed points pointed by leaf
idx, len
original points pointed by leaf
p0 p1 p2 p3 p4 p5 p6 p7 p8 p9

Figure 4.4 Compressing points in a leaf node. Load the points into the ZipPts Buffer. Find coordinates with same $< sign, exponent >$ pairs, e.g., x coordinate (setting cX to 1). Reorder bits in the ZipPts Buffer and set the compression encoding. Store compressed data in the memory (*cmprsd_strct_array*), adding a reference to it in the leaf node for future look-up.

ZipPts Buffer. In summary, we can load points into the buffer to be compressed, store the compressed data back in the memory, and load compressed data to be decompressed. Also, we can write values from the ZipPts Buffer into the Register File, exposing them to the Functional Units (FUs). The ZipPts Buffer is tightly coupled with the Compress/Decompress Logic, which is responsible for re-arranging the data bits, discussed as follows.

Compress/Decompress Logic. This unit re-arranges the data in the ZipPts Buffer compressing and decompressing points from a k-d tree leaf. In both cases, the number of points must be provided to the logic. During compression, this unit reads and compares the tuple $< sign, exponent >$ on each coordinate of the points in the ZipPts Buffer, see Figure 4.4. If they are the same across all points, only one *copy* of $< sign, exponent >$ will appear in the resulting compressed data. Each coordinate has a compression bit flag (cX, cY, cZ) to indicate whether or not its $< sign, exponent >$ is compressed. During decompression, this unit reads the compression bit flags, re-organizing the data and re-creating the multiple instances of the single copy of $< sign, exponent >$ across all values.

To exemplify, Figure 4.4 details the compression flow and the organization of the compressed data. First, the mantissa values are directly bypassed to the buffer as they are not compressed. Then, the compressed tuples of $< sign, exponent >$ are placed in the ZipPts Buffer, followed by the remaining non-compressed tuples of $< sign, exponent >$. The three compression bits are placed at the very beginning of the buffer.

Approximate Square of Differences Functional Unit. When compressed points are fetched from memory and decompressed into 16-bit values, they can be moved from the ZipPts Buffer to the Vector Register File. At this point, the FU for the square difference with error computation can take place. The unit implements Equation 4.5, and can be used successive times (for each coordinate) to compute Equations 4.7 and 4.8 to perform the classification. Figure 4.5 details the internal scheme of the FU. It has two input operands,

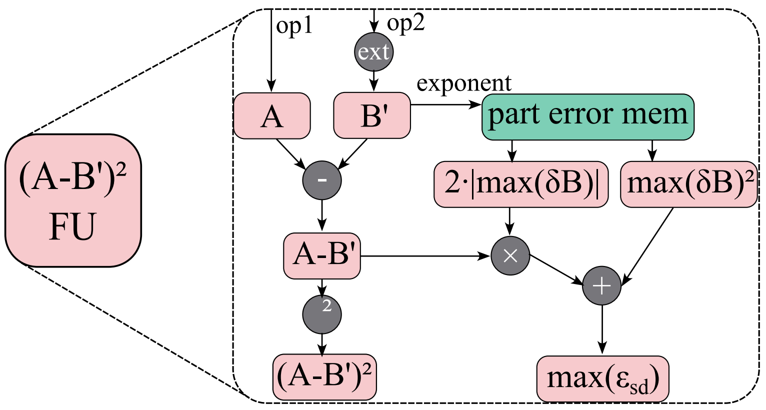


Figure 4.5 Details of the FU for square of the difference with error computation (Equation 4.5).

A is a 32-bit value (e.g., a coordinate of the query point), and B' is a 16-bit value (e.g., the same coordinate of one of the points in the leaf), which is then extended to 32-bit (without changing the value of B') so computation takes place in 32-bit hardware, preventing 16-bit errors to be magnified. The square of the differences proceeds with conventional subtraction and square operations.

The calculation of the worst case error ($max(\epsilon_{sd})$, Equation 4.6) has more operations than the square of the differences itself. Fortunately, we can take advantage of some observations to simplify its computation. First, since the $max(\delta B)$ depends only on the exponent of B', and there are only $2^5 = 32$ possible exponents, we can pre-compute the values of $2 \cdot |max(\delta B)|$ and $|max(\delta B)|^2$ and store them in a small (32 lines) lookup table. This small table (named *part error mem* in Figure 4.5) is looked up with the exponent of B' in the beginning of the operation. Also, the term $|A - B'|$ computed for the square of the differences can be borrowed to compute the worst-case associated error $max(\epsilon_{sd})$.

Since decompression outputs multiple points at once, they are simultaneously available for computation. To leverage this, we instantiate multiple approximate squares of difference FUs (Figure 4.6), to compute them in a vector manner. In each FU we compute the square of the differences and the associated error at the same time, each working on a part of the input vectors vA and vB'. For the EN classification, a coordinate of the query q is loaded into all indices of vA, while the same coordinate of multiple points is loaded into vB'.

4.2.3 Software Impact

Now we discuss how to use the new hardware from the software. We expose the new hardware functionalities mentioned in Section 4.2.2 as new CPU instructions. The set of new instructions, which we refer to as *Bonsai-extensions*, is described in Table 4.2. We

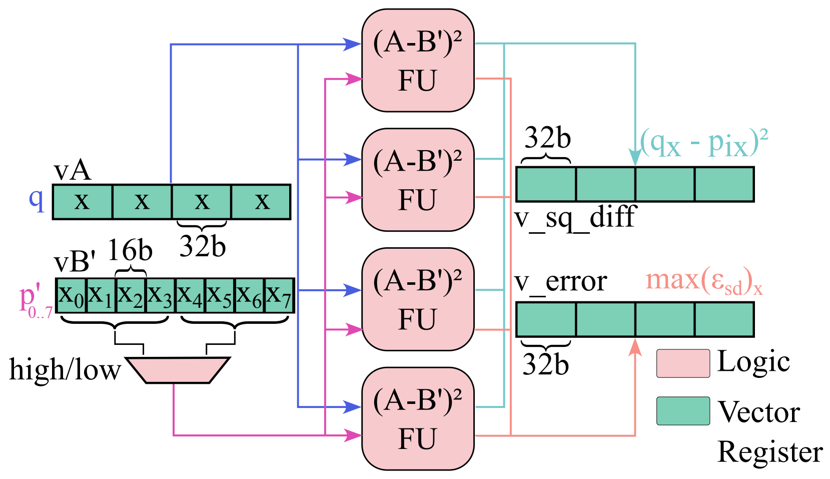


Figure 4.6 Vector square of the differences FUs.

divide the *Bonsai-extensions* into three instruction categories: compress, decompress, and computation. Some instructions trigger multiple micro-operations, as we explain together with their usage following.

When the leaf node is created during the k-d tree construction, we can use the compress instructions over the leaf points we have at hand (Figure 4.4). For such we have to load the points, one by one, into the ZipPts Buffer using the **LDSPZPB** instruction. The load converts the original 32-bit into 16-bit before placing the coordinates in the buffer. We can further compress the data in the ZipPts Buffer, looking for sign and exponent sharing, with the **CPRZPB** instruction. At this point, we have a compressed structure in the ZipPts Buffer and the resulting size in bytes (length). We can proceed and store the compressed data with the **STZPB** instruction, indicating the amount of *ZiptPts Buffer slices* that must be stored in memory. The decoder will generate one store micro-operation for each *slice*, storing them in consecutive addresses.

In our modified PCL code, we create an extra array of bytes, *cmprsd_strct_array*, to store the compressed structures consecutively as we visit and compress leaf nodes during the tree construction. Also, we keep track of the starting address and length of the compressed structure placed in the *cmprsd_strct_array* in the k-d tree, so that we can fetch the compressed data later, during the radius-search (tree traversal). We use C unions to re-use fields of the tree that are not used on leaf nodes (e.g., the splitting coordinate and distances to children), to store this information. Hence, we hold auxiliary compression information without increasing the size of the k-d tree. In the PCL code, we also keep track of the next free index in the array, to be occupied by the next compressed structure.

Later, when we do the radius search we can use the **LDDCP** instruction to load and decompress the compressed structure into registers, whenever we reach a tree leaf. This

Table 4.2 The proposed bonsai-extension instructions.

	Instruction	Description
Compress	**LDSPZPB** r_index, [r_addr]	**L**oa**D** **S**ingle-float **P**oint into **Z**ip**P**ts **B**uffer - Loads one 3D point in single-float from address *[r_addr]*, converts it to 16-bit, and place it on the ZipPts Buffer at position *[r_index]*.
	CPRZPB r_size, r_num_pts	**C**om**PR**ess **Z**ip**P**ts**B**uffer - Compress the 16-bit points from the ZipPts Buffer, exploiting the value similarity concept (Section 4.1.1). The number of points is informed in *r_num_pts*. The result of the compression is the ZipPts Buffer itself. The size in bytes of the resulting compressed structure is placed in *r_size*.
	STZPB [r_addr], #ZipPtsSlices	**ST**ore **Z**ip**P**ts**B**uffer - Stores the ZipPts Buffer in the memory. Due to port size limitations, the ZipPts Buffer will be stored in slices through several store micro-operations (in a total of *#ZipPtsSlices*).
Decompress	**LDDCP** v_base, r_num_pts, [r_addr], #ZipPtsSlices	**L**oa**D** **D**ecompressing **C**ompressed **P**oints - Load the compressed structure from memory into the ZipPts Buffer, in slices, through several load micro-operations (in a total of *#ZipPtsSlices*). Decompress the ZipPts Buffer on itself with one micro-operation. Writes-back the points to vector registers, per coordinate, from *v_base* up to *v_base + 5*, with 3 micro-operations. Since two 128-bit registers can hold up to sixteen 16-bit values (enough for one coordinate), we write-back to *six* (two at a time) 128-bit registers to hold forty-eight 16-bit values (enough for three coordinates).
Computation	**SQDWEL** v_sq_diff, v_error, vA, vB'	**SQ**uare **D**ifference **W**ith **E**rror **L**ow part - Performs a vector operation in the form $(A_i - B_i')^2$ with error calculation (see Eq. 4.5). The four values in the low part of *vB'* will be extended from 16-bit to 32-bit when pushed in the units (see Figures 4.5 and 4.6). The square difference will be placed in *v_sq_diff*, and the associated error in *v_error*.
	SQDWEH v_sq_diff, v_error, vA, vB'	**SQ**uare **D**ifference **W**ith **E**rror **H**igh part - Same as SQDWEL, but using the high part of *vB'*.

instruction is broken down by the decoder into a sequence of micro-operations. First it loads the compressed structure into the ZipPts Buffer. For this we need the address and size of the compressed data in the *cmprsd_strct_array*, which is kept in the tree leaf, to indicate how many *slices* (chunks of 128-bits) must be brought from memory, starting from the provided address. The decoder use the indicated number of *slices* to generate an equivalent number of load micro-operations from memory to the ZipPts Buffer. Once the whole compressed structure is inside the ZipPts Buffer, a decompression micro-operation takes place, reading the compression encoding and reordering the bits into 16-bit points accordingly. Finally, write-back micro-operations are issued to move the value of the points into the vector register file. In this case, we write back the decompressed points from the ZipPts Buffer into six vector registers. We need two vector registers for each coordinate since each vector register can hold up to eight 16-bit values, and we support up to sixteen 16-bit values per coordinate.

Finally, when we have decompressed the 16-bit values of the coordinates in the vector arrays, we can use the square of the differences FUs (Figure 4.6). For such, we perform instructions **SQDWEL** and **SQDWEH**, calculating the square of differences for points with a vector of the query point, for each coordinate. The coordinate values of the query point can be loaded into vector registers using existing vector instructions. Since we have *four* 32-bit lanes in the baseline CPU SIMD unit (Arm Neon, details in Section 4.3.1), but *eight* values on each coordinate (16-bit computed in 32-bit in the FUs, Figure 4.5), we split the values in two groups of *four* values, the low part and the high part, and compute them one at a time in the four lanes (details in Figure 4.6). The result, per point index, is available in two vector registers, one holding the calculated square of the differences, and another one with the maximum error ($max(\epsilon_{sd})$). Thereafter, it is possible to accumulate the distances for each index on each coordinate, using existing instructions, and compare it with r^2, performing the classification (Equation 4.9). If the result is inconclusive (inside the white shell in Figure 4.2), one can proceed with the baseline code, i.e., read the 32-bit point and compute the 32-bit distance. This should be rare to guarantee good performance, otherwise compression/decompression will consume time with no real benefit.

Finally, we highlight that, for AD tasks, the tree is generally built once for each frame, in the beginning, and then searched multiple times, during the frame processing. This is important because compressing the leaf node points represents an overhead during tree creation. However, the compression benefits will appear during the search, when we load fewer data from the memory. For example, we verified an average of 52 visits for each created leaf node during the radius search for one of the input frames. Thus, the expectation is that loading less data, multiple times, amortizes the initial overhead.

4.3 Results

In this section, we explain the evaluation methodology and obtained results for K-D Bonsai.

4.3.1 Evaluation Methodology

From the software perspective, we rely on Autoware.ai [68] to experiment with our idea. Autoware is a state-of-the-art and open source software stack for AD, built with contributions from both academia and industry companies [13]. It has several algorithms to perform AD, from sensor processing and perception to actuation. In this work, we choose

Table 4.3 Sub-sampling error.

Mean Standard Error for Latency	IPC Relative Error	L1- D Cache Miss Ratio Difference	Branch Mispred. Difference
2.94%	4.68%	0.10%	0.03%

a representative algorithm from Autoware, namely euclidean cluster [129] to verify the benefits of our proposal in k-d tree *radius search*, although other algorithms are also subject to our optimizations (e.g., Autoware's localization algorithm [93]).

The euclidean cluster algorithm is a vital part of the perception pipeline of Autoware.ai. The algorithm clusters points of a source point cloud, useful for inferring objects' shape, geometry, and distance. Notably, it has been reported by previous works as one of the tasks with higher latency in the Autoware.ai pipeline [19]. Importantly, the euclidean cluster extensively performs the *radius search* operation to find nearby points that should belong to the same cluster.

We stimulate the euclidean cluster algorithm with a *subset* of point cloud frames from an eight-minute car driving sequence [67]. Because our cycle-accurate simulator (details next) executes several orders of magnitude slower than real hardware, we used systematic sub-sampling (fixed-size samples equally spaced in time) to select the *subset* of point cloud frames. The idea was inspired by previous work [5] and yields good results if the parameters (interval amount and length) are properly chosen. We experimented with several parameters finally settling on 20 samples of 300 milliseconds each – adding up to six seconds of real-life data and handling a total of 60 frames. Table 4.3 details sub-sampling errors, evidencing it as a fast and accurate proxy to the code behavior.

We implemented the Bonsai-extensions (Table 4.2) in the gem5 simulator [29, 92], targeting an OoO CPU with the Arm's AArch64 ISA. We base our model (see Table 4.4) on the pre-defined *big* CPU in gem5, adjusting parameters such as the frequency to match technology scaling, to replicate an Arm Cortex A72 behavior. Although our solution is ISA-agnostic, we used Arm as a representative ISA for AD (e.g., used by NVIDIA DRIVE [108]). We modified the PCL [117] version 1.10 and its auxiliary library FLANN [94] version 1.9.2, using our instructions during the *radius search*, as explained in Section 4.2.3. We did not modify the compiler but instead wrote our instructions directly with byte-code using the *.inst* directive in Arm *asm* inside the library. We expose a Boolean variable in PCL so that users can activate the use of the new instructions for *radius search*. When the variable is *true*, the code uses the Bonsai-extensions, otherwise, it uses the baseline code. The search result is the same in both cases.

Table 4.4 Baseline CPU model.

Parameter	Value
CPU	OoO ARM v8 64-bit @3GHz, Fetch Width: 3, Issue Width: 8, Int Physical Reg.: 90, Float/Vector Physical Reg.: 256, ARM v8 NEON (128-bit SIMD operations)
Memory System	L1: 32KB (I) 2-way + 32KB (D) 2-way, L2: 1MB 16-way, Main Memory: 8GB DDR3-1600

We execute Autoware's euclidean cluster algorithm in gem5, running in Full System mode (Ubuntu 18.04). We use gem5 fast-forwarding capabilities with KVM hardware virtualization [60, 131] to reach the regions of the sub-sampled frames. For energy results, we model the CPU in McPAT [83, 84] in a 32 nm technology, and use gem5 reported statistics to feed the McPAT power model. We estimate the area and power of the new FUs (compression/decompression, and square of the differences with associated error) synthesizing Verilog descriptions on Synopsys Design Compiler [41] with a 14 nm technology [95]. To unify results in a single technology we scale the baseline CPU data reported by McPAT using the methodology described by Stillmaker et al. [142] (from 32 nm to 14 nm technology).

4.3.2 Performance Analysis

Figure 4.7a presents key performance metrics for the execution of the *extract* kernel of euclidean clustering, both for the baseline with and without the Bonsai-extensions. This is the main kernel of the algorithm and accounts for 90% of its execution time (measured with Valgrind [104]), and where both k-d tree build and search are performed. Since each metric has different scales, we normalized each of them w.r.t. the baseline code. We can see that the Bonsai-extensions reduce the number of memory instructions, by 23% for loads and 18% for stores.

Figure 4.7b gives intuition for this improvement, depicting a great reduction in the number of required bytes to bring the *points* from memory during the search on one frame. When we load compressed points using the Bonsai-extensions, we load a fraction (37%) of the bytes we would normally need in the baseline code. Although this value is for the first frame of the data set, the behavior is similar across all frames.

This reduction in memory usage converts into several benefits. First, it decreases the number of committed instructions by 16%, ultimately indicating that our Bonsai-extensions cut computation costs and increase efficiency on *radius search* processing. Second, it reduces accesses to L1 D-cache by 14%, making the application less memory-bound, which also

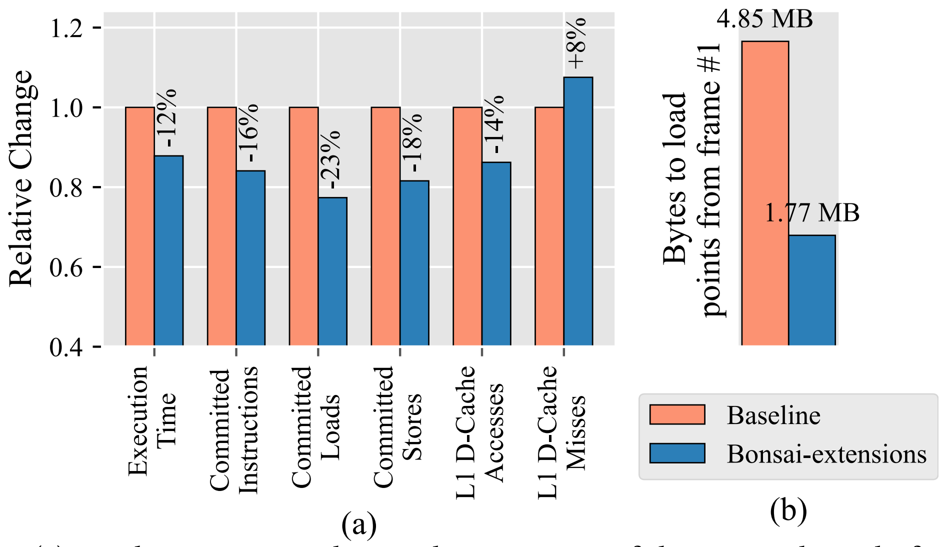


Figure 4.7 (a) Hardware metrics during the execution of the *extract* kernel of euclidean clustering considering the baseline code and the proposed Bonsai-extensions. Average across all executed frames. (b) Number of loaded bytes to fetch points from the first frame of the data-set during *radius search* (traversal).

increases efficiency in the use of the CPU. Third, due to both former reasons, it decreases the execution time of the *extract* kernel by 12%. Latency, as we further discuss, is a major concern for AD algorithms [86]. Nevertheless, this is particularly significant when we observe that benefits come from the addition of only five new instructions to the ISA.

Figure 4.7a also indicates K-D Bonsai increases L1 D-cache misses. Although the Bonsai-extensions load compressed points from the *cmprsd_strct_array*, which is contiguous in memory, it also accesses the original list of points when classification is inconclusive (white shell in Figure 4.2). These infrequent accesses to another data structure are the main cause for misses in higher levels of the memory hierarchy. In absolute numbers, however, this is not a concern. Since the L1 cache is accessed 47× more than L2 and 300× more than main memory we still see the benefits in execution time. Figure 4.8 puts the number of memory accesses in perspective, according to the different memory hierarchy levels. This phenomenon highlights the importance of choosing the appropriate reduced FP representation, as we discussed in Section 4.1.2, Table 4.1, to minimize overheads of issuing 32-bit re-computation. In our experimentation, only 0.37% of the classifications had to rely on the baseline computation. If we were not careful in selecting the representation, errors would not be as infrequent, and the K-D Bonsai benefits could be compromised.

Next, we evaluate end-to-end latency for euclidean cluster processing of frames. This is important because the *extract* kernel, evaluated so far, is a subset of the algorithm's work. Other tasks such as point cloud pre-processing and labeling the points into their respective

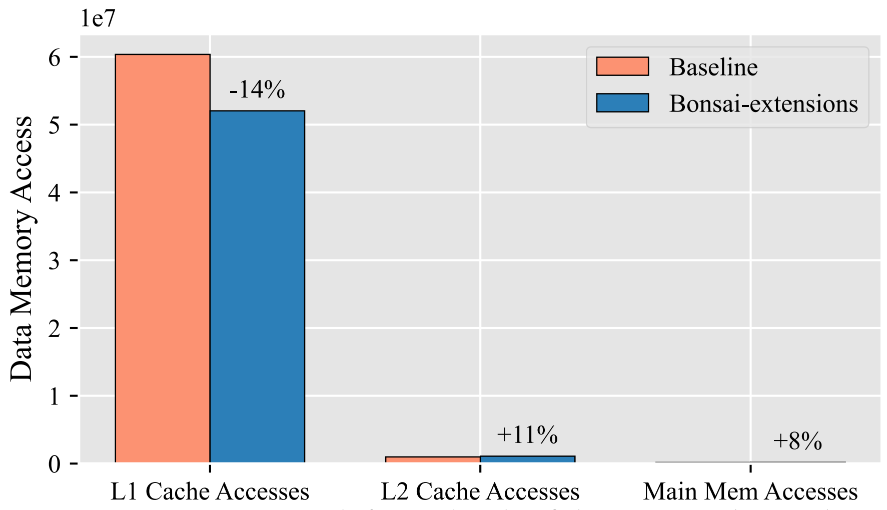


Figure 4.8 Accesses on different levels of the memory hierarchy.

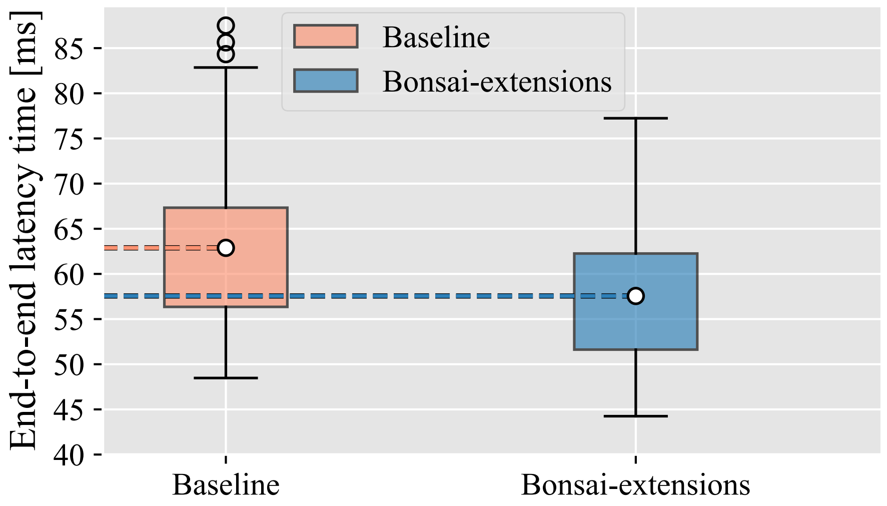


Figure 4.9 The distribution of the end-to-end latencies for the euclidean cluster algorithm. The dashed line indicates the mean value. Half the values are within the box limits.

clusters must also be performed. Figure 4.9 depicts two box plots with the distribution of the euclidean cluster end-to-end processing time for all sub-sample frames. As in any standard box plot, the boxes contain 50% of the values. We indicate the mean value of each distribution (not the median, typical of box plots) with a white circle and auxiliary dashed lines. The use of Bonsai-extensions speeds up the average end-to-end latency by 9.26%. In the context of AD, reducing the end-to-end latency translates into reducing the reaction time of the vehicle, hence actuating faster, and increasing overall safety. At this point, we recall that K-D Bonsai benefits come with the same baseline accuracy (Section 4.1.3). Also, since the euclidean cluster is generally a perception bottleneck [19, 69, 172] K-D Bonsai improvements are directly converted into overall AD improvements.

Table 4.5 Area and power for baseline CPU and K-D Bonsai.

		Area (mm^2)	Dynamic Power (W)	Static Power (W)
Processor (L2 included)		14.26	1.86	1.15
K-D Bonsai	Compression Decompression FU	0.0191	0.0095	6.29E-06
	4x (A-B')^2 FU	0.0320	0.0144	4.55E-06
	Total	0.0511	0.0240	1.08E-05
Relative change		0.36%	1.29%	0.001%

Another important aspect for AD algorithms is their end-to-end *tail latency*. Different from the average, the tail latency assesses the performance of the algorithm in situations where computation takes the most (e.g., in the *euclidean_cluster*, when point clouds have a higher number of points to be processed). K-D Bonsai again proves to be advantageous considering the 99th percentile tail latency, speeding it up by 12.19%. Hence, K-D Bonsai improves performance when it is needed the most.

4.3.3 Area and Power Analysis

Let us now examine the hardware costs of implementing our technique. Table 4.5 presents area and power overheads introduced to support K-D Bonsai, according to the methodology explained in Section 4.3.1. Overall, the hardware to support the new instructions is simple, increasing area by 0.051 mm^2, which represents an increase of 0.36% w.r.t the baseline. Likewise, supporting K-D Bonsai increases dynamic power by 24 mW (+1.29% w.r.t the baseline). These results reinforce how non-intrusive our solution is. In the context of AD, introducing minimal overheads in power and area is particularly important for meeting cooling constraints [86] and design of small autonomous vehicles (e.g., for delivery [106]), respectively.

4.3.4 Energy Analysis

Finally, we go through K-D Bonsai energy consumption results in the *extract* kernel of the *euclidean_cluster*. Figure 4.10 depicts a box plot (in the same fashion we did for end-to-end latency, Figure 4.9). The reduction in energy consumption is driven by a reduction in execution time, number of instructions, and number of memory accesses, which pays off

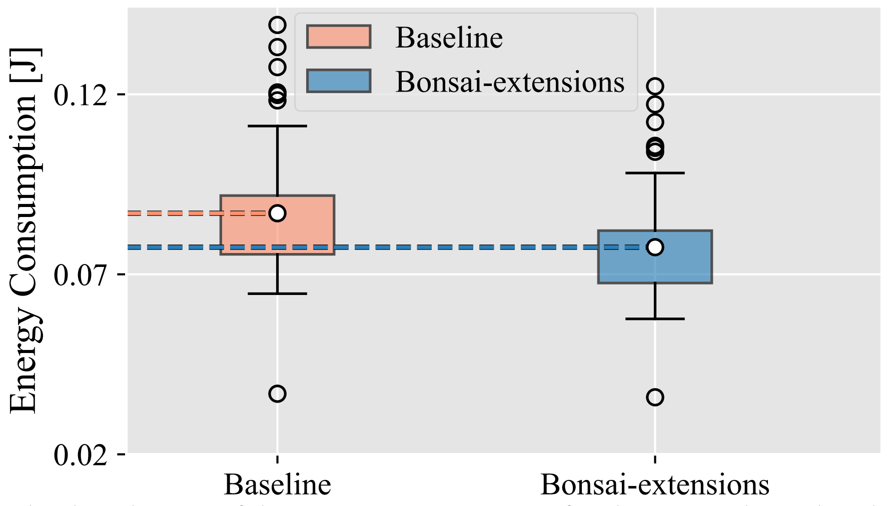


Figure 4.10 The distribution of the energy consumption for the *extract* kernel in the euclidean cluster algorithm. The dashed line indicates the mean value. Half the values are within the box limits.

the small increase in dynamic power (Table 4.5). On average, the use of Bonsai-extensions reduces energy consumption by 10.84%. K-D Bonsai successfully improves energy efficiency, which is a concern on AV so the computational platform does not reduce driving range [86, 88] (e.g., on battery-powered vehicles).

4.4 Chapter Takeaways

In this Chapter, we proposed K-D Bonsai, a novel approach to improve leaf processing during k-d tree radius search, a key operation in modern point cloud processing algorithms for AVs. K-D Bonsai reduces memory operations with a (de)compression scheme that takes advantage of value similarity and precision reduction tolerance in the points of k-d tree leaves, without harming baseline accuracy. Unlike from previous works that rely on out-of-core accelerators, K-D Bonsai is implemented in the form of ISA-extensions (Bonsai-extensions) in an OoO CPU, and validated with state-of-the-art software for AV in a the gem5 simulator in full-system mode. Our solution, K-D Bonsai, is very efficient in reducing both end-to-end latency and energy consumption while incurring minimal overheads in area and power. Besides, it requires only incremental hardware modifications on commodity CPUs while being simple to be used by the programmer (setting a flag in PCL), hence being a hands-down solution for next-generation AV systems.

5

Caravan: Maximizing SIMD Efficiency in Point Cloud Search

"We call it MMX: Media Enhancement Technology. You'll call it fun."

Intel's TV advertising on the MMX extensions, the first family of SIMD instruction in Intel processors.

In the previous Chapter, we exploited similarity in the points of the k-d tree leaves to reduce the data movement between the CPU and the memory hierarchy during *neighbor search*. However, as we explain next, we can also find similarities within the *queries* that search on the k-d tree, which we hereby exploit for improving *neighbor search*.

In this Chapter we present Caravan: a hardware (Caravan-HW) and software (Caravan-SW) co-design to exploit the similarity in k-d tree navigation among subsequent queries. Caravan packs similar queries to traverse together while reducing computing costs. Caravan first acts at the software level (Caravan-SW), gathering similar queries, leveraging the VPU, and guaranteeing correct execution even if queries face some divergence when searching on the k-d tree. Then, it acts at the hardware level (Caravan-HW) by means of two new instructions that generate dense valid indices, allowing fast operand shuffling to suppress runtime and hard-to-predict SIMD sparsity that arises from Caravan-SW.

The spatial locality (in the Euclidean space) held by consecutive LiDAR-sensed points is the cornerstone observation for our proposal. Figure 5.1 exemplifies the observation. Because LiDAR devices scan the 3D space by rotating its laser emitters [85, 153], consecutive collected points tend to be near, along the same surface. Thus, a new point that is pushed to the sensed point cloud is *likely* to have similar coordinates (x, y, z) to the previous one.

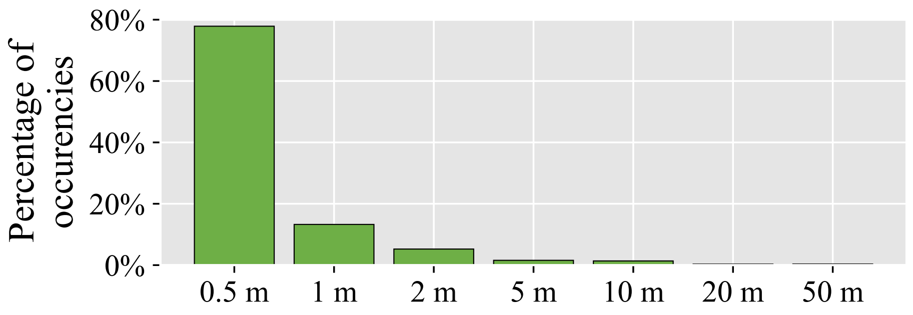


Figure 5.1 Histogram of the distance between consecutive points collected by a Velodyne LiDAR [67].

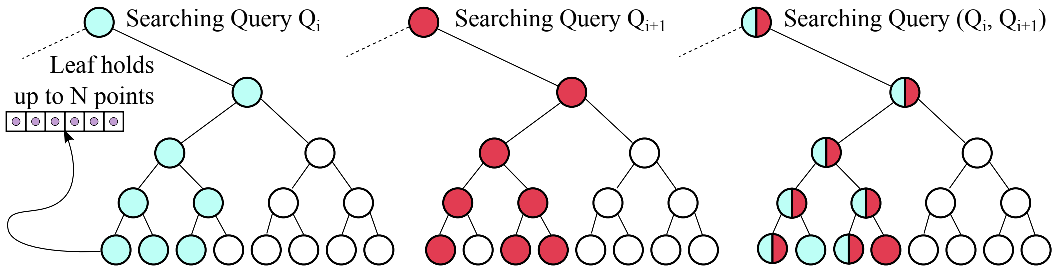

Searching Query Q_i
Searching Query Q_{i+1}
Searching Query (Q_i, Q_{i+1})
Leaf holds up to N points

Figure 5.2 Subsequent queries Q_i and Q_{i+1} share most of their visited nodes, and can traverse the k-d tree together.

This later results in similar k-d tree traversal among subsequent searches, as illustrated in the leftmost and center drawings in Figure 5.2.

Caravan-SW exploits this behavior at the application level. When consuming the list of sensed points to process, we pack consecutive points to search on the k-d tree together. Thus, the search operations that were formerly done by a single query are now performed jointly by all queries in the pack. Figure 5.2 also illustrates this concept, with two consecutive queries packed to search together. In practice, we pack more queries, up to the CPU's Vector Length (VL), to take the most out of our SIMD approach. This way, Caravan-SW avoids re-visiting nodes and re-loading k-d tree metadata, benefiting from the similarity to reduce computation costs. More importantly, this approach allows SIMD instructions to take place during *neighbor search*, drastically decreasing the instruction count. Modifying the code of the widely used PCL [117, 130] with Caravan-SW improves end-to-end latency of point cloud segmentation by 1.85×.

Despite its benefits, Caravan-SW faces runtime and hard-to-predict sparsity, which leads to sub-optimal utilization of SIMD resources and, thus, fails to achieve peak performance. The origin for runtime sparsity is two-fold. First, the number of points in the visited leaves often do not perfectly fit the CPU's VL. Forcing leaves to contain VL points would harm the balancing properties of the k-d tree, worsening search complexity [72]. Second, packed queries are similar but not equal, and thus sometimes have to follow different paths in the k-d tree. Therefore, visited nodes are not always of interest to all queries in the pack.

The deeper the search goes in the tree, the higher the chances for queries to diverge, as the subspaces get ever more restrictive, being worse in the leaf nodes. The sparsity in leaves is particularly concerning since leaf processing has extra computation compared to regular nodes of the k-d tree. Particularly, we have to compare the valid queries against the points held in that leaf. From an SIMD point of view, this means placing two structures A (e.g., the queries) and B (e.g., the points) in SIMD lanes, where each element of A has to be compared with each element of B. We call this an *all-to-all SIMD pattern.* The standard SIMD approach to solve it requires, for example, broadcasting the elements of A one by one into an auxiliary vector A'. Then, iteratively perform vector-wise operations between A' and B. Naturally, if the operations are commutative, A and B are interchangeable.

This approach works well if A and B are dense vectors. If they are sparse, however, the vector-wise operations will under-utilize the SIMD lanes. In the previous example, even though A' will be dense (with one valid element from A), B will be sparse. To overcome this, we propose Caravan-HW, two new SIMD instructions to maximize SIMD efficiency for sparse all-to-all SIMD patterns. The instructions serve to generate a dense combination of the indices of valid elements between two vectors (in this case, the packed queries and the leaf points). The combination is used to permute elements from the original vectors, shuffling them in a way that all vector lanes are utilized. This greatly improves the SIMD efficiency and reduces the number of iterations to identify neighbors in the leaves. Caravan-HW improves leaf processing and when combined with Caravan-SW pushes end-to-end performance improvements from 1.85$\times$ to 1.97$\times$.

5.1 Caravan

In this section, we detail the functioning of Caravan, our hardware and software co-design that exploits the similarity in k-d tree navigation among subsequent queries. We first discuss why subsequent queries are similar, and how we can leverage that to improve neighbor search on spatial data structures, such as k-d trees. We then explain how we change the software (Caravan-SW) to take advantage of such similarities, followed by an evaluation that identifies limitations due to sparsity. Finally, we propose hardware support (Caravan-HW) to deal with the SIMD sparsity with a programmable solution, in the form of two new instructions.

5.1.1 Exploiting Similarity in Subsequent Queries

The typical pipeline for real-time 3D vision applications includes: i) sensing the point cloud, ii) indexing it in a search-friendly data structure such as a k-d tree, iii) performing neighbor searches, and then iv) post-processing the found neighbors to increase the abstraction and infer semantics from the scene. Steps (i) and (ii) happen only once for a given point cloud frame, and step (iv) changes for each application. The neighbor search step (iii), however, not only is performed many times but is also of general use across different applications, often being their main bottleneck [22, 165]. Thus, being a key target for improvement.

Before performing neighbor searches, applications need the list of pending queries, whose neighbors are to be found. The list of pending queries depends on the algorithm. A typical case is when we need to search the neighbors of each point from the sensed point cloud independently. This happens, e.g., in the localization algorithm. In this case, the raw list of LiDAR collected points can be seen as the list of pending queries. One by one, the queries from the pending list are consumed to perform the neighbor search. Similarly, 3D CNN (see Section 3.1.2) can also search neighbors of points from the sensed point cloud in the order they come from the sensor.

In other algorithms, the list of pending queries is constructed on the fly. In segmentation, for example, we consume one point from the point cloud, find its neighbors, and set these neighbors as the list of pending queries. One by one, the queries from the pending list are consumed to perform the neighbor search, and the found neighbors are pushed to the end of the same list. This process repeats until no new neighbors are found and no pending queries exist, meaning we segmented a part of the point cloud. The process starts again with another point from the point cloud, which has not been processed yet, to segment another part of the point cloud. The algorithm finishes when all points in the point cloud have been assigned to a segment (also known as a cluster).

Regardless of which of these two methods, however, subsequent queries in the pending list are *likely* to be close in the Euclidean space. Figure 5.1 illustrates the case when we directly use the raw list of points sensed by the LiDAR. Also, we can trivially expect similarity between subsequent queries in the pending list when we search within a list of neighbors of a query, as the query's search criteria will constrain them.

Now, because of the spatial properties of k-d trees, if subsequent queries are similar in space, they are *likely* to be close in the k-d tree as well. Therefore we can expect great overlap in the visited nodes of a k-d tree among similar queries. This behavior is illustrated in Figure 5.2. We hereby refer to this as *search locality*, and it is a main observation for our proposal. Caravan exploits *search locality* to reduce computation costs and improve overall neighbor search performance. This is achieved by packing multiple similar queries together in a *query pack* ($\mathcal{Q}_{\mathcal{P}}$), and searching their neighbors in the same k-d tree inspection. The benefits of doing that are many-fold. For explanation purposes, let us assume a perfect case, where the packed queries are exactly the same. In that case, the visited nodes will also be the same, meaning that we reduce recursion costs (variables to the stack, function calls, etc.) by the number of packed queries. Also, the metadata (the splitting axis, its value, the pointers to the next node) of each visited node is read once, and applicable to all queries in the pack, instead of being re-loaded on separate searches. Moreover, the points in a leaf can be loaded once and used to be compared against all queries in the pack. Finally, since a pack can now leverage SIMD instructions, the instruction count drops proportionally to the number of packed queries.

5.1.2 Caravan-SW

Below we explain how to convert the observed *search locality* benefits into Caravan-SW, a practical software implementation. We derive our ideas upon the neighbor search implementation in the PCL, the most popular open-source library [130] for point cloud processing, employed as the baseline software for recent related works as well (e.g., [22, 35, 165, 166]).

Caravan-SW extends the PCL, adding code to perform the search with multiple queries simultaneously. The user determines whether to use such a feature, by calling the appropriate overloaded methods. While the baseline PCL neighbor search requires only one query point, Caravan-SW requires a $\mathcal{Q}_{\mathcal{P}}$. In C++ terms, a $\mathcal{Q}_{\mathcal{P}}$ is as simple as an array of queries. The pack size is defined at compile-time in our implementation, up to a maximum of the CPU's VL size, as we want to leverage SIMD instructions. For the latest Intel processors (the case study for our work), users can pack up to 16 points, which underneath are processed with AVX512 [64]. A similar implementation can be achieved with other ISAs, e.g., Arm's SVE [141].

The main changes proposed by Caravan-SW happen inside the library, thus being transparent for the final user. Particularly, the baseline single-query recursion (Section 3.1.3) needs to be adapted. The first goal is to visit the same set of nodes we would with multiple baseline searches, so the functionality is maintained. Additionally, we want to visit each necessary node only once, to leverage the benefits of *search similarity* (e.g., metadata reuse). Formally, let $\mathcal{Q}_{\mathcal{P}}$ be the query pack with queries $\{q_1, q_2, ..q_n\}$, and $\mathcal{N}(q)$ the nodes visited by q during its neighbor search. We want $\mathcal{N}(\mathcal{Q}_{\mathcal{P}}) = \mathcal{N}(q_1) \cup \mathcal{N}(q_2) \cup \ \ldots\ \mathcal{N}(q_n)$. If we are not careful in the design of Caravan-SW, however, this requirement may not happen. Due to divergence (i.e. the need of queries in the $\mathcal{Q}_{\mathcal{P}}$ to take different directions in the k-d tree), naive solutions could either visit unnecessary nodes, or re-visit nodes multiple

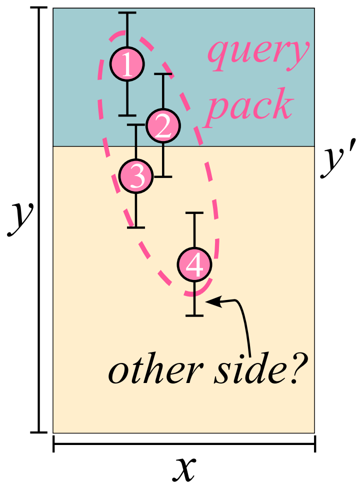


Figure 5.3 Packed queries can disagree on the next sub-tree to visit.

times. Following, we explain mechanisms to ensure that we only visit necessary nodes, and exactly one time.

To exemplify, let us assume that a $\mathcal{Q}_{\mathcal{P}}$ with four queries $\{q_1, q_2, q_3, q_4\}$ have been descending the k-d tree with no divergence, until it reaches a node at k-d tree level L, as shown in Figure 5.3. Level L splits the space in the y coordinate (at value y'). On each query in the figure, we also indicate a distance range that queries take into account to decide whether they have to visit the other side (see Section 3.1.3). By inspecting the figure, we can infer that q_1 needs to visit the *left* sub-tree, q_2 and q_3 have to visit both, and q_4 needs to visit the *right* sub-tree. Therefore, there is no consensus on what is the next sub-tree to visit, which we call *search divergence*. Still, we have to visit both sides to ensure functionality, even though each visited sub-tree will be relevant to a portion of the $\mathcal{Q}_{\mathcal{P}}$.

The order in which these visits happen will be explained soon, and for now let us assume that we start visiting the left sub-tree, attending q_1, q_2, and q_3. From this left sub-tree (at level $L + 1$) and below, q_4 is known to be irrelevant. Still, it belongs to the $\mathcal{Q}_{\mathcal{P}}$, which is part of the recursion arguments. Therefore, we need a way of ignoring q_4 when deciding the next sub-tree to visit (from level $L + 1$ to level $L + 2$), otherwise, it might diverge again from queries q_1, q_2, and q_3, forcing the algorithm to descend into an unnecessary sub-tree (until reaching a leaf, as explained in Section 3.1.3). This can happen in any subsequent level, even though q_4 is known to be irrelevant since the left sub-tree was taken in level L.

For this reason, we enhance the recursion with an extra ***valid mask*** parameter. It contains one bit per element in the query pack, indicating the *valid queries* (e.g., q_1, q_2, and q_3 in the example), and the *invalid queries* (e.g., q_4 in the example). The lack of such a ***valid mask*** could make the number of visited nodes $|\mathcal{N}(\mathcal{Q}_{\mathcal{P}})|$ explode to a number much

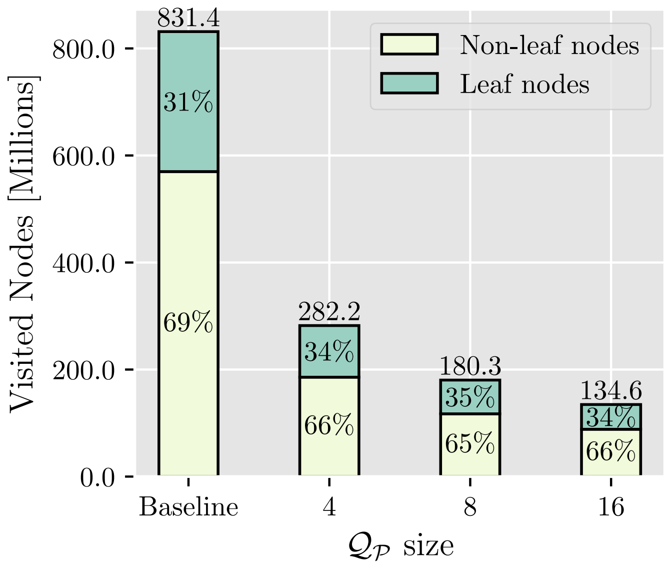


Figure 5.4 The number of visited nodes for the neighbor search operation in a segmentation algorithm. Caravan-SW successfully exploits *search locality* to avoid visiting redundant nodes.

higher than the multiple baseline searches, $|\mathcal{N}(\mathcal{Q}_\mathcal{P})| > |\mathcal{N}(q_1)| + |\mathcal{N}(q_2)| + \ \dots \ + |\mathcal{N}(q_n)|$, harming Caravan-SW potential.

Also, when queries diverge we need to re-think the way of visiting sub-trees. Let us again use Figure 5.3 to exemplify. If we directly apply the decision policy explained in Section 3.1.3, q_2 would visit the left sub-tree first, and q_3 would visit the right sub-tree first. When unwinding the recursion, they would find they have to go to the other side as well, causing the search to descend twice on each sub-tree. This again could cause $|\mathcal{N}(\mathcal{Q}_\mathcal{P})| > |\mathcal{N}(q_1)| + |\mathcal{N}(q_2)| + \ \dots \ + |\mathcal{N}(q_n)|$.

In Caravan-SW we first find the distance of each query against the splitting coordinate. With this, we know what is the best sub-tree to visit for any of them. Also, we anticipate the calculations to check if going to the other sub-tree is necessary, instead of waiting for the recursion to unwind and thus avoiding the problem of re-visiting a node. With this, we can group queries that have to visit the left sub-tree and the queries that have to visit the right sub-tree. At the same time, we can update the recursion parameters depending if they are visiting the best sub-tree or the other sub-tree (as the calculations are different). At the implementation level, we do this with the help of a ***pivot query***. The pivot query is simply one valid element from the $\mathcal{Q}_\mathcal{P}$. In our implementation, we find the pivot query with a bit-scan along the valid mask, choosing the first valid query we can find. We use this pivot query to create an order when visiting the sub-queries. First, we visit the best sub-tree for

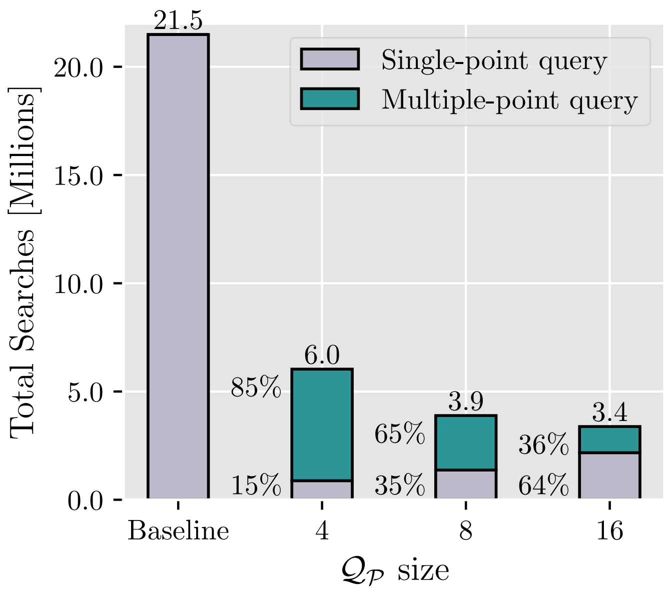


Figure 5.5 Total neighbor searches performed, with a category breakdown in the segmentation task with Caravan-SW.

the pivot query and all other queries that also have to visit that sub-tree (setting the valid mask accordingly). And later, we visit the other sub-tree (also adjusting the valid mask), if any queries need to visit it. The pivot query will generally agree with others, thus being an efficient approach regardless of whether divergence happens or not.

Finally, to avoid loops when calculating the decision of each query, we leverage the VPU of the CPUs. With this, we broadcast the values that are common to all queries (for example, the position of the splitting coordinate) and do vector-wise calculations at once, greatly reducing the instruction count. Likewise, when processing the leaves, SIMD instructions can also take place to compute the distances.

5.1.3 Implications of Caravan-SW

We now take a brief moment to inspect the consequences of Caravan-SW, to understand the behavior of the modified code. Particularly, we are interested in verifying if our hypothesis of *search locality* holds when used in practice with real-life data (details in Section 5.2.1). Apart from that, we also want to identify possible points of improvement.

The major goal of grouping queries in a pack to perform the neighbor search is to avoid visiting redundant nodes, and therefore reduce the search costs. Figure 5.4 depicts the number of visited nodes for neighbor searches performed during a segmentation task. The figure shows data for the baseline PCL code and Caravan-SW with different fixed

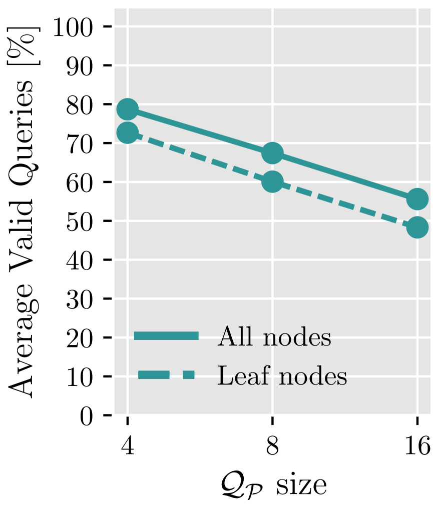


Figure 5.6 The percentage of valid queries on Caravan-SW with varying $\mathcal{Q}_\mathcal{P}$sizes in a segmentation task.

$\mathcal{Q}_\mathcal{P}$ sizes, whenever there are enough pending queries available. It also breaks down the visited nodes into leaf and non-leaf nodes, which is relevant because leaf processing adds extra computation as explained in Section 3.1.3.

The first observation is that Caravan-SW effectively reduces the number of visited nodes. For example, from the baseline code to Caravan-SW with a 4 queries in the $\mathcal{Q}_\mathcal{P}$, we see a 66% decrease in visited nodes. This reduction directly converts into fewer operations to perform the neighbor search, such as instructions to manage the stack for the recursion, and to load metadata in the nodes including points stored in the leaves. The reduction reaches a maximum of 83% for Caravan-SW with a $\mathcal{Q}_\mathcal{P}$ of size 16, compared to the baseline. Therefore, our *search locality* hypothesis has proven true. More importantly, all the benefits so far come only from data pattern observation and appropriate algorithm adjustments.

Although the benefits are good, it is relevant to look at how the idea scales. Since the VL size of CPU vector units has been increasing in the last decade [64, 141], new solutions should make the best use of such a size increase, to pay off the hardware cost they require. If we look at the number of visited nodes and how they vary as we double the elements in the $\mathcal{Q}_\mathcal{P}$ (from 4 to 8, and then from 8 to 16), we do not see a perfectly proportional reduction. From a $\mathcal{Q}_\mathcal{P}$ of 4 queries to 8, the reduction is 36%, and from 8 to 16, 25%.

To understand this behavior, it is necessary to look deeper into the application. First, we remind the reader that the segmentation searches incrementally, fetching queries from a previously found list of neighbor points, as explained in Section 5.1.1. Not always this list will have enough queries to fill up the $\mathcal{Q}_\mathcal{P}$. This is shown in Figure 5.5 with a breakdown of how many searches were performed with Caravan-SW (multiple queries), and how many with baseline (single query). The more queries in the $\mathcal{Q}_{\mathcal{P}}$, the fewer times we employ Caravan-SW. We still reduce the total number of searches because each search contains more queries, but certainly, this limits Caravan-SW benefits. In practice (Section 5.2), we set a threshold of minimum queries, and allow the size of the $\mathcal{Q}_{\mathcal{P}}$ to vary from this threshold up to the VPU's VL.

Another concern is the under-utilization of the VPU lanes, caused by *search divergence*. As we increase the size of $\mathcal{Q}_{\mathcal{P}}$, we reduce the probability of all queries descending into the same sub-tree. Figure 5.6 quantifies this behavior, indicating the average number of valid queries we have in the $\mathcal{Q}_{\mathcal{P}}$ throughout neighbor searches. While a $\mathcal{Q}_{\mathcal{P}}$with 4 queries has 79% of valid queries on average, it decays to 67% with 8 queries, and 55% with 16 queries. This can compromise the benefits of Caravan-SW (the Turquoise bar in Figure 5.5). In the worst case, if we have only $\frac{1}{\mathcal{Q}_{\mathcal{P}}\ size}$ valid queries per visited node, it means we have to visit the same number of nodes as $\mathcal{Q}_{\mathcal{P}}$ size baseline searches, but with Caravan-SW overheads (e.g., bookkeeping the valid mask). Although the found percentages are far from this catastrophic example, they are still relevant in understanding how Caravan-SW benefits scale with the number of queries.

We also indicate the average valid queries for the leaf nodes (dashed line) in Figure 5.6. The percentage is worse than the overall, because as previously explained, the deeper the search goes, the more restrictive the subspace is, causing even small coordinate differences between queries in the $\mathcal{Q}_{\mathcal{P}}$ to diverge in the next sub-tree decision. Note that divergence in the leaves is particularly concerning due to extra computations for leaf processing, i.e., comparing the Euclidean distance between the queries and the points, and pushing the found neighbors to the *list of neighbors* of the corresponding query. Therefore, although Caravan-SW is capable of reducing visited nodes on its own, it is under-utilizing the SIMD hardware available in CPUs, which prevents it from reaching maximum benefits.

5.1.4 Caravan-HW

The sparsity in Caravan-SW threatens the efficiency of the SIMD operations executing in the VPU. In this context, the tree leaves are the most affected, as they face the worst sparsity (Figure 5.6) while also executing a higher number of SIMD operations compared to regular nodes. Therefore, suppressing the sparsity for *leaf processing* is crucial to avoid SIMD drawbacks in Caravan-SW.

Algorithm 1 provides a high-level description of the leaf processing steps with a traditional CPU. We iterate on each query from the $\mathcal{Q}_{\mathcal{P}}$, and check if they are valid. If they are, we broadcast the query into a vector (replicating the query in all SIMD lanes) to perform

Algorithm 1 SIMD Leaf Processing

```
Require: $\mathcal{Q}_\mathcal{P}$, leaf_points, valid_mask, neighbors_list
1: for idx, $q$ in enumerate($\mathcal{Q}_\mathcal{P}$) do
2:     if !(valid_mask[idx]) then
3:         continue
4:     end if
5:     q_vec ← broadcast($q$)
6:     dist_vec ← simd_euclidean_dist(q_vec, leaf_points)
7:     neighbor_mask ← simd_less_than(dist_vec, MAX_DIST)
8:     neighbors_list[idx].simd_append(leaf_points, neighbor_mask)
9: end for
```

SIMD Euclidean distance against the `leaf_points` held by the leaf. Although abstracted in the algorithm, the Euclidean distance computes the summation of the square of the differences between the query and the points in coordinates x, y, z. This is similar to the definition in Equation 3.1, but performed vector-wise. Also, comparisons are performed with the square distance, to avoid performing the costly square root. Points fitting the neighborhood criteria (e.g., are closer than the `MAX_DIST`) are vectorially stored in the list of neighbors of the query.

In the example of Algorithm 1, the broadcasted queries fill the `q_vec`, but still, there is SIMD sparsity since the `leaf_points` vector does not necessarily have VL elements, and instead, vary depending on how the subspaces are divided during the k-d tree construction (see Figure 3.5). Forcing the leaves to hold exactly VL points would harm the tree balance, which is not desired as it harms the search complexity [72]. Alternatively, we could swap the variable roles, and iterate on each point from the `leaf_points`, broadcast them into a vector, and do SIMD operations against the $\mathcal{Q}_\mathcal{P}$. However, sparsity would also appear in this case, since queries in $\mathcal{Q}_\mathcal{P}$ diverge, as shown in Figure 5.6.

Therefore, regardless of the variable we choose to iterate and broadcast (either queries or points), *leaf processing* will underutilize the VPU. The left and center columns of Figure 5.7 depict this behavior. We have two vectors A and B that need to perform SIMD all-to-all operations, as in the *leaf processing*. In the example figure, the valid elements of vector A are $\{0, 1, 3, 6, 7\}$, while the valid elements of vector B are $\{0, 1, 2\}$.

From the figure, we can see that it is best to iterate and broadcast on the sparser vector (the Broadcast B center column in Figure 5.7) while maintaining the denser vector fixed. This reduces the number of steps and increases the SIMD utilization to perform all-to-all comparisons with respect to the alternative (Broadcast A option). However, hard-coding this pattern in the algorithm is a pitfall, as the best vector to iterate and broadcast will vary. Figure 5.8 shows a comparison of valid queries versus points during leaf processing for

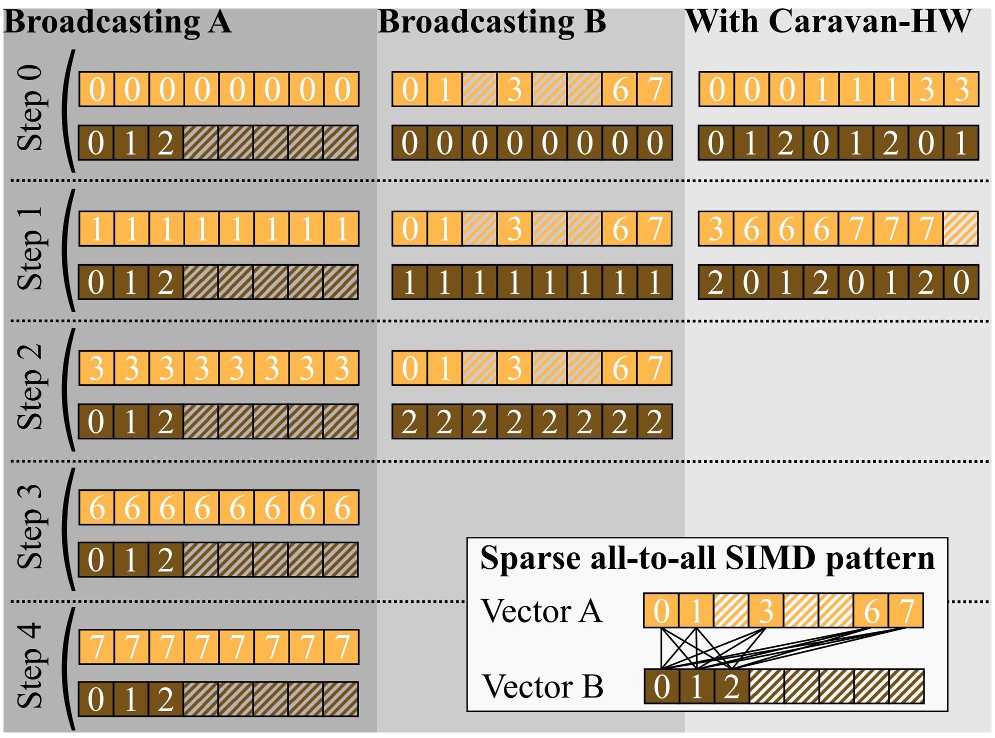


Figure 5.7 The number of steps to perform all-to-all comparisons between two sparse vectors. For a traditional CPU, it is necessary to choose one vector to iterate and broadcast, while fixing the other one, leading to VPU underutilization. With Caravan-HW support, vectors can be shuffled to fully occupy the SIMD lanes, increasing VPU efficiency.

our experimentation (details in Section 5.2.1). Sometimes we have fewer valid queries than points, sometimes the opposite. This will depend on the $\mathcal{Q}_{\mathcal{P}}$ we have and the leaves they visit. In any case, whenever a lane from either A or B is not valid, we are underutilizing the VPU capabilities. In both cases (Broadcasting A or Broadcasting B) we have some degree of underutilization if A and B are sparse. So even if we choose the best-suited variable to iterate, we would still not leverage the full capabilities of the VPU.

We propose Caravan-HW to provide hardware support to *densify* SIMD vectors, and maximize VPU utilization. The goal of Caravan-HW is to allow the programmer to fill

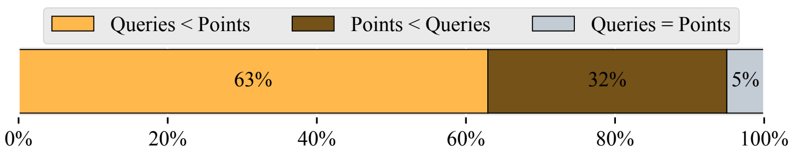


Figure 5.8 The leaf node sparsity varies at runtime. Sometimes, we have less valid queries than valid points, sometimes the opposite. Results for a *Min* $\mathcal{Q}_{\mathcal{P}}$*size* of 16 queries.

sparse lanes with valid elements from the sparse vectors. The right column of Figure 5.7 illustrates the idea. To achieve this, we need to re-arrange the valid elements of the vectors, so that lanes have maximal effective computation. In other words, we need to place the pairs of valid elements from A and B consecutively, without invalid values in between.

In our example, we would like to compute the pairs $\{(A_0, B_0)$, (A_0, B_1), (A_0, B_2), (A_1, B_0), (A_1, B_1), (A_1, B_2), (A_3, B_0), (A_3, B_1), (A_3, B_2), (A_6, B_0), (A_6, B_1), (A_6, B_2), (A_7, B_0), (A_7, B_1), $(A_7, B_2)\}$. As shown in Figure 5.7, these pairs would then be split in chunks of size VL, so it could be executed by the VPU, in two steps for a $VL = 8$ as shown in the figure. Notice that for a $VL = 16$ all computations could be done in one step for this example. Modern ISAs, however, do not provide an instruction to arrange dense pairs from sparse vectors. Instead, the dense vectors have to be generated with multiple instructions. While this can be a solution if the same vector lane configuration is used for many SIMD instructions, it will take too much time for smaller code snippets, such as a *leaf processing* iteration.

Caravan-HW proposes a pair of instructions to solve this: **Extract Dense IDs Repeating Sequence (EDIRS)**, and **Extract Dense IDs Repeating Element (EDIRE)**. Algorithms 2 and 3 detail their behavior. By looking at the desired pairs, we can see that one vector needs to consecutively repeat the sequence of valid indices (vector B in the example, with B_0, B_1, B_2, B_0, $\ldots$), while the other (vector A in the example) repeats one element at a time (e.g., A_0, A_0, A_0, A_1, $\ldots$), for as many times as the size of the other's sequence.

Algorithm 2 Extract Dense IDs Repeating Sequence

```
Require: valid_mask, step, seq_size
Ensure: dense_sequence_ids
1: indices ← [0 .. VL]
2: compressed_indices ← broadcast(−1)
3: compressed_indices ← compress(indices, valid_mask)
4: for i in range(VL) do
5:     src_idx ← ((step × VL) + i) mod seq_size
6:     dense_sequence_ids[i] ← compressed_indices[src_idx]
7: end for
8: return dense_sequence_ids
```

To perform the ***EDIRS*** and ***EDIRE*** instructions, we need the mask with valid elements and the size of the repeating sequence. In the context of Caravan-SW and point cloud neighbor search, we have the ***valid mask*** at hand for the $\mathcal{Q}_{\mathcal{P}}$ (see Section 5.1.2), and we can easily build a mask for the points, using the available information of how many points there are in the leaf. Also, since the list of pairs can be bigger than VL, we need to inform

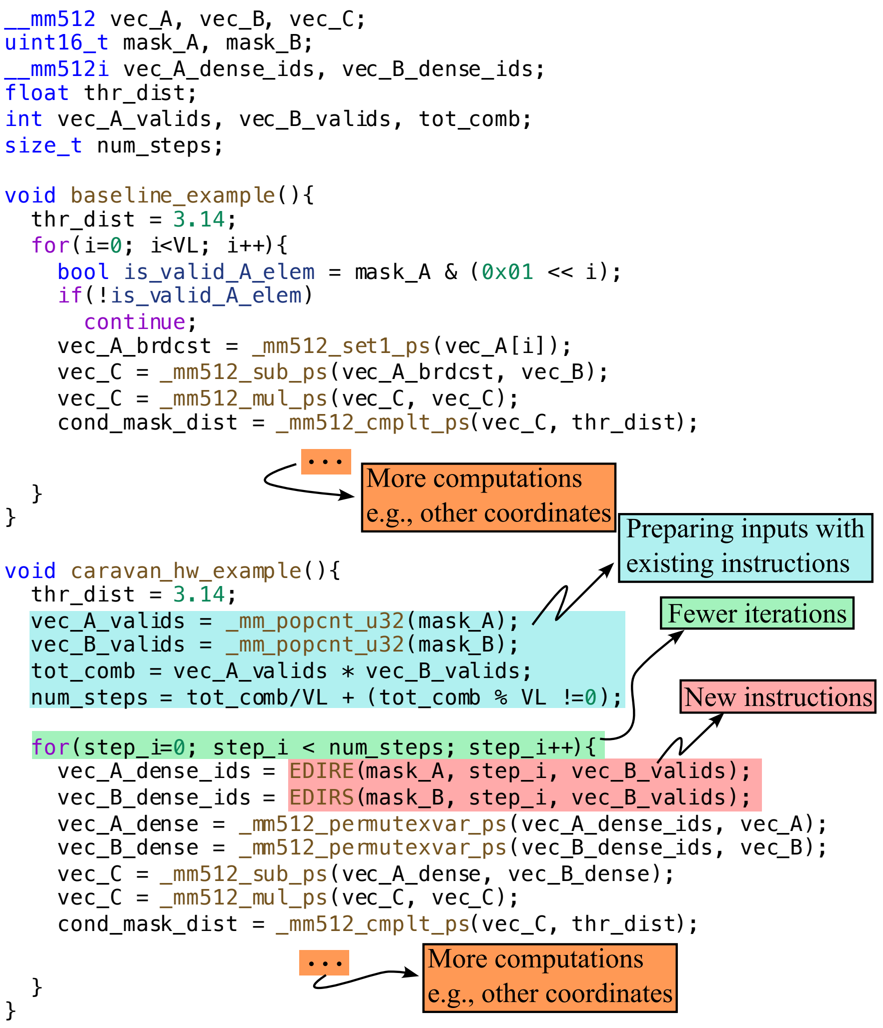
Algorithm 3 Extract Dense IDs Repeating Element

```
Require: valid_mask, step, seq_size
Ensure: dense_elements_ids
1: indices ← [0 ..  VL]
2: compressed_indices ← broadcast(−1)
3: compressed_indices ← compress(indices, valid_mask)
4: for i in range(VL) do
5:     src_idx ← ((step × VL) + i) ÷ seq_size
6:     dense_sequence_ids[i] ← compressed_indices[src_idx]
7: end for
8: return dense_elements_ids
```

the instruction which iteration `step` we are to generate a corresponding *window* of the list of pairs with size VL.

The list of indexes to process a given iteration step is generated by the instructions, according to algorithms 2 and 3. As from the algorithms, each output is independent from the others. Therefore, we can compute all outputs together, outputting a SIMD vector of indices. For each output element, we multiply the `step` value with the VL. Then we divide (***EDIRE***) by the sequence size or compute the modulo (***EDIRS***) to obtain the element for the corresponding position in that step. A functional unit for each instruction can be directly derived from Algorithms 2 and 3, each requiring three integer source registers, and one SIMD destination register. For the hardware implementation, multiplying by VL can be a simple shift, since VL is typically a power of 2. Also, since the `steps` has a maximum value of 16, dividers are 5-bits only. This allows the hardware implementation of the two instructions to be cheap, as we show in the Results, Section 5.2.

Therefore, both instructions generate a list of indices from one of the sparse vectors. One repeats the whole sequence, while the other repeats the elements by the former's sequence size. When combined, they act as the desired list of valid pairs. We synergically use the list of indexes with existing instructions in modern ISAs. Figure 5.9 exemplifies with a simplified code snippet. For example, in x86, the *permutexvar* class of intrinsic instructions [64] (Figure 5.10) is able to shuffle the elements of an input vector, placing them in an output vector according to a third vector that indicates the source position for each output element. The *permutexvar* instructions are generally used for shuffling the input vector with a shuffling pattern known at compile-time. For Caravan-SW, however, this is not enough since the $\mathcal{Q}_\mathcal{P}$ will have a different number of valid elements (and their position within the pack) potentially varying in every visited leaf, which in turn will also have a different number of points. Nevertheless, Caravan-HW can overcome this unpredictable behavior, quickly generating a dense list of indices to be used at runtime.

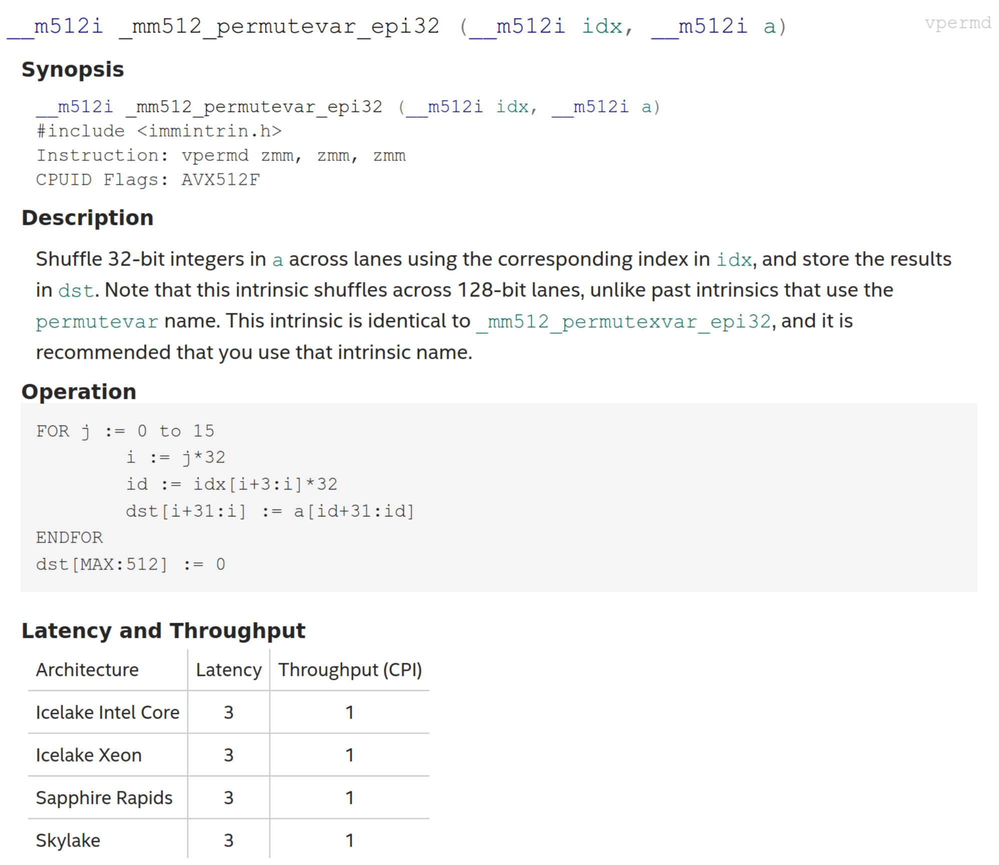


```
__mm512 vec_A, vec_B, vec_C;
uint16_t mask_A, mask_B;
__mm512i vec_A_dense_ids, vec_B_dense_ids;
float thr_dist;
int vec_A_valids, vec_B_valids, tot_comb;
size_t num_steps;

void baseline_example(){
  thr_dist = 3.14;
  for(i=0; i<VL; i++){
    bool is_valid_A_elem = mask_A & (0x01 << i);
    if(!is_valid_A_elem)
      continue;
    vec_A_brdcst = _mm512_set1_ps(vec_A[i]);
    vec_C = _mm512_sub_ps(vec_A_brdcst, vec_B);
    vec_C = _mm512_mul_ps(vec_C, vec_C);
    cond_mask_dist = _mm512_cmplt_ps(vec_C, thr_dist);

                    ...
  }
}

void caravan_hw_example(){
  thr_dist = 3.14;
  vec_A_valids = _mm_popcnt_u32(mask_A);
  vec_B_valids = _mm_popcnt_u32(mask_B);
  tot_comb = vec_A_valids * vec_B_valids;
  num_steps = tot_comb/VL + (tot_comb % VL !=0);

  for(step_i=0; step_i < num_steps; step_i++){
    vec_A_dense_ids = EDIRE(mask_A, step_i, vec_B_valids);
    vec_B_dense_ids = EDIRS(mask_B, step_i, vec_B_valids);
    vec_A_dense = _mm512_permutexvar_ps(vec_A_dense_ids, vec_A);
    vec_B_dense = _mm512_permutexvar_ps(vec_B_dense_ids, vec_B);
    vec_C = _mm512_sub_ps(vec_A_dense, vec_B_dense);
    vec_C = _mm512_mul_ps(vec_C, vec_C);
    cond_mask_dist = _mm512_cmplt_ps(vec_C, thr_dist);

                    ...
  }
}
```

More computations e.g., other coordinates
Preparing inputs with existing instructions
Fewer iterations
New instructions
More computations e.g., other coordinates

Figure 5.9 Example code on how to use EDIRE and EDIRS instructions along with existing x86 instructions. In the example, two sparse vectors A and B compare their elements all-to-all to check which are closer than the `thr_dist`. The Caravan-HW instructions reduce the loop iterations.

In summary, we use Caravan-HW instructions to generate the list of indices, and then we use existing instructions to shuffle the elements according to the lists. For the distance computation during leaf processing, for example, we can use the same shuffling pattern (indices list) to assign the different coordinates of points and queries to the same lanes. On each iteration step, we can perform the different operations for distance calculation in each coordinate, and accumulate in the end. Then, we can retrieve the respective queries and points accounted for in a lane with an indirection to the list of indices. For example, in Figure 5.7, the computation with Caravan-HW at lane 7 and step 0, refers to (A_3, B_1). If A and B are queries and points, respectively, we could append point$_1$ to the list neighbors of query$_3$ if the calculated distance indicates so.



```
__m512i _mm512_permutevar_epi32 (__m512i idx, __m512i a)          vpermd
```

Synopsis

```
__m512i _mm512_permutevar_epi32 (__m512i idx, __m512i a)
#include <immintrin.h>
Instruction: vpermd zmm, zmm, zmm
CPUID Flags: AVX512F
```

Description

Shuffle 32-bit integers in `a` across lanes using the corresponding index in `idx`, and store the results in `dst`. Note that this intrinsic shuffles across 128-bit lanes, unlike past intrinsics that use the `permutevar` name. This intrinsic is identical to `_mm512_permutexvar_epi32`, and it is recommended that you use that intrinsic name.

Operation

```
FOR j := 0 to 15
        i := j*32
        id := idx[i+3:i]*32
        dst[i+31:i] := a[id+31:id]
ENDFOR
dst[MAX:512] := 0
```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Icelake Intel Core	3	1
Icelake Xeon	3	1
Sapphire Rapids	3	1
Skylake	3	1

Figure 5.10 The *permutexvar* instruction from Intel Intrinsics [64] that, in our case, consume the indices vectors generated by *EDIRE* and *EDIRS* instructions to arrange valid elements from sparse vectors into new vectors to maximize SIMD efficiency. Adapted from [64].

5.2 Results

In this section, we explain the evaluation methodology and associated results for Caravan.

5.2.1 Evaluation Methodology

We explain our methodology in parts. First, we discuss how we implement Caravan-SW and Caravan-HW. Then we detail how we evaluate Caravan benefits in a state-of-the-art AV software, and the computing platform used.

Caravan-SW. We use the PCL [117, 130] version 1.10 and its auxiliary library FLANN [100, 101] version 1.9.2 as our baseline software for performing *neighbor search*. These libraries are widely used for point cloud processing and serve as baseline software for recent related works as well (e.g., [22, 35, 165, 166]). We implement the Caravan-SW relying on AVX512 from Intel Intrinsics [64]. The internal implementation is abstracted from the user of the

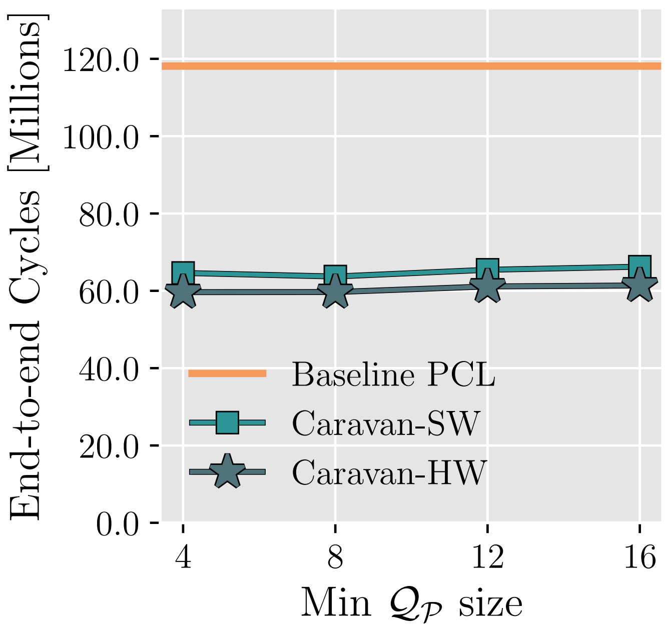


Figure 5.11 End-to-end speedup from Caravan-SW and Caravan-HW for Autoware's segmentation.

library, who only has to provide a vector of queries (the $\mathcal{Q}_{\mathcal{P}}$) and a vector of *list of neighbors* (where the found neighbors will be pushed). During leaf processing, we choose to broadcast and iterate on queries, calculating their distance to multiple points at once per iteration. This allows multiple points to be pushed to the query's *list of neighbors* with a vector store. If we iterate and broadcast points, we would find the same neighbors but would have to perform more stores. We compile the library with the `-O3` flags.

Caravan-HW. We perform software emulation to implement the behavior of the Caravan-HW instructions ***EDIRS*** and ***EDIRE***, through multiple SIMD instructions. We account for data preparation (e.g., mask generation and vector initialization) and adapt the software to use the generated indices with Intel's *permutxvar* instructions. When in a leaf, we pre-generate the indices for each step and store them in an auxiliary software structure. We measure the time to generate the indices following Intel's manual on accurate cycle measurement for Intel CPUs [112]. Later, to perform the leaf processing, we fetch the pre-generated indices for each iteration step with two SIMD loads, that behave as ***EDIRS*** and ***EDIRE*** instructions. In the end, we substitute the software emulation costs (to pre-generate the indices) by the latency obtained with synthesis (Section 5.3) to have the final estimation of the execution cycles of the application with Caravan-HW. Since our instructions are well-behaved and predictable (e.g., no memory references or control involved) we believe this methodology is fair. It also allows us to run real-life applications, instead of micro-

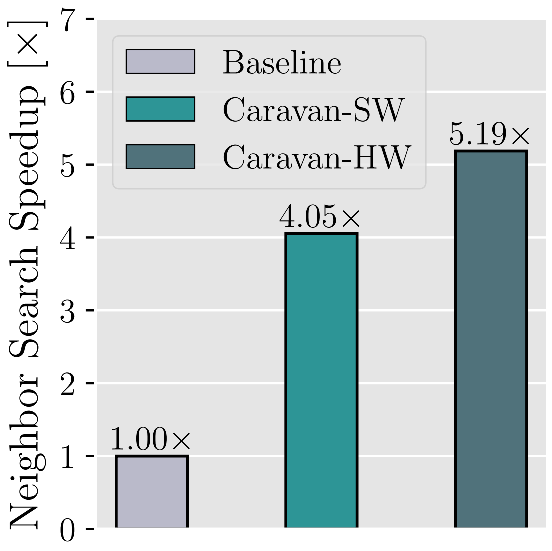


Figure 5.12 Speedup from Caravan-SW and Caravan-HW considering only the neighbor search.

benchmarks or fractions of applications typically employed on CPU simulators. Also, since we use a commercial CPU, we prevent errors from assuming CPU parameters which is sometimes needed in simulators as well. To estimate instruction latency, area, and power, we describe the hardware of both instructions with Verilog, considering a VPU unit with 16x32-bit lanes (as in AVX512). The approach is similar to the one described in Section 4.3.1. The behavior is verified functionally, with Icarus Verilog [161], and the hardware is synthesized with Design Compiler [41], using a 14 nm node technology library [95], the same technology node of Intel Xeon W2155 used for our experiments [65].

Application and Dataset. We evaluate Caravan potential with Autoware.ai [13, 68], an open-source software for AD. Autoware.ai provides a complete set of algorithms to perform AD, from sensor processing and perception to actuation. In this work, we test Caravan implementation in Autoware's point cloud segmentation module [129]. The segmentation module is responsible for clustering points, separating the point cloud into labeled objects. Later, Autoware consumes the segmented objects, to either extract their size and distance, or to perform 3D object classification. Notably, point cloud segmentation yields the highest latency across typical point cloud perception modules employed in AD [19, 69, 172], thus being a timely and relevant use-case to experiment Caravan on.

We stimulate Autoware with an eight-minute LiDAR sequence, collected with a vehicle with a sampling frequency of 10 Hz and released by Autoware [67]. In Autoware, the point cloud is pre-processed before the Segmentation. Ground points are removed, and

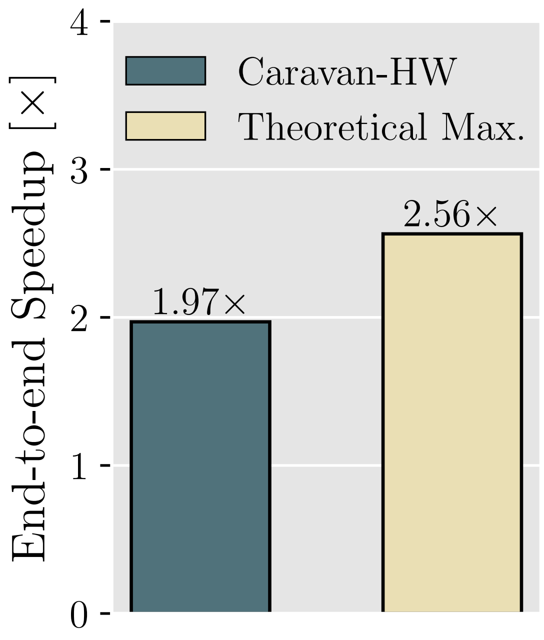


Figure 5.13 Caravan-HW end-to-end speedup is close to the Theoretical Maximum.

the remaining points are down-sampled with a voxel grid filter [117]. We compile the code with `gcc/g++` version 13.1 and run it on an Intel Xeon W-2155 fixing the frequency at 2.5GHz to reduce measurement variation, with 256 GB, and Ubuntu 20.04. The Xeon supports AVX512, thus being able to perform up to 16 `float` operations in parallel (i.e., VL=16).

5.3 Synthesis Analysis

While Caravan-SW has zero hardware cost (completely software approach), Caravan-HW does require additional hardware for the new instructions. Table 5.1 presents synthesis results for the functional units for ***EDIRE*** and ***EDIRS*** (see Methodology details in Section 5.2.1). Caravan-HW instructions incur an almost negligible area cost (0.03175 mm^2 per core), even when accounted for in the 10 cores available in the Intel Xeon W-2155 (0.31747mm^2). Comparatively, even though Intel does not report area per core for the Xeon Models, it does report an area of $45\ mm \times 52.5\ mm = 2362.5\ mm^2$ for the whole package [65]. While we acknowledge that the area for the cores is a fraction of the package area, showing how little extra area is required to support Caravan-HW. Likewise, intel does not provide a breakdown of power consumption per core or functional units. However, we know from the measurements in Section 2.3.2, that an intel i7 core running Autoware

Table 5.1 Synthesis data for area and power required by the instructions added with Caravan-HW.

	Area (mm^2)	Dynamic Power (W)	Static Power (W)	Latency @2.5GHz (cycles)
EDIRE	0.01570	0.0012392	9.18E-10	9
EDIRS	0.01604	0.0011936	1.07E-09	9
Total per Core	0.03175	0.0024328	1.99E-09	-
Total for 10 Cores	0.31747	24.328	1.99E-08	-

tasks would require around 40 W (see Table 2.4). On the other hand, Caravan-HW adds only 2.43 mW dynamic per core, i.e., four orders of magnitudes lower. In general, since our overheads are constrained inside specific new functional units, we can leverage most of the baseline CPU infrastructure (e.g., the VPU registers), to provide performance benefits at low cost in area and power. Finally, latency-wise the new SIMD instructions are also in pair with existing Intel Intrinsics [64], each taking 9 cycles to execute.

5.3.1 Performance Analysis

In Section 5.1, we set different fixed $\mathcal{Q}_{\mathcal{P}}$sizes to show how they reduce the number of visited nodes (Figure 5.4). The larger the $\mathcal{Q}_{\mathcal{P}}$, the higher the reduction. However, increasing the $\mathcal{Q}_{\mathcal{P}}$ size reduces the chances of using Caravan (Figure 5.5) because the list of pending queries for segmentation might not have enough elements to fill the $\mathcal{Q}_{\mathcal{P}}$. Also, more queries increase the divergence likelihood (Figure 5.6).

Thus, to maximize the benefits of Caravan, we allow the search to occur with a variable number of elements, increasing the chances of using our SIMD approach. However, allowing SIMD searches with too few elements in the $\mathcal{Q}_{\mathcal{P}}$ can harm Caravan's benefits. For example, if there is only one query pending to be searched, it is naturally better to search with the baseline PCL code. Thus, to use Caravan one must account for the overheads of initializing vectors and the heavier recursion costs that come with multiple-query variables. We thus use a threshold *Min $\mathcal{Q}_{\mathcal{P}}$size*, and use Caravan only when the $\mathcal{Q}_{\mathcal{P}}$ has at least *Min $\mathcal{Q}_{\mathcal{P}}$size* elements. If we have fewer elements, we perform baseline searches until enough neighbors are found and placed in the list of pending queries, when Caravan can be used again. This control is performed at the software level.

Considering this, Figure 5.11 shows the end-to-end number of cycles for the Segmentation application in Autoware.ai. The chart contains data for i) the baseline code; ii) Caravan-SW (i.e., PCL + existing AVX512 instructions exploiting *search similarity* observed

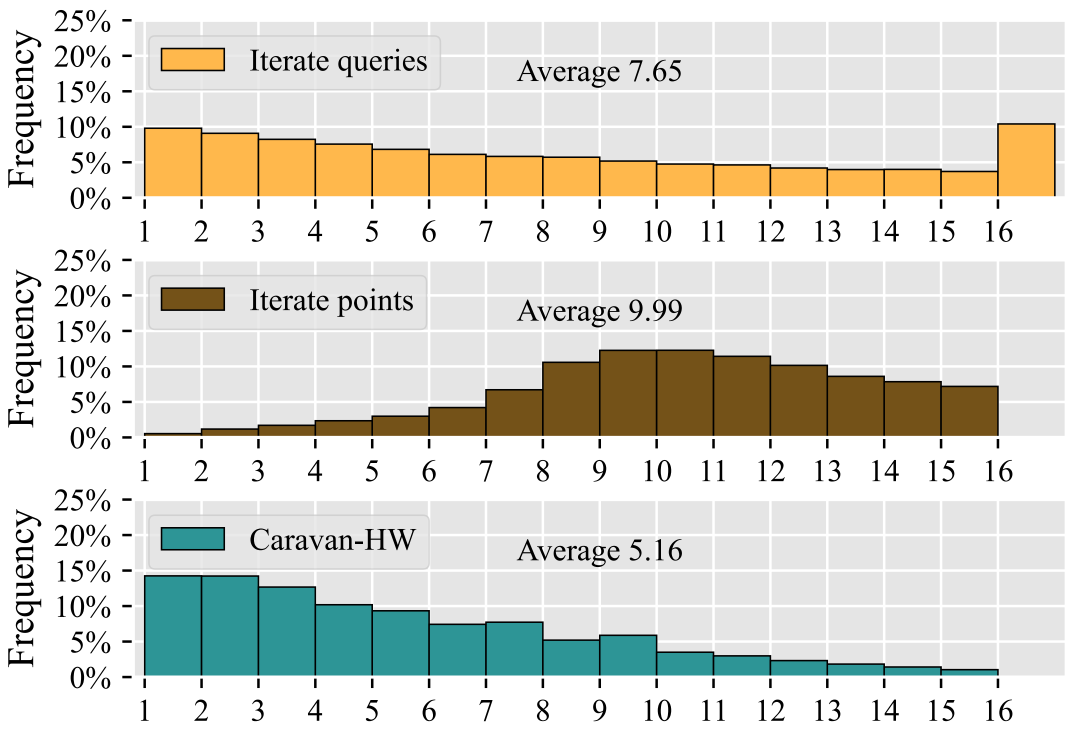


Figure 5.14 Required steps on leaf processing to vectorly compute all-to-all iterating on different variables with a *Min* $\mathcal{Q}_{\mathcal{P}}$*size* of 16.

in Section 5.1.1); and iii) and Caravan-HW (i.e., Caravan-SW + architectural support with two new instructions ***EDIRS*** and ***EDIRE***, see Section 5.1.4). It shows results for different *Min* $\mathcal{Q}_{\mathcal{P}}$*sizes*, being best with a *Min* $\mathcal{Q}_{\mathcal{P}}$*size* of 8. The value of 8 can balance good usage of the VPU and the percentage of searches with multiple-point queries while being large enough to avoid overheads. From this point on we discuss results for this *Min* $\mathcal{Q}_{\mathcal{P}}$*size* value.

Regardless of the chosen *Min* $\mathcal{Q}_{\mathcal{P}}$*size*, Caravan proves to be very effective in reducing end-to-end latency for the segmentation. By exploiting query similarity observations (Section 5.1.1) with Caravan-SW, we can greatly improve the baseline code, by up to 1.85× with a *Min* $\mathcal{Q}_{\mathcal{P}}$*size* of 8. The end-to-end benefits come directly from the neighbor search improvement. As shown in Figure 5.12, Caravan-SW improves the neighbor search by 4.05×. Notice these benefits can be directly applied in today's CPUs, since all we did was exploit data similarity to unleash SIMD parallelism during k-d tree search, leveraging available VPUs for an application formerly hard to vectorize.

However, Caravan-SW encounters sparsity, particularly in the leaf nodes as we discussed in Section 5.1.2. Yet, with the architectural support of Caravan-HW, the *leaf processing* sparsity can be greatly mitigated. Figure 5.14 illustrates how. The upper two histograms plot the distribution of valid queries and points found in the k-d tree leaves. This empirical data correlates with the left and center columns of Figure 5.7. As shown in Figure 5.7, if we want to compare all elements of A with all elements of B (as queries and points during leaf processing), we need to iterate and broadcast element by element of either A or B. In the context of neighbor search (Figure 5.14), this means iterating query by query, with an average of 7.65 iterations to compare against all points, or iterating point by point, with an average of 9.99 iterations to compare against all queries. In either case, with sparsity.

Caravan-HW instructions break the sparsity barrier that held back maximum VPU utilization. With the use of the new instructions (Section 5.1.4), the VPU unused lanes can be filled with useful operators, anticipating calculations and reducing the average number of iterations to 5.16 in average. Notice that in Caravan-SW we iterate by queries as it incurs fewer iterations, and also to leverage locality when writing to the list of neighbors of queries.

Caravan-HW improves VPU utilization and, by doing so, boosts end-to-end speedup to 1.97$\times$. If looking at the neighbor search only, the benefits are more than 5-fold. The non-SIMD portion of the code (for example, the function calls that perform recursion) limits the benefits from reaching the maximum 16$\times$ possible with AVX512. In any case, Caravan-HW has its end-to-end speedup limited by the fraction of the application it can accelerate, as shown in Figure 1.5. Still, Caravan-HW delivers end-to-end speedups that are not that far from the theoretical maximum (i.e., neighbor search taking 0 cycles to execute) with 1.97$\times$ versus 2.56$\times$. It is very important to acknowledge the fraction subject to acceleration of the target applications (Amdahl's law) to avoid very aggressive and hardware-hungry solutions that in the end would reach similar end-to-end gains as Caravan (see Section 3.2). In summary, we show that a modern CPU with two new instructions and appropriate software adaptation can be a very effective approach to improve neighbor search. These benefits can directly translate into faster 3D computer vision algorithms, which are fundamental for safety-critical and real-time applications such as AD.

5.4 Chapter Takeaways

In this Chapter, we presented Caravan, a hardware/software co-design to maximize SIMD efficiency on point cloud neighbor search. Caravan exploits similarity between consecutive queries, which translates into *search similarity* in k-d trees. Caravan-SW exploits this behavior at the software level, by packing multiple queries together. When a visited node is of common interest for multiple queries, they benefit from sharing node metadata, including subspace divisions that guide k-d tree navigation, and leaf points, the potential neighbors to be found. Internally, Caravan-SW utilizes SIMD instructions, to exploit data-level parallelism and restraint instruction count, and speeds up the code by $1.85\times$. The small differences in the queries, however, cause search divergence, particularly in leaf nodes. The divergence causes traditional SIMD code to underutilize the VPU lanes. Caravan-HW provides architectural support to mitigate this. With two new instructions, the programmer can obtain a pair of dense elements indices, and use it to shuffle the vectors so that VPU lanes are occupied with valid operators. The co-design is capable of improving neighbor search performance by 5.19$\times$, and end-to-end point cloud segmentation by 1.97$\times$. Its end-to-end performance improvement, therefore, is comparable to ASIC accelerators, while maintaining programmability and requiring only two new simple instructions.

6 Conclusions

“Success means being successful, not just having the potential for success.”

The Book of Disquiet (1982)
Fernando Pessoa

After presenting the thesis contributions in detail in the previous Chapters, we now summarize the ideas with the conclusions. We also point out some research problems for future work.

6.1 Conclusions

The rollout of AVs promises important societal advances, with increased safety and a world of market possibilities. But designing the computer systems for ADS is challenging. Different requirements must be balanced including performance, energy efficiency, volume, and cost. In this context, providing appropriate hardware support for AD is key to pushing advances while accounting for these constraints.

In this thesis, we confronted this challenge. From a computer architecture point of view, we started by understanding the AD workloads. For that, we characterized a state-of-the-art AD software stack, to understand how the different algorithms work, and how demanding they were for the underlying hardware. With this characterization, we could identify which tasks require higher attention. At the same time, looking at a real-life solution shed light on the important role of AV’s sensors, and how costly it is for the computing platform to cope with their generated data. Looking closer, we found LiDAR processing to be a relevant problem for future AD, as multiple algorithms rely on their collected point clouds.

Particularly, the neighbor search operation stood out, accounting for the majority of the execution time of segmentation and localization, but also relevant in many other scenarios such as 3D DNNs.

From this point, we proposed two techniques to enhance the hardware support for the neighbor search. We focused on extending the CPU, considering they are the typical execution platform for this operation. Additionally, CPUs provide several benefits compared to alternatives like hardware accelerators, including programmability, ease of integration, and lack of communication overheads.

Most of the insights for our techniques came from understanding the use of k-d trees – the *de facto* data structure used for neighbor search – and LiDAR sensors functioning. For example, the first technique, K-D Bonsai, observed and exploited value similarity among points stored in k-d tree leaves to tackle a particular case of a long-known problem for computer architects: accessing memory. We proposed new CPU instructions to compress leaf points, sharing the sign and exponent FP fields, reducing their representation considering the LiDAR range of operation, and also shrinking the mantissa field. Additionally, instructions were added to decompress leaf points and to account for worst-case errors during distance calculation to detect potential errors. This technique significantly reduced data movement to fetch k-d tree leaves, improving the average end-to-end latency of point cloud segmentation by 9.26% on average, with little area and power cost.

Following, in a second technique, Caravan, we observed and exploited similarities among subsequent searches to pursue data-level parallelism by employing and increasing the efficiency of SIMD operations. The approach was implemented through a hardware and software co-design. As we observed, subsequent queries had similar coordinates, incurring similar tree traversals that could operate in a SIMD fashion. The software was responsible for adapting the k-d tree traversal code to support SIMD instructions, grouping subsequent queries together, and handling divergences. The hardware part consisted of two new instructions to cope with the sparsity in the leave's SIMD vectors. Combined, the new instructions generate the list of indices to shuffle valid elements from source SIMD vectors, such that SIMD efficiency is maximized. This technique achieved 97% speedup for end-to-end latency of point cloud segmentation on average, also being cheap in area and power budgets.

Overall, this thesis showed ways for adapting CPU designs to the AVs challenges ahead. We believe our ideas can be adopted in next-generation CPUs, in the form of ISA extensions. By leveraging the rock solid CPU programming model and existing hardware structures, our incremental CPU changes allow for a programmable way of improving hardware support for AD. On the one hand, the reduction in latency achieved by our techniques can directly serve to reduce reaction time, aiding safety. Alternatively, the time slack created by our solutions can be spent to perfect other stages on the AD pipeline to, for example, increase the quality of perception algorithms. Finally, the modifications are minimally intrusive, incurring minimal area and power overheads. Also, since they are constrained in specific functional units, they do not affect general CPU structures, hence not interfering in the CPU performance in other use cases.

6.2 Future Work

Due to time and workforce limitations, a PhD thesis cannot explore all research opportunities. Yet, we try to list potential future work for the reader interested in the topics presented in this document. The list is by no means exhaustive.

For point clouds, generalizing the presented ideas in different domains and devices would be valuable. For example, VR glasses rely on point cloud processing, having their own particularities and constraints. In indoor uses, the point clouds can be denser, and with points much closer (less distance for angular aperture to place them apart). Although requiring further study, this can boost similarities in either points and queries, which we exploited with K-D Bonsai and Caravan.

Also, porting the ideas to devices such as GPUs could be worth investigating, particularly on integrated ones that are less power-hungry (a concern for AD). These devices have higher SIMD capabilities than CPUs, but less cache space per core, making them a good target platform to extend Caravan and K-D Bonsai techniques. This also aligns with the crescent use of 3D DNNs, which commonly rely on GPU execution for convolutions. While neighbor search proceeds and complements the convolution step, it does not have the embarrassingly parallel characteristic of the other. This pressures for efficient neighbor search in the solutions as well.

Complementary ideas to ours could also be carried out. For example, while our instructions were manually added to the PCL and used by the application, providing compiler support would significantly increase their chances of adoption. One challenge here is to know whether locality (in queries or points) holds for the target usage. In our case, this was discovered with profiling, but identifying it at compile time *seems* harder. A possible solution is a scheme that allows for run-time adjustments based on hardware counters to choose whether or not to use our instructions.

Similarities in search navigation could also be exploited to adjust major CPU structures, such as the prefetcher and the branch prediction units. One way of exploiting that would be to pass context to these structures (e.g., the current node and query values). The challenge would be not to hurt baseline CPU performance running other algorithms, where this scheme may not be useful.

Moving to a broader view, on architectures for AD, we believe there are several open research areas as well. For example, other algorithms in the software stack could also benefit from a deeper look as we did for segmentation (and to some extent, localization). In the SoC level, hardware to pre-adjust point clouds in different structures could be valuable. For example, some algorithms organize point clouds on k-d trees, but others organize them in voxels, others in radial slices, and so on. A central unit to organize and filter point clouds for these different uses in advance could be advantageous. In a more aggressive specialization, hardware support for 3D registers could also be studied. For example, direct 3D operations could be used to reduce instruction count even further.

References

[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. General-Purpose Graphics Processor Architectures. *Synthesis Lectures on Computer Architecture*, 13(2): 1–121, may 2018. ISSN 19353243. doi: 10.2200/S00848ED1V01Y201804CAC044. URL https://www.morganclaypool.com/doi/10.2200/S00848ED1V01Y201804CAC044.

[2] James Anderson, Nidhi Kalra, Karlyn Stanley, Paul Sorensen, Constantine Samaras, and Oluwatobi Oluwatola. *Autonomous Vehicle Technology: A Guide for Policymakers*. RAND Corporation, 2016. ISBN 9780833083982. doi: 10.7249/RR443-2. URL www.rand.org/giving/contributehttp://www.rand.org/pubs/research_reports/RR443-2.html.

[3] Fabien André. Exploiting Modern Hardware for High-Dimensional Nearest Neighbor Search. 2017. URL http://arxiv.org/abs/1712.02912.

[4] ApolloAuto/Apollo. An open autonomous driving platform, 2021. URL https://github.com/ApolloAuto/apollo.

[5] Ehsan K. Ardestani and Jose Renau. ESESC: A fast multicore simulator using Time-Based Sampling. In *Proceedings - International Symposium on High-Performance Computer Architecture*, pages 448–459, 2013. ISBN 9781467355858. doi: 10.1109/HPCA.2013.6522340.

[6] ARM. Understanding the Armv8.x extensions. page 15, 2019.

[7] ARM. ARM Architecture Reference Manual - Armv8, for A-profile architecture. pages 1–1138, 2021. ISSN <null>.

[8] Arm. Arm Automotive Enhanced – Arm®, 2024. URL https://www.arm.com/products/silicon-ip-cpu/automotive-enhanced.

[9] Arm. Arm and Nuro Partner to Deliver AI-first Autonomous Technology for Commercial Scale - Arm Newsroom, 2024. URL https://newsroom.arm.com/news/arm-nuro-autonomous-partnership.

[10] Autoware. Autoware.AI · GitLab, . URL https://gitlab.com/autowarefoundation/autoware.ai.

[11] Autoware. Autoware Wiki, . URL https://github.com/Autoware-AI/autoware.ai/wiki/Overview.

[12] Autoware. Autonomous Valet Parking 2020, 2020. URL https://autoware.org/autonomous-valet-parking-2020/.

[13] Autoware. Autoware.AI · GitHub, 2023. URL https://github.com/Autoware-AI.

[14] Autoware Foundation. The Autoware Foundation - Open Source for Autonomous Driving, 2024. URL https://www.autoware.org/.

[15] Baidu. Releases · ApolloAuto/apollo · GitHub. URL https://github.com/ApolloAuto/apollo/releases?page=1.

[16] Baidu. Apollo Go Robotaxi: Baidu's autonomous ride-hailing service provider, 2022. URL https://en.apollo.auto/robotaxi.

[17] Baidu. Apollo, 2022. URL https://en.apollo.auto/.

[18] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. Computer and Redundancy Solution for the Full Self-Driving Computer. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–22. IEEE, aug 2019. ISBN 978-1-7281-2089-8. doi: 10.1109/HOTCHIPS.2019.8875645. URL https://ieeexplore.ieee.org/document/8875645/.

[19] Pedro H. E. Becker, José-María Arnau, and Antonio González. Demystifying Power and Performance Bottlenecks in Autonomous Driving Systems. In *Proceedings - 2020 IEEE International Symposium on Workload Characterization, IISWC 2020*, pages 205–215. IEEE, oct 2020. ISBN 9781728176451. doi: 10.1109/IISWC50251.2020.00028. URL https://ieeexplore.ieee.org/document/9251251/.

[20] Pedro H. E. Becker, José-María Arnau, and Antonio González. Characterizing Self-driving Tasks in General-purpose Architectures. In *ACACES 2021 Poster Abstracts*, pages 117–120. HiPEAC, the European Network of Excellence on High Performance Embedded Architecture and Compilation., 2021. ISBN 978-88-905806-8-0.

[21] Pedro H. E. Becker, José-María Arnau, and Antonio González. Boosting Point Cloud Search with a Vector Unit. In *RoboARCH @ MICRO*, 2023.

[22] Pedro H. E. Becker, José-María Arnau, and Antonio González. K-D Bonsai: ISA-Extensions to Compress K-D Trees for Autonomous Driving Tasks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, New York, NY, USA, jun 2023. ACM. ISBN 9798400700958. doi: 10.1145/3579371.3589055. URL https://dl.acm.org/doi/10.1145/3579371.3589055.

[23] Pedro H. E. Becker, Franyell Silfa, José-María Arnau, and Antonio González. Caravan: A Hardware/Software Co-Design for Efficient SIMD Neighbor Search on Point Clouds. In *Under Review*, 2024.

[24] Reinhold Behringer, Sundar Sundareswaran, Robert Daily, David Bevly, Brian Gregory, Richard Elsley, Bob Addison, and Wayne Guthmiller. The DARPA grand challenge - development of an autonomous vehicle. In *IEEE Intelligent Vehicles Symposium*, pages 226–231. IEEE, 2004. ISBN 0-7803-8310-9. doi: 10.1109/IVS.2004.1336386. URL http://ieeexplore.ieee.org/document/1336386/.

[25] Behnam Behroozpour, Phillip A. M. Sandborn, Ming C. Wu, and Bernhard E. Boser. Lidar System Architectures and Circuits. *IEEE Communications Magazine*, 55(10): 135–142, oct 2017. ISSN 0163-6804. doi: 10.1109/MCOM.2017.1700030. URL http://ieeexplore.ieee.org/document/8067701/.

[26] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, sep 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL https://dl.acm.org/doi/10.1145/361002.361007.

[27] Paul J. Besl and Neil D. McKay. A Method for Registration of 3-D Shapes. In Paul S. Schenker, editor, *Sensor Fusion IV: Control Paradigms and Data Structures*, volume 1611, pages 586–606, apr 1992. doi: 10.1117/12.57955. URL http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=981454.

[28] Peter Biber. The Normal Distributions Transform: A New Approach to Laser Scan Matching. In *IEEE International Conference on Intelligent Robots and Systems*, volume 3, pages 2743–2748. IEEE, 2003. ISBN 0-7803-7860-1. doi: 10.1109/iros.2003.1249285. URL https://www.researchgate.net/publication/4045903http://ieeexplore.ieee.org/document/1249285/.

[29] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, aug 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL https://doi.org/10.1145/2024716.2024718.

[30] Brian Bushard. Driverless Food Deliveries Grow: Uber Eats Launching In Calif., Texas This Fall, 2022. URL https://www.forbes.com/sites/brianbushard/2022/09/08/driverless-food-deliveries-grow-uber-eats-launching-in-calif-texas-this-fall/.

[31] Canon Inc. Canon Technology | Canon Science Lab | Photographs. URL https://global.canon/en/technology/s_labo/light/003/01.html.

[32] Tomislav Capuder, Danijela Miloš Sprčić, Davor Zoričić, and Hrvoje Pandžić. Review of challenges and assessment of electric vehicles integration policy goals: Integrated risk analysis approach, 2020. ISSN 01420615. URL https://doi.org/10.1016/j.ijepes.2020.105894.

[33] Lukas Cavigelli and Luca Benini. Origami: A 803 GOp/s/W Convolutional Network Accelerator. *arXiv*, dec 2015. doi: 10.1109/TCSVT.2016.2592330. URL http://arxiv.org/abs/1512.04295http://dx.doi.org/10.1109/TCSVT.2016.2592330.

[34] Cen Chen, Xiaofeng Zou, Hongen Shao, Yangfan Li, and Kenli Li. Point Cloud Acceleration by Exploiting Geometric Similarity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023*, pages 1135–1147, New York, NY, USA, oct 2023. ACM. ISBN 9798400703294. doi: 10.1145/3613424.3614290. URL https://doi.org/10.1145/3613424.3614290https://dl.acm.org/doi/10.1145/3613424.3614290.

[35] Faquan Chen, Rendong Ying, Jianwei Xue, Fei Wen, and Peilin Liu. ParallelNN: A Parallel Octree-based Nearest Neighbor Search Accelerator for 3D Point Clouds. In *Proceedings - International Symposium on High-Performance Computer Architecture*, volume 2023-Febru, pages 403–414, 2023. ISBN 9781665476522. doi: 10.1109/HPCA56546.2023.10070940.

[36] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284, New York, NY, USA, feb 2014. ACM. ISBN 9781450323055. doi: 10.1145/2541940.2541967. URL http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2541940.2541967https://dl.acm.org/doi/10.1145/2541940.2541967.

[37] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, jan 2017. ISSN 0018-9200. doi: 10.1109/JSSC.2016.2616357. URL http://ieeexplore.ieee.org/document/7738524/.

[38] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, dec 2014. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.58. URL http://ieeexplore.ieee.org/document/7011421/.

[39] Hiroyuki Chishiro, Kazutoshi Suito, Tsutomu Ito, Seiya Maeda, Takuya Azumi, Kenji Funaoka, and Shinpei Kato. Towards heterogeneous computing platforms for autonomous driving. *2019 IEEE International Conference on Embedded Software and Systems, ICESS 2019*, (December 2018), 2019. doi: 10.1109/ICESS.2019.8782446.

[40] Lauranne Choquin and Staff Information Developer. Arm Custom Instructions: Enabling Innovation and Greater Flexibility on Arm. Technical Report February, 2020. URL https://developer.arm.com/documentation/102891/0100.

[41] Design Compiler. Design Compiler. URL https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[42] Cruise LLC. Cruise Self Driving Cars | Autonomous Vehicles | Driverless Rides & Delivery, 2024. URL https://getcruise.com/.

[43] Daimler Truck. Autonomous Driving. URL https://www.daimlertruck.com/en/innovation/autonomous-driving.

[44] Hatem Darweesh, Eijiro Takeuchi, Kazuya Takeda, Yoshiki Ninomiya, Adi Sujiwo, Luis Yoichi Morales, Naoki Akai, Tetsuo Tomizawa, and Shinpei Kato. Open Source Integrated Planner for Autonomous Navigation in Highly Dynamic Environments. *Journal of Robotics and Mechatronics*, 29(4):668–684, aug 2017. ISSN 1883-8049. doi: 10.20965/jrm.2017.p0668. URL https://www.fujipress.jp/jrm/rb/robot002900040668.

[45] Hatem Darweesh, Eijiro Takeuchi, Kazuya Takeda, Yoshiki Ninomiya, Adi Sujiwo, Luis Yoichi Morales, Naoki Akai, Tetsuo Tomizawa, and Shinpei Kato. Open source integrated planner for autonomous navigation in highly dynamic environments. *Journal of Robotics and Mechatronics*, 29(4):668–684, 2017. ISSN 18838049. doi: 10.20965/jrm.2017.p0668. URL https://www.researchgate.net/publication/319201866.

[46] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory power estimation and capping. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 189–194, 2010. ISBN 9781450301466. doi: 10.1145/1840845.1840883.

[47] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio López, and Vladlen Koltun. CARLA: An Open Urban Driving Simulator.

[48] Daniel J. Fagnant and Kara M. Kockelman. The travel and environmental implications of shared autonomous vehicles, using agent-based model scenarios. *Transportation Research Part C: Emerging Technologies*, 40:1–13, 2014. ISSN 0968090X. doi: 10.1016/j.trc.2013.12.001. URL https://www.sciencedirect.com/science/article/abs/pii/S0968090X13002581.

[49] Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. Mesorasi: Architecture support for point cloud analytics via delayed-aggregation. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, volume 2020-Octob, pages 1037–1050, 2020. ISBN 9781728173832. doi: 10.1109/MICRO50266.2020.00087. URL https://github.com/horizon-research/efficient-deep-.

[50] Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics. In *Proceedings - International Symposium on Computer Architecture*, volume 1, pages 962–977. ACM, 2022. ISBN 9781450386104. doi: 10.1145/3470496.3527395. URL https://doi.org/10.1145/3470496.3527395.

[51] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3):209–226, sep 1977. ISSN 0098-3500. doi: 10.1145/355744.355745. URL https://dl.acm.org/doi/10.1145/355744.355745.

[52] Péter Gáspár, Zsolt Szalay, and Szilárd Aradi. Highly Automated Vehicle Systems. (January 2018):187, 2014. URL http://www.mogi.bme.hu/TAMOP/jarmurendszerek_iranyitasa_angol/index.html.

[53] Andreas Geiger, Philip Lenz, and Raquel Urtasun. The KITTI Vision Benchmark Suite - 3D Object Detection Evaluation 2017, . URL https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d.

[54] Andreas Geiger, Philip Lenz, and Raquel Urtasun. The KITTI Vision Benchmark Suite - Visual Odometry / SLAM Evaluation 2012, . URL https://www.cvlibs.net/datasets/kitti/eval_odometry.php.

[55] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the KITTI vision benchmark suite. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, 2012. ISSN 10636919. doi: 10.1109/CVPR.2012.6248074. URL http://www.cvlibs.net/publications/Geiger2012CVPR.pdf.

[56] Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. Buffer k-d trees: Processing massive nearest neighbor queries on GPUs. In *31st International Conference on Machine Learning, ICML 2014*, volume 1, pages 312–320, 2014. ISBN 9781634393973.

[57] Amos Goldman. SIMD K-nearest-neighbors implementation, U.S. Patent 10042813, 2018.

[58] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jimenez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the Samsung Exynos CPU Microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51. IEEE, may 2020. ISBN 978-1-7281-4661-4. doi: 10.1109/ISCA45697.2020.00015. URL https://ieeexplore.ieee.org/document/9138988/.

[59] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS 2011 - IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144. IEEE, apr 2011. ISBN 9781612843681. doi: 10.1109/ISPASS.2011.5762730. URL http://ieeexplore.ieee.org/document/5762730/.

[60] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008(166):8, 2008.

[61] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Transactions on Graphics*, 33(4):1–11, jul 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601174. URL https://dl.acm.org/doi/10.1145/2601097.2601174.

[62] Simon Heinzle, Gaël Guennebaud, Mario Botsch, and Markus Gross. A hardware processing unit for point sets. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 21–31, 2008. ISBN 9783905674095.

[63] Rasheed Hussain and Sherali Zeadally. Autonomous Cars: Research Results, Issues, and Future Challenges. *IEEE Communications Surveys and Tutorials*, 21(2):1275–1313, 2019. ISSN 1553877X. doi: 10.1109/COMST.2018.2869360.

[64] Intel. Intel® Intrinsics Guide, 2023. URL https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[65] Intel Corp. Intel Xeon W2155 Processor 13.75M Cache 3.30 GHz Product Specifications, 2023. URL https://ark.intel.com/content/www/us/en/ark/products/125042/intel-xeon-w-2155-processor-13-75m-cache-3-30-ghz.html.

[66] Tatsuki Inuzuka, Toshiaki Nakamura, Shinichi Shinoda, and Yasuyuki Kojima. Image signal processing apparatus, 1993.

[67] Tier IV. Autoware Data. URL https://data.tier4.jp/.

[68] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6): 60–68, 2015. ISSN 02721732. doi: 10.1109/MM.2015.133.

[69] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. *Proceedings - 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018*, (April):287–296, 2018. doi: 10.1109/ICCPS.2018.00035.

[70] Kaz Sato, Cliff Young, and David Patterson. An in-depth look at Google's first Tensor Processing Unit (TPU) | Google Cloud Blog, 2017. URL https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[71] Klaas Klasing, Dirk Wollherr, and Martin Buss. A clustering method for efficient segmentation of 3D laser data. In *2008 IEEE International Conference on Robotics and Automation*, pages 4043–4048. IEEE, may 2008. ISBN 978-1-4244-1646-2. doi: 10.1109/ROBOT.2008.4543832. URL http://ieeexplore.ieee.org/document/4543832/.

[72] Donald E Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201896850. doi: 10.5555/280635.

[73] Lukas König, Christian Heinzemann, Alberto Griggio, Michaela Klauck, Alessandro Cimatti, Franziska Henze, Stefano Tonetta, Stefan Küperkoch, Dennis Fassbender, and Michael Hanselmann. Towards Safe Autonomous Driving : Model Checking a Behavior Planner during Development. *30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2024.

[74] Philip Koopman and Michael Wagner. Challenges in Autonomous Vehicle Testing and Validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016. ISSN 23275634. doi: 10.4271/2016-01-0128.

[75] KPMG LLP and Research Center for Automotive. Self-driving cars: The next revolution. Technical report, 2012. URL https://www.cargroup.org/wp-content/uploads/2017/02/Self_driving-cars-The-next-revolution.pdf.

[76] Ian Krietzberg. What's stopping Tesla from achieving Level 3 self-driving - TheStreet, 2023. URL https://www.thestreet.com/technology/whats-stopping-tesla-from-achieving-level-3-self-driving.

[77] Rico Krueger, Taha H Rashidi, and John M Rose. Preferences for shared autonomous vehicles. *Transportation Research Part C: Emerging Technologies*, 69:343–355, aug 2016. ISSN 0968090X. doi: 10.1016/j.trc.2016.06.015. URL http://dx.doi.org/10.1016/j.trc.2016.06.015https://linkinghub.elsevier.com/retrieve/pii/S0968090X16300870.

[78] Kristofer D Kusano, John M Scanlon, Yin-Hsiu Chen, Timothy L. McMurry, Ruoshu Chen, Tilia Gode, and Trent Victor. Comparison of Waymo Rider-Only Crash Data to Human Benchmarks at 7.1 Million Miles. dec 2023. URL http://arxiv.org/abs/2312.12675.

[79] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. Technical report, 2019. URL https://github.com/nutonomy/second.pytorch.

[80] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, New York, NY, USA, jun 2010. ACM. ISBN 9781450300537. doi: 10.1145/1815961.1816021. URL https://dl.acm.org/doi/10.1145/1815961.1816021.

[81] LG Electronics Inc. LGSVL Simulator: An Autonomous Vehicle Simulator, 2019. URL https://www.lgsvlsimulator.com/.

[82] Bai Li, Shaoshan Liu, Jie Tang, Jean Luc Gaudiot, Liangliang Zhang, and Qi Kong. Autonomous Last-Mile Delivery Vehicles in Complex Traffic Environments, 2020. ISSN 15580814.

[83] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *Transactions on Architecture and Code Optimization*, 10(1), 2013. ISSN 15443566. doi: 10.1145/2445572.2445577.

[84] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *Transactions on Architecture and Code Optimization*, 10(1), 2013. ISSN 15443566. doi: 10.1145/2445572.2445577. URL http://dx.doi.org/10.1145/2445572.2445577.

[85] You Li and Javier Ibanez-Guzman. Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems. *IEEE Signal Processing Magazine*, 37(4):50–61, jul 2020. ISSN 15580792. doi: 10.1109/MSP.2020.2973615. URL http://www.hesaitech.com/en/autonomous_driving.htmlhttps://ieeexplore.ieee.org/document/9127855/.

[86] Shih Chieh Lin, Yunqi Zhang, Chang Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. *ACM SIGPLAN Notices*, 53(2):751–766, 2018. ISSN 15232867. doi: 10.1145/3173162.3173191.

[87] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. PointAcc: Efficient point cloud accelerator. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, volume 13, pages 449–461. ACM, 2021. ISBN 9781450385572. doi: 10.1145/3466752.3480084. URL https://doi.org/10.1145/3466752.3480084.

[88] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing Systems for Autonomous Driving: State of the Art and Challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, apr 2021. ISSN 2327-4662. doi: 10.1109/JIOT.2020.3043716. URL https://ieeexplore.ieee.org/document/9288755/.

[89] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector - GitHub. URL https://github.com/weiliu89/caffe/tree/ssd.

[90] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. *arXiv*, dec 2016. doi: https://doi.org/10.48550/arXiv.1512.02325. URL http://arxiv.org/abs/1512.02325.

[91] Volker Lohweg, Carsten Diederichs, and Dietmar Müller. Algorithms for hardware-based pattern recognition. *Eurasip Journal on Applied Signal Processing*, 2004 (12):1912–1920, sep 2004. ISSN 11108657. doi: 10.1155/S1110865704404247/

METRICS. URL https://link.springer.com/articles/10.1155/S1110865704404247https://link.springer.com/article/10.1155/S1110865704404247.

[92] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 Simulator: Version 20.0+. Technical report, 2020. URL http://arxiv.org/abs/2007.03152.

[93] Martin Magnusson. *The Three-Dimensional Normal-Distributions Transform — an Efficient Representation for Registration, Surface Analysis, and Loop Detection*, volume 10. Örebro universitet, Örebro, 2008. ISBN 978-91-7668-696-6. URL http://www.aass.oru.se/Research/Learning/publications/2009/Magnusson_2009-Doctoral_Thesis-3D_NDT.pdf.

[94] Marius Muja and David Lowe. FLANN - Fast Library for Approximate Nearest Neighbors User Manual. 2013. URL http://www.cs.ubc.ca/research/flann/.

[95] Vazgen Melikyan, Meruzhan Martirosyan, Anush Melikyan, and Gor Piliposyan. 14nm educational design kit: Capabilities deployment and future. In *Small Systems Simulation Symposium*, 2018.

[96] Ashutosh Mishra, Hyunbin Park, Shiho Kim, and Jaekwang Cha. *Artificial Intelligence and Hardware Accelerators*. 2023. ISBN 9783031221705. doi: 10.1007/978-3-031-22170-5.

[97] Mobileye. Meet EyeQ®6: Our Most Advanced Driver-Assistance Chips Yet | Mobileye Blog, 2024. URL https://www.mobileye.com/blog/eyeq6-system-on-chip/.

[98] Mobileye. Mobileye Drive™ Enabling autonomous mobility, 2024. URL https://www.mobileye.com/future-of-mobility/.

[99] Ali Mosleh, Avinash Sharma, Emmanuel Onzon, Fahim Mannan, Nicolas Robidoux, and Felix Heide. Hardware-in-the-Loop End-to-End Optimization of Camera Image Processing Pipelines. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7526–7535. IEEE, jun 2020. ISBN 978-1-7281-7168-5. doi: 10.1109/CVPR42600.2020.00755. URL https://ieeexplore.ieee.org/document/9156332/.

[100] Marius Muja and David Lowe. FLANN - Fast Library for Approximate Nearest Neighbors, 2021. URL https://github.com/tkircher/flann.

[101] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP 2009 - Proceedings of the 4th International Conference on Computer Vision Theory and Applications*, volume 1, pages 331–340, 2009. ISBN 9789898111692. doi: 10.5220/0001787803310340.

[102] Toshio Nakakuki. IMAGE SIGNAL PROCESSOR, 2003.

[103] National Highway Traffic Safety Administration and US Department of Transportation. Automated driving systems: a vision for safety 2.0. Technical report, National Highway Traffic Safety Administration US Department of Transportation, 2017. URL https://www.nhtsa.gov/sites/nhtsa.gov/files/13069a-ads2.0_090617_v9a_tag.pdf.

[104] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007. ISSN 15232867.

[105] Anh Nguyen, Abraham Monrroy Cano, Masato Edahiro, and Shinpei Kato. Fast euclidean cluster extraction using GPUS. *Journal of Robotics and Mechatronics*, 32(3): 548–560, 2020. ISSN 18838049. doi: 10.20965/jrm.2020.p0548.

[106] Nuro. Delivering Safety: Nuro's Approach. Technical report, Nuro, 2018.

[107] NVIDIA. Leading Lidar Sensor Makers Build on NVIDIA DRIVE. URL https://blogs.nvidia.com/blog/lidar-sensor-nvidia-drive/.

[108] NVIDIA. The Journey to Zero Accidents - NVIDIA Drive. Technical report, Nvidia, 2019. URL https://www.nvidia.com/en-us/self-driving-cars/.

[109] NVIDIA. CUDA Math API API Reference Manual. 2022. URL https://docs.nvidia.com/cuda/pdf/CUDA_Math_API.pdf.

[110] NVIDIA. System Management Interface SMI | NVIDIA Developer, 2024. URL https://developer.nvidia.com/nvidia-system-management-interface.

[111] Daoshan OuYang and Hsi-Yung Feng. On the normal vector estimation for point cloud data from smooth surfaces. *Computer-Aided Design*, 37(10):1071–1079, sep 2005. ISSN 00104485. doi: 10.1016/j.cad.2004.11.005. URL www.elsevier.com/locate/cadhttps://linkinghub.elsevier.com/retrieve/pii/S001044850400226X.

[112] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel ® IA-32 and IA-64 Instruction Set Architectures. *Intel Manual*, (September):1–37, 2010.

[113] David A. Patterson and John L. Hennessy. *Computer architecture a quantative approach v6*. 2019. ISBN 978-0-12-811905-1. doi: 10.1016/B978-0-12-811905-1.

[114] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture*, 129:102561, aug 2022. ISSN 13837621. doi: 10.1016/j.sysarc.2022.102561. URL https://doi.org/10.1016/j.sysarc.2022.102561https://linkinghub.elsevier.com/retrieve/pii/S1383762122001138.

[115] Salvatore Pennisi. The Integrated Circuit Industry at a Crossroads: Threats and Opportunities. *Chips*, 1(3):150–171, oct 2022. ISSN 2674-0729. doi: 10.3390/chips1030010. URL https://www.mdpi.com/2674-0729/1/3/10.

[116] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. QuickNN: Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds. In *Proceedings - 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA 2020*, pages 180–192. Institute of Electrical and Electronics Engineers Inc., feb 2020. ISBN 9781728161495. doi: 10.1109/HPCA47549.2020.00024.

[117] Point Cloud Library. Point Cloud Library | The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing., 2020. URL https://pointclouds.org/.

[118] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep learning on point sets for 3D classification and segmentation. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, volume 2017-Janua, pages 77–85, 2017. ISBN 9781538604571. doi: 10.1109/CVPR.2017.16.

[119] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 5100–5109, 2017.

[120] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. Technical report. URL http://stair.stanford.edu.

[121] Arya Senna Abdul Rachman. *3D-LIDAR Multi Object Tracking for Autonomous Driving: Multi-target Detection and Tracking under Urban Road Uncertainties.* PhD thesis, 2016.

[122] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. 2018. URL http://arxiv.org/abs/1804.02767.

[123] Joseph Redmon and Ali Farhadi. YOLO: Real-Time Object Detection, 2018. URL https://pjreddie.com/darknet/yolo/.

[124] Ursa Robotics. Ursa Robotics, 2023. URL https://ursa.ai/.

[125] Daniel L. Rosenband. Inside Waymo's self-driving car: My favorite transistors. In *IEEE Symposium on VLSI Circuits, Digest of Technical Papers*, volume 3, pages C20–C21. IEEE, jun 2017. ISBN 9784863486065. doi: 10.23919/VLSIC.2017.8008500. URL http://ieeexplore.ieee.org/document/8008500/.

[126] Ros.org. Bags - ROS Wiki, . URL http://wiki.ros.org/Bags.

[127] Ros.org. rosbag - ROS Wiki, . URL http://wiki.ros.org/rosbag.

[128] Santiago Royo and Maria Ballesta-Garcia. An Overview of Lidar Imaging Systems for Autonomous Vehicles. *Applied Sciences*, 9(19):4093, sep 2019. ISSN 2076-3417. doi: 10.3390/app9194093. URL https://www.mdpi.com/2076-3417/9/19/4093.

[129] Radu Bogdan Rusu. Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments. *KI - Kunstliche Intelligenz*, 24(4):345–348, 2010. ISSN 16101987. doi: 10.1007/s13218-010-0059-6.

[130] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *2011 IEEE International Conference on Robotics and Automation*, volume 74, pages 1–4. IEEE, may 2011. ISBN 978-1-61284-386-5. doi: 10.1109/ICRA.2011.5980567. URL http://ieeexplore.ieee.org/document/5980567/.

[131] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. Full speed ahead: Detailed architectural simulation at near-native speed. In *Proceedings - 2015 IEEE International Symposium on Workload Characterization, IISWC 2015*, pages 183–192. Institute of Electrical and Electronics Engineers Inc., oct 2015. ISBN 9781509000883. doi: 10.1109/IISWC.2015.29.

[132] Matthias Schreier, Volker Willert, and Jürgen Adamy. Compact Representation of Dynamic Driving Environments for ADAS by Parametric Free Space and Dynamic Object Maps. *IEEE Transactions on Intelligent Transportation Systems*, 17(2):367–384, 2016. ISSN 15249050. doi: 10.1109/TITS.2015.2472965.

[133] Brent Schwarz. Mapping the world in 3D. *Nature Photonics*, 4(7):429–430, jul 2010. ISSN 1749-4885. doi: 10.1038/nphoton.2010.148. URL http://www.nature.com/articles/nphoton.2010.148.

[134] Heiko G. Seif and Xiaolong Hu. Autonomous Driving in the iCity—HD Maps as a Key Challenge of the Automotive Industry. *Engineering*, 2(2):159–162, 2016. ISSN 20958099. doi: 10.1016/J.ENG.2016.02.010. URL http://dx.doi.org/10.1016/J.ENG.2016.02.010.

[135] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, volume 2016-Decem, pages 1–12. IEEE, oct 2016. ISBN 978-1-5090-3508-3. doi: 10.1109/MICRO.2016.7783751. URL http://ieeexplore.ieee.org/document/7783751/.

[136] N.A. Shneydor. Chapter 3 - Pure Pursuit. In *Missile Guidance and Pursuit*, number January, pages 47–76. Elsevier, 1998. ISBN 978-1-904275-37-4. doi: 10.1533/9781782420590.47. URL https://linkinghub.elsevier.com/retrieve/pii/B9781904275374500083.

[137] Sanman Singh Brar and Neeru Jindal. Camera Based Wearable Devices: A Strategic Survey from 2010 to 2021. *Wireless Personal Communications*, 133:667–681, 2023. doi: 10.1007/s11277-023-10787-5. URL https://doi.org/10.1007/s11277-023-10787-5.

[138] IEEE Computer Society. *IEEE Std 754™-2008 (Revision of IEEE Std 754-1985), IEEE Standard for Floating-Point Arithmetic*, volume 2008. 2008. ISBN 978-0-7381-5752-8. doi: 10.1109/IEEESTD.2008.4610935. URL https://ieeexplore.ieee.org/document/4610935.

[139] Society for Automotive Engineers. SAE J3016 - Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. *SAE International*, 4970(724):41, 2021.

[140] Zhuoran Song, Heng Lu, Gang Li, Li Jiang, Naifeng Jing, and Xiaoyao Liang. PRADA: Point Cloud Recognition Acceleration via Dynamic Approximation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, number 62202288, pages 1–6. IEEE, apr 2023. ISBN 9798400702334. doi: 10.23919/DATE56975.2023.10137301. URL https://ieeexplore.ieee.org/document/10137301/.

[141] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, mar 2017. ISSN 0272-1732. doi: 10.1109/MM.2017.35. URL http://ieeexplore.ieee.org/document/7924233/.

[142] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58:74–81, jun 2017. ISSN 01679260. doi: 10.1016/j.vlsi.2017.02.002. URL https://linkinghub.elsevier.com/retrieve/pii/S0167926017300755.

[143] Muhammad Sualeh and Gon Woo Kim. Dynamic Multi-LiDAR based multiple object detection and tracking. *Sensors (Switzerland)*, 19(6), 2019. ISSN 14248220. doi: 10.3390/s19061474.

[144] Tesla. Autopilot | Tesla, 2024. URL https://www.tesla.com/autopilot.

[145] Simon Thorpe, Denis Fize, and Catherine Marlot. Speed of processing in the human visual system. *Nature*, 381(6582):520–522, 1996. ISSN 00280836. doi: 10.1038/381520a0.

[146] TIER IV Inc. TIER IV certified in Level 4 autonomous driving: Sharing its design and process with partners. URL https://tier4.jp/en/media/detail/?sys_id=1dWp9ReZIYKHvP0ZtCKeVF&category=NEWS.

[147] TIER IV Inc. TIER IV, 2024. URL https://tier4.jp/en/.

[148] Alessandro Toschi, Mustafa Sanic, Jingwen Leng, Quan Chen, Chunlin Wang, and Minyi Guo. Characterizing Perception Module Performance and Robustness in Production-Scale Autonomous Driving System. In Xiaoxin Tang, Quan Chen, Pradip Bose, Weiming Zheng, and Jean-Luc Gaudiot, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11783 LNCS, pages 235–247, Cham, 2019. Springer International Publishing. ISBN 9783030307080. doi: 10.1007/978-3-030-30709-7_19.

[149] John Wilder Tukey. *Exploratory data analysis*, volume 2. Springer, 1977.

[150] Uber Technologies Inc. US Safety Report. Technical report, Uber Technologies Inc., San Francisco, CA, USA, 2020. URL https://www.uber.com/us/en/about/reports/us-safety-report/.

[151] Chris Urmson and William Whittaker. Self-driving cars and the Urban challenge. *IEEE Intelligent Systems*, 23(2):66–68, 2008. ISSN 15411672. doi: 10.1109/MIS.2008.34.

[152] Velodyne Lidar Inc. Product guide.

[153] Velodyne Lidar Inc. HDL-64E User's Manual, 2007.

[154] Velodyne Lidar Inc. Velodyne LiDAR HDL-64E | High Definition Real-Time 3D LiDAR, 2016. URL https://pdf.directindustry.com/pdf/velodynelidar/hdl-64e-datasheet/182407-676099.html.

[155] E. A. Wan and R. Van Der Merwe. The unscented Kalman filter for nonlinear estimation. In *IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium, AS-SPCC 2000*, pages 153–158, 2000. ISBN 0780358007. doi: 10.1109/ASSPCC.2000.882463.

[156] Yige Wang. Putting the brain in driverless vehicles with MDC. *Huawei Communicate*, 1(86):42–45, 2019. URL https://www.huawei.com/uk/about-huawei/publications/communicate/86/driverless-vehicles-with-mdc.

[157] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual. *RISC-V Foundation*, I, 2017. URL https://riscv.org/technical/specifications/.

[158] Waymo LLC. Designing the 5th-generation Waymo Driver. URL https://waymo.com/blog/2020/03/designing-5th-generation-waymo-driver/.

[159] Waymo LLC. The World's Most Experienced Driver™, 2024. URL https://waymo.com/.

[160] Vince Weaver. Reading RAPL energy measurements from Linux. URL https://web.eece.maine.edu/$\sim$vweaver/projects/rapl/.

[161] Stephen Williams. Icarus Verilog, 2008. URL http://iverilog.icarus.com/http://sourceforge.net/projects/iverilog/.

[162] Frank Winkler. Redesigning PAPI's High-Level API. Technical report, 2020.

[163] Thomas Wong. Autonomous Driving and Sensor Fusion SoCs - GSA - Global Semiconductor Alliance, 2019. URL https://www.gsaglobal.org/forums/autonomous-driving-and-sensor-fusion-socs/.

[164] World Health Organization. Road safety, 2024. URL https://www.who.int/health-topics/road-safety/.

[165] Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. Tigris: Architecture and algorithms for 3d perception in point clouds. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 629–642, 2019. ISBN 9781450369381. doi: 10.1145/3352460.3358259. URL http://horizon-lab.org.

[166] Ziyu Ying, Shulin Zhao, Sandeepa Bhuyan, Cyan Subhra Mishra, Mahmut T. Kandemir, and Chita R. Das. Pushing Point Cloud Compression to the Edge. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 282–299. IEEE, oct 2022. ISBN 978-1-6654-6272-3. doi: 10.1109/MICRO56248.2022.00031. URL https://ieeexplore.ieee.org/document/9923794/.

[167] Ziyu Ying, Sandeepa Bhuyan, Yan Kang, Yingtian Zhang, Mahmut T. Kandemir, and Chita R Das. EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–14, New York, NY, USA, jun 2023. ACM. ISBN 9798400700958. doi: 10.1145/3579371.3589113. URL https://dl.acm.org/doi/10.1145/3579371.3589113.

[168] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, volume 2020-Octob, pages 1067–1081. IEEE Computer Society, oct 2020. ISBN 9781728173832. doi: 10.1109/MICRO50266.2020.00089.

[169] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*, 8:58443–58469, 2020. ISSN 21693536. doi: 10.1109/ACCESS.2020.2983149.

[170] Dimitris Zermas, Izzat Izzat, and Nikolaos Papanikolopoulos. Fast segmentation of 3D point clouds: A paradigm on LiDAR data for autonomous vehicle applications. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5067–5073. IEEE, may 2017. ISBN 978-1-5090-4633-1. doi: 10.1109/ICRA.2017.7989591. URL https://www.researchgate.net/publication/318325507http://ieeexplore.ieee.org/document/7989591/.

[171] Fuquan Zhao, Hao Jiang, and Zongwei Liu. Recent development of automotive LiDAR technology, industry and trends. In Xudong Jiang and Jenq-Neng Hwang, editors, *Eleventh International Conference on Digital Image Processing (ICDIP 2019)*, volume 11179, page 178. SPIE, aug 2019. ISBN 9781510630758. doi: 10.1117/12.2540277. URL https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11179/2540277/Recent-development-of-automotive-LiDAR-technology-industry-and-trends/10.1117/12.2540277.full.

[172] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Li Erran Li, Tiancheng Lou, and Jishen Zhao. Driving Scenario Perception-Aware Computing System Design in Autonomous Vehicles. In *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, volume 2020-Octob, pages 88–95, 2020. ISBN 9781728197104. doi: 10.1109/ICCD50377.2020.00031.

[173] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A Modern Library for 3D Data Processing. 2018. URL http://www.open3d.http://arxiv.org/abs/1801.09847.

[174] Yuhao Zhu. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 76–89, New York, NY, USA, apr 2022. ACM. ISBN 9781450392044. doi: 10.1145/3503221.3508409. URL https://dl.acm.org/doi/10.1145/3503221.3508409.