# K-D Bonsai: ISA-Extensions to Compress K-D Trees for Autonomous Driving Tasks

Pedro H. E. Becker, José María Arnau, Antonio González
*Department of Computer Architecture*
*Universitat Politècnica de Catalunya*
{pedro, jarnau, antonio}@ac.upc.edu

***Abstract*—Autonomous Driving (AD) systems extensively manipulate 3D point clouds for object detection and vehicle localization. Thereby, efficient processing of 3D point clouds is crucial in these systems. In this work we propose *K-D Bonsai*, a technique to cut down memory usage during *radius search*, a critical building block of point cloud processing. *K-D Bonsai* exploits value similarity in the data structure that holds the point cloud (a k-d tree) to compress the data in memory. *K-D Bonsai* further compresses the data using a reduced floating-point representation, exploiting the physically limited range of point cloud values. For easy integration into nowadays systems, we implement *K-D Bonsai* through *Bonsai-extensions*, a small set of new CPU instructions to compress, decompress, and operate on points. To maintain baseline safety levels, we carefully craft the *Bonsai-extensions* 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. We evaluate *K-D Bonsai* over the euclidean cluster task of Autoware.ai, a state-of-the-art software stack for AD. We achieve an average of 9.26% improvement in end-to-end latency, 12.19% in tail latency, and a reduction of 10.84% in energy consumption. Differently from expensive accelerators proposed in related work, *K-D Bonsai* improves *radius search* with minimal area increase (0.36%).**

## I. Introduction

As recent advances in sensors, algorithms, and hardware crystallize the viability of Autonomous Driving (AD), concerns shift toward how to make these systems more efficient. In this context, improving hardware support for point cloud manipulation is critical since Autonomous Vehicles (AVs) heavily depend on point cloud-based algorithms [2], [24], [32]. Point clouds contain a 3D representation of the environment (see Figure 1), being richer than their 2D counterparts (e.g., images). For this reason, point clouds are suitable for a multitude of tasks, such as object detection, distance measurement, and vehicle localization, which are vital for AD.

A crucial point cloud operation performed by AD algorithms is *radius search*, whose goal is to return all points within a distance $r$ from a query point $q$ (where $q$ belongs to the point cloud). *Radius search* is used, for example, when clustering nearby points together (to infer shapes and objects around the vehicle) [26], [40], [48], or when optimizing the localization estimation of the vehicle [10], [11], [38], which are among the most time- and energy-consuming tasks performed by AVs [8], [59]. In fact, *radius search* accounts for more than half of the execution time of these tasks, as depicted in Figure 2 . Likewise, *radius search* is also used in 3D Convolutional Neural Networks (CNNs) processing [35], [47], in order to fetch neighbors of points to push them together through convolutions.

In this work we propose *K-D Bonsai*, a technique to compress point clouds to reduce data movement during *radius search* execution, improving its performance and energy efficiency. To perform the compression, we first observe that sensors have a physically limited range of operation, defining an upper-bound value for the coordinates of the collected points. For example, the Velodyne HDL-64E [56] - a typically employed LiDAR sensor - has a maximum operation range in the order of 120 m. This ultimately limits the values of the exponent fields in the Floating-Point (FP) representation of all points in the sensed point cloud. Second, we observe that k-d trees [9], [18], the typical data structure used for efficient point cloud searches (e.g., used by the prominent Point-Cloud Library (PCL) [49]), intrinsically group points with similar values in the tree leaves. As a consequence, the sign and exponent fields (in IEEE 754 FP representation [54]) are frequently repeated across different points in the same leaf 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 *radius search* without losing radius 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.

The mechanism is implemented with minor hardware and Instruction Set Architecture (ISA) extensions, which we named Bonsai-extensions, into a traditional CPU. We use the Bonsai-extensions to compress the k-d tree during build, and to read

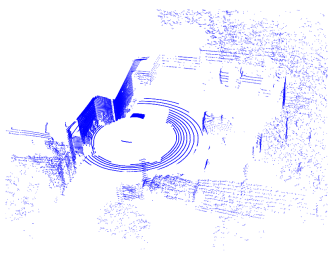


Fig. 1. A point-cloud obtained with LiDAR. Data from [23].

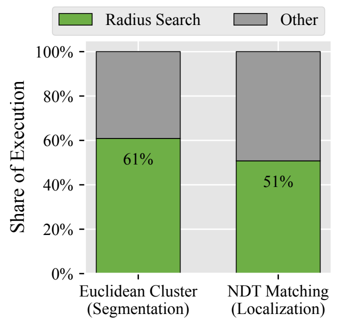


Fig. 2. *Radius search* execution time share in two Autoware.ai [6], [24] tasks.

and operate over compressed data during traversal. Since it reduces the number of necessary bytes to perform *radius search*, it reduces the number of memory accesses, energy consumption, and execution time. The CPU modifications are punctual, refining over already existing hardware. Moreover, our solution has minimal programmability impact, since improvements are exposed as new CPU instructions, and thus being straightforward to be used in existing applications. This makes it easy to adopt K-D Bonsai in today's systems, in contrast to expensive and hard-to-program out-of-core accelerators.

We validate our idea by extending ARM's AArch64 ISA on the gem5 simulator [12], [36]. We modify the PCL [49] to make use of our new instructions and use it to improve the execution of the euclidean cluster task on Autoware.ai [6], [24] - a state-of-the-art and open-source software stack for AVs. We demonstrate how *K-D Bonsai* effectively compress points, reducing the number of necessary load instructions by 23% and energy consumption by 10.84%, and improving end-to-end performance by 9.26% on average and tail-latency by 12.19%.

In summary, this paper presents the following contributions:

- We identify redundancy on bit-fields of FP representation in point cloud data stored in k-d trees.
- We verify that k-d tree radius 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 radius 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 *radius search*, improving performance and energy efficiency.
- We implement K-D Bonsai as 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.

The paper is organized as follows. Section II introduces important background concepts such as point clouds, k-d trees, and radius search. Section III explains how compression can be applied to k-d tree data. Section IV discusses the design details, including new instructions and necessary hardware. The results are analyzed in Section V. Finally, we review related work in Section VI and present final conclusions in Section VII.

## II. Background

In this section, we introduce important concepts to contextualize our work. We explain i) point clouds; ii) how they are used by modern AD software; iii) the k-d tree data structure, used to search on point cloud data; and iv) the *radius search* operation, used by different AD algorithms.

### A. Point Cloud for Autonomous Driving

A point cloud is a set of points in a given coordinate system. In the context of AD, point clouds are in the 3D space, where each point has coordinates ($x$, $y$, $z$). Point clouds can be obtained with sensors such as LiDAR, which sends laser beams around and measures the time for them to reflect back to the sensor [32]. In the absence of sensing noise, each point in a point cloud belongs to a surface in the real world (of a wall, a car, a tree, etc.) within the sensor range, as depicted in Figure 1.

Given the 3D information held by point clouds, they are commonly used by AD systems for perception and localization tasks [24], [32]. When used for perception, LiDAR-based algorithms serve to understand the surrounding environment. This includes tasks such as object detection and recognition, object tracking, and motion prediction [24], [32]. Typical perception algorithms that use point cloud include points clustering [26], [40], [48] and neural network classification [29], [46], [47]. When used for localization, LiDAR-based algorithms try to match the sensed point cloud with a previously existent point cloud map [51], sometimes referred as High-Definition (HD) map, in a process known as *registration* [10], [11], [38]. When the sensed point cloud overlaps an already mapped region, localization can be derived with centimeter-level precision.

### B. K-d Tree

Raw point cloud data obtained from sensors is unorganized. Points are usually pushed back to an array as they are collected by the sensor. For this reason, searching (a common operation across different LiDAR-based algorithms) exhaustively on raw point cloud data is prohibitive, especially in the context of AD where latency deadlines are strict [34]. The solution is to use a search-friendly data structure, such as a k-d tree [9].

A k-d tree is a binary tree that allows efficient search in $k$ dimensional data. In this work, we consider the k-d tree implemented by the widely used PCL [49] and FLANN [17] libraries, which are adopted by state-of-the-art AD software stacks such as Autoware.ai [6], [24] and Baidu Apollo [7]. The tree is created as follows. A root is created when all points are still to be sorted. On each level, starting with the root, the k-d tree selects one coordinate $\mathbf{c}$ (among $k$ possible) to split the data. In the 3D case $\mathbf{c}$ can be either the $x$, $y$, or $z$ coordinate. The median value in coordinate $\mathbf{c}$ across the set of points is found. Points with $\mathbf{c}$ value less than the median go to the left sub-tree, and points with $\mathbf{c}$ value greater than the median go to the right sub-tree. On each sub-tree, a new splitting coordinate is selected, and the process repeats. A common criterion (e.g., used by PCL) is to select the splitting coordinate whose data is more spread out. This helps posterior search traversal to quickly reach the nodes of interest.

Originally, each k-d tree node would hold a point [9] (e.g., the median point of the splitting coordinate). Later, an optimized k-d tree [18] proposed to store points on leaves only, up to a maximum $m$ number of points per leaf. Whenever a sub-tree contains less than $m$ points, the splitting process stops and the node is defined as a leaf. Restricting points to the leaves reduces the number of examinations while traversing the tree. The optimal number of points per leaf depends on the data and will impact the tree topology (e.g., the tree depth). PCL has a default of 15 points per leaf.

Notwithstanding, during the tree creation, each sub-tree has its bounding box calculated (i.e., the maximum and minimum values in all coordinates that can be found in that sub-tree). The parent node uses this information to hold its distance to each sub-tree, in the splitting coordinate. This will be used when searching, as we explain in the next subsection. Overall, non-leaf nodes on the tree serve to guide the tree traversal during the search, to reach leaves that contain a set of points that fits the search criteria.

### C. Radius Search

The main goal of *radius search* is to return all points within a distance $r$ from a query point $q$. Formally, given a three-dimensional point cloud $\mathscr{P} = \{p_1, p_2, ..., p_N\}$, $p_i \in \mathbb{R}^3$, we want to find the set of neighbor points

$\mathscr{N}(\mathbf{q}, r) = \{\mathbf{p} \in \mathscr{P} \mid dist(\mathbf{p}, \mathbf{q}) <= r\}$

of a query point $\mathbf{q} \in \mathscr{P}, \mathbb{R}^3$, within a distance $r \in \mathbb{R}$. The operation is used, for example, when clustering points from a point cloud, to retrieve the shape of objects in the environment. In that case, *radius search* is successively used to associate nearby points in clusters: e.g., if point $A$ is in the radius of point $B$, and point $B$ is in the radius of point $C$, then $A$, $B$, and $C$ are all parts of the same cluster [48].

To perform a radius search on a k-d tree one must provide a query point $q$ and the target radius $r$. The tree will be traversed comparing the splitting coordinate value of the current node with the correspondent coordinate of $q$. This comparison gives a best-effort hint of which child sub-tree is closer, and thus more likely to lead to a leaf where points within $r$ can be found. This descending process will lead to the leaf containing $q$ itself (along with other points in that leaf). When unwinding the tree navigation, the alternative sub-tree (not taken when descending) is also considered. If the distance in the splitting coordinate from $q$ to the sub-tree is smaller than $r$, the sub-tree is visited, and the descending continues. Every time the search finds a leaf, the distance between $q$ and each point $p_i$ on the leaf is calculated. The euclidean distance $d$ is generally used.

$$d(q, p_i) = \sqrt{(q_x - p_i x)^2 + (q_y - p_i y)^2 + (q_z - p_i z)^2} \quad (1)$$

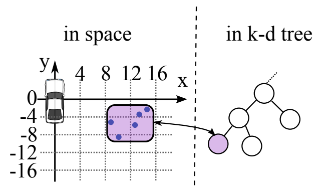
To avoid performing the square root, a common optimization is to calculate the squared euclidean distance.

$$d^2(q, p_i) = (q_x - p_i x)^2 + (q_y - p_i y)^2 + (q_z - p_i z)^2 \quad (2)$$

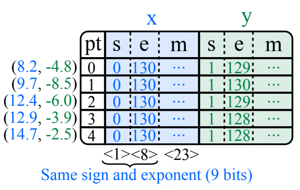
Then we can compare $d^2$ with the square radius $r^2$ to classify the point.

$$classification^2(q, p_i) = \begin{cases} in\ radius, & \text{if } d^2 <= r^2 \\ not\ in\ radius, & \text{if } d^2 > r^2 \end{cases} \quad (3)$$

Whenever $p_i$ is in the radius of $q$, it is added to the radius search result list.



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



| | 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 | ··· |

<1><8> <23>
Same sign and exponent (9 bits)

(b) Floating-point fields for each point.

Fig. 3. 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.

## III. 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 radius 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.

### A. Compression based on value similarity

When the k-d tree is built (as explained in section II), 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 3, in two dimensions for simplicity.

Figure 3a 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 3b lists the coordinates of the points ($x$ and $y$ in this example), exposing their internal FP representation (in 32-bit IEEE 754 [54]). We depict the sign ($s$), exponent ($e$), and mantissa ($m$) fields of FP representation separately. Following the IEEE 754 standard, the stored value is given by the following equation.

$$value = -1^{sign} \times 1.mantissa \times 2^{exponent-bias} \quad (4)$$

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 3 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 3a). We inspected a set of point clouds spanning more than 37 million points that feed the *euclidean cluster* node in Autoware.ai [6], [24] (details about data-set can be found in Section V). We identified that 78% of leaf

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

TABLE I
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% |

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 IV).

## B. Compression via a smaller representation

Compressing the *sign* and *exponent* of FP representation fields (Section III-A) 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 I depicts the error in classification (Eq. 3) using different FP formats with less than 32-bits. We use the same set of point clouds as in Section III-A. We experimented with two common 16-bit FP representations: *IEEE-754 16-bit* (IEEE half-precision format [54]), the *bfloat 16* (used for machine learning applications, and e.g., supported by CUDA [43]); 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 [56] (a typically employed LiDAR sensor) has a maximum cover range of 120 m. Indeed, none of the errors depicted in Table I 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].

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 [5]) 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 III-A).

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.

## C. 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 [22] and pose consequences which are hard to test [27]. 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 $\delta B$ associated with the loss of precision. Let $B'$ denote the resulting value of $B$ in 16-bit representation.

$$B' = B + \delta B \tag{5}$$

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 III-B), the rounding in the mantissa is the single source of error. In 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.

[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 [51], [52] are created, several point clouds are combined to represent a region. Hence, 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.

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 (Eq. 4). In these conditions, the maximum error $\delta$ for rounding a number $B$ when converting it from 32-bit to 16-bit IEEE-754 FP is given by:

$$max(\delta B) = 2^{exponent-bias} \times \frac{2^{-10}}{2} = 2^{exponent-bias} \times 2^{-11} \quad (6)$$

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 Eq. 5.

$$A - B' \quad = \quad (A) - (B + \delta B) \quad = \quad (A - B) - \delta B \quad (7)$$

Where $-\delta B$ is the associated error. We can proceed and evaluate the error for the square operation $(A-B')^2$ applying Eq. 5, Eq. 7, and Newton's binomial theorem.

$$\begin{aligned}(A-B')^2 &= [(A-B)-\delta B]^2 \\ &= (A-B)^2 - 2(A-B)\delta B + \delta B^2 \\ &= (A-B)^2 - 2[A-(B'-\delta B)]\delta B + \delta B^2 \\ &= (A-B)^2 - 2(A-B'+\delta B)\delta B + \delta B^2 \\ &= (A-B)^2 - 2[(A-B')\delta B + \delta B^2] + \delta B^2 \\ &= (A-B)^2 - 2(A-B')\delta B - 2\delta B^2 + \delta B^2 \\ &= (A-B)^2 - 2(A-B')\delta B - \delta B^2\end{aligned} \quad (8)$$

Where $-2(A-B')\delta B - \delta B^2$ is the associated error of the square of the differences operation ($\varepsilon_{sd}$). Notice that $\delta B$ 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 $\varepsilon_{sd}$, using the $max(\delta B)$ (Eq. 6) instead of $\delta B$.

$$max(\varepsilon_{sd}) \quad = \quad 2 \cdot |A - B'| \cdot |max(\delta B)| + max(\delta B)^2 \quad (9)$$

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 $d'^2$.

$$d'^2(q, p'_i) = (q_x - p'_i x)^2 + (q_y - p'_i y)^2 + (q_z - p'_i z)^2 \quad (10)$$

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

$$T\varepsilon_{sd} = max(\varepsilon_{sd})_x + max(\varepsilon_{sd})_y + max(\varepsilon_{sd})_z \quad (11)$$

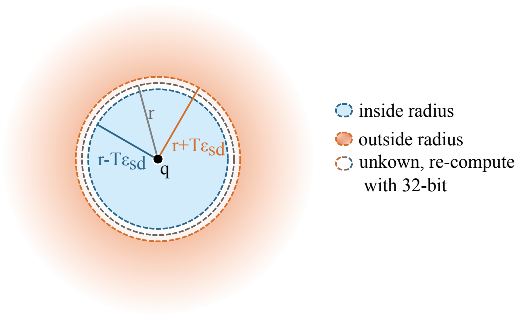


Fig. 4. Visual representation of Equation 12.

We can finally use Eqs. 10 and 11 to perform the classification (with $p'_i$ instead of $p_i$).

$$classification'^2(q, p'_i) = \begin{cases} in\ radius, & \text{if } d'^2 <= r^2 - T\varepsilon_{sd} \\ not\ in\ radius, & \text{if } d'^2 > r^2 + T\varepsilon_{sd} \\ use\ Eq.\ 3, & \text{otherwise} \end{cases} \quad (12)$$

In other words, we can use the worst-case error $T\varepsilon_{sd}$ to confirm the correctness of the classification with $p'_i$. We do so by defining a shell around $r^2$ with values $r^2 - T\varepsilon_{sd}$ and $r^2 + T\varepsilon_{sd}$, as depicted in Figure 4. Whenever $d'^2$ falls outside the shell, the classification is the same as the baseline, computed by Eq. 3. For instance, a point inside the radius but outside the shell cannot be outside the radius even if we add $T\varepsilon_{sd}$ 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 Eq. 3.

## IV. 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*.

### A. Hardware support for k-d tree compression

After deriving a compression scheme (Section III), hereby referred to as *K-D Bonsai*, it is of our interest to use it in tasks that perform *radius search*. A naïve 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 and experimentation platform in Section V-A), 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 IV-B). Thus, using proper

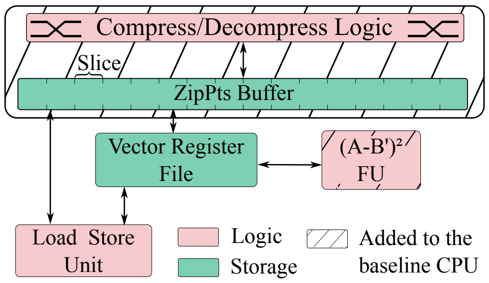


Fig. 5. The new components added to the baseline CPU and how they interact with pre-existing ones.

hardware inside the CPU to perform (de)compression and classify points avoids communication costs [53]. Alternatively, (de)compression operations could be coalesced to amortize communication costs. However, accelerators are likely to be more expensive (see Section VI). At the same time, leaf processing is only a fraction of the point cloud handling, limiting the maximum performance improvement (Ahmdal's law), and jeopardizing accelerator adoption. Nevertheless, industry favors less experimental approaches to accelerate tasks in their real-life solutions, rarely employing accelerators [44].

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 [4]. Also, some ARM processors support to-be-defined custom instructions [13]. 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 [57]. This set of reasons motivates us to propose specific instructions in the CPU to implement *K-D Bonsai* effectively.

## B. Changing the CPU

A main advantage of the ideas discussed in Section III 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 $(A-B')^2$ (Eq. 8).

Figure 5 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.

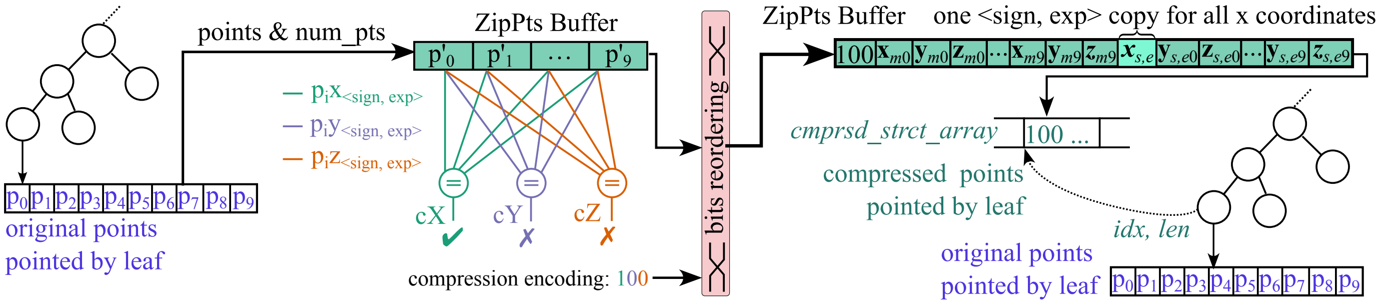
The buffer has two 128-bit ports to interface with the Vector Register File and one 128-bit port to interface with the Load Store Unit. Hence, data is exchanged in chunks of 128-bit, which we refer to as a *ZipPts Buffer slice*. When less than 128-bits must be transferred (e.g., the last chunk of a compressed data), we pad data with zeroes. The width of the ports equals the ones that already exist in our baseline CPU (see Section V), for example in the Vector Register File. Hence, we can load and store data from/to memory directly to the 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 6. 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 6 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 Eq. 8, and can be used successive times (for each coordinate) to compute Eq. 10 and 11 to perform the classification. Figure 7 details the internal scheme of the FU. It has two input operands, 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 ($max(\varepsilon_{sd})$, Eq. 9) has more operations than the square of the differences itself. Fortunately, we can take advantage of some observations to



points & num_pts
ZipPts Buffer
p'0 p'1 ··· p'9
— $p_ix_{<sign, exp>}$
— $p_iy_{<sign, exp>}$
— $p_iz_{<sign, exp>}$
cX ✓ cY ✗ cZ ✗
original points pointed by leaf
compression encoding: 100
bits reordering
ZipPts Buffer one <sign, exp> copy for all x coordinates
cmprsd_strct_array 100 ...
compressed points pointed by leaf
idx, len
original points pointed by leaf

Fig. 6. 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.

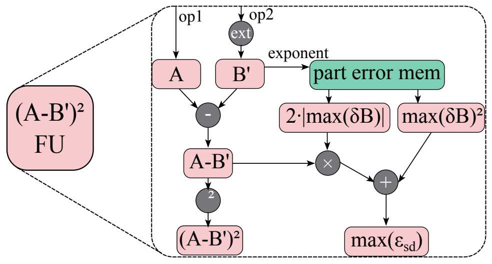


Fig. 7. Details of the FU for square of the difference with error computation.

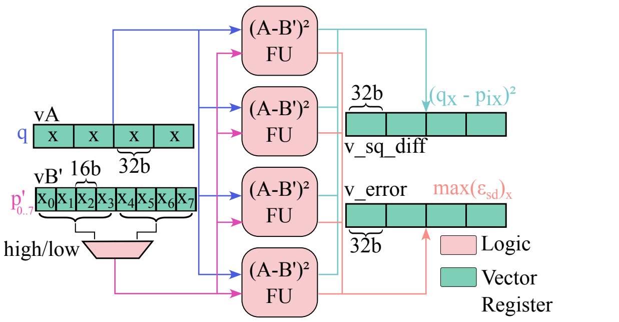


Fig. 8. Vector square of the differences FUs.

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 7) 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 $max(\varepsilon_{sd})$.

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 8), 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 radius search classification, a coordinate of the query $q$ is loaded into all indices of $vA$, while the same coordinate of multiple points is loaded into $vB'$.

### C. Software impact

Now we discuss how to use the new hardware from the software. We expose the new hardware functionalities mentioned in Section IV-B as new CPU instructions. The set of new instructions, which we refer to as *Bonsai-extensions*, is described in Table II. We 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 6). 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 instruction

TABLE II
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 III-A). 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. 8). 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 7 and 8). 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'*. |

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 8). 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 V-A), but *eight* values on each coordinate (16-bit computed in 32-bit in the FUs, Figure 7), 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 8). 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 ($max(\varepsilon_{sd})$). 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 (Eq. 12). If the result is inconclusive (inside the white shell in Figure 4), 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.

## V. RESULTS

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

### A. Evaluation Methodology

From the software perspective, we rely on Autoware.ai [24] 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 [6]. It has several algorithms to perform AD, from sensor processing and perception to actuation. In this work, we choose a representative algorithm from Autoware, namely euclidean cluster [48] 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 [38]).

TABLE III
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% |

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 [8]. 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 [23]. 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 [3] 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 III details sub-sampling errors, evidencing it as a fast and accurate proxy to the code behavior.

We implemented the Bonsai-extensions (Table II) in the gem5 simulator [12], [36], targeting an Out-of-Order (OoO) CPU with the ARM's AArch64 ISA. We base our model (see Table IV) 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 [42]). We modified the PCL [33] version 1.10 and its auxiliary library FLANN [17] version 1.9.2, using our instructions during the *radius search*, as explained in Section IV-C. 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.

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 [20], [50] to reach the regions of the sub-sampled frames. For energy results, we model the CPU in McPAT [28], [31] 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 [14] with a 14 nm technology [37].

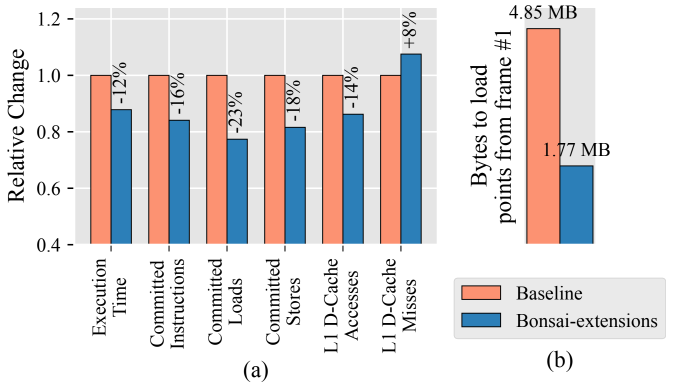


Fig. 9. (a) Hardware metrics during the execution of the *extract* kernel of euclidean clustering considering the baseline code and the proposed Bonsai-extensions. Average across all executed frames. (b) Number of loaded bytes to fetch points from the first frame of the data-set during *radius search* (traversal).

To unify results in a single technology we scale the baseline CPU data reported by McPAT using the methodology described by Stillmaker et al. [55] (from 32 nm to 14 nm technology).

## B. Performance Analysis

Figure 9a 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 [39]), 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 9b 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 increases efficiency in the use of the CPU. Third, due to

TABLE IV
BASELINE CPU MODEL USED

| 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 |

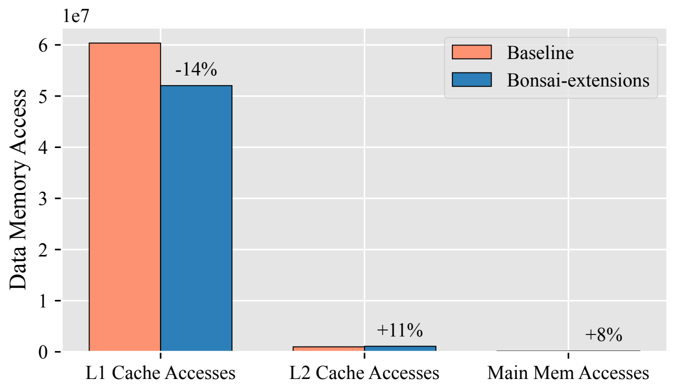


Fig. 10. Accesses on different levels of the memory hierarchy.

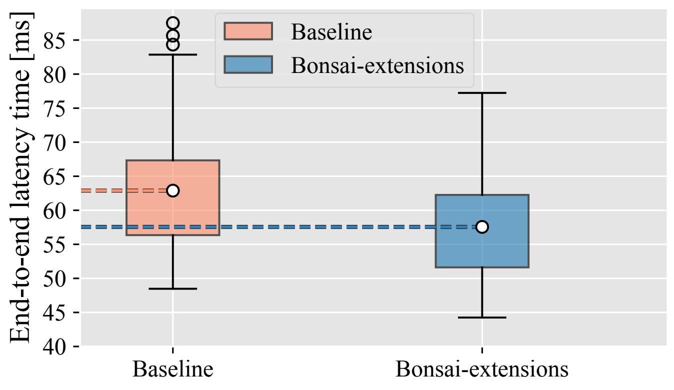


Fig. 11. The distribution of the end-to-end latencies for the euclidean cluster algorithm. The dashed line indicates the mean value. Half the values are within the box limits.

TABLE V
AREA AND POWER FOR BASELINE CPU AND K-D BONSAI

| | | Area (mm²) | 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% |

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 [34]. Nevertheless, this is particularly significant when we observe that benefits come from the addition of only five new instructions to the ISA.

Figure 9a 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). 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 10 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 III-B, Table I, 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 clusters must also be performed. Figure 11 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 III-C). Also, since the euclidean cluster is generally a perception bottleneck [8], [25], [59] K-D Bonsai improvements are directly converted into overall AD improvements.

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.

### C. *Area and Power Analysis*

Let us now examine the hardware costs of implementing our technique. Table V presents area and power overheads introduced to support K-D Bonsai, according to the methodology explained in V-A. Overall, the hardware to support the new instructions is simple, increasing area by 0.051 mm², 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 are particularly important for meeting cooling constraints [34] and design of small autonomous vehicles (e.g., for delivery [41]), respectively.

### D. *Energy Analysis*

Finally, we go through K-D Bonsai energy consumption results in the *extract* kernel of the euclidean cluster. Figure 12

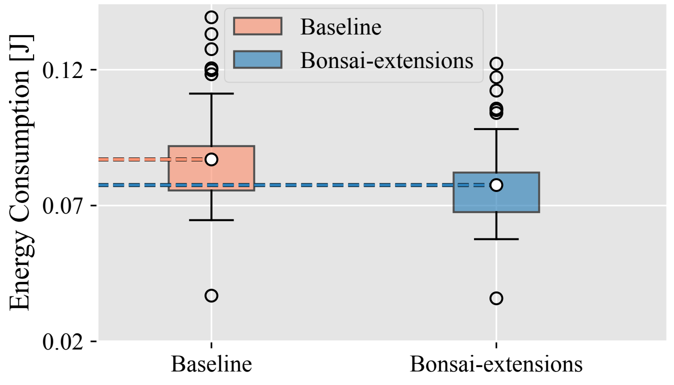


Fig. 12. The distribution of the energy consumption for the *extract* kernel in the euclidean cluster algorithm. The dashed line indicates the mean value. Half the values are within the box limits.

depicts a box-plot (in the same fashion we did for end-to-end latency, Figure 11). The reduction in energy consumption is driven by a reduction in execution time, number of instructions and number of memory accesses, which pays off the small increase in dynamic power (Table V). 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 [34] (e.g., on battery-powered vehicles).

## VI. Related Work

Recent advances in LiDAR technology and AV motivated previous works on improving point cloud processing. Heinzle et al. propose specific hardware to improve radius and nearest-neighbor search in k-d tree point clouds [21]. Their main idea is to search slightly more points than necessary (e.g., asking for a larger radius), and use the extended result-set for subsequent, spatially closed queries. They implement it in an FPGA and, compared to a CPU, it improves query throughput by 68%, *ignoring* CPU-FPGA transfer costs. The work claims the used platform lacks an efficient CPU-FPGA interface, resulting in *half* of the baseline CPU performance if communication costs are taken into account. This highlights an important challenge for accelerator integration. Since K-D Bonsai is implemented as part of the CPU, there is no overhead to transfer data in and out of the core. Also, while their work focuses on speeding up the traversal, K-D Bonsai improves *leaf processing*, reducing the total number of loads to bring the data (via (de)compression in the CPU) after the traversal is performed, hence being orthogonal to their technique.

A more recent work introduces Tigris [58], an accelerator to speed up radius and nearest-neighbor search for point cloud registration (a major application for k-d tree search, see Section I). Tigris divides traversal and leaf processing in a front-end/back-end fashion. Multiple queries are traversed in parallel, offloading leaf data to the back-end, where multiple FUs will perform distance checks. To exploit higher performance, Tigris also has a scheme to search on previously obtained result-sets, causing their search to be *approximate*. The accelerator improves end-to-end latency for registration w.r.t a CPU by 86.6% but requires a total area of 15.57 mm$^2$ (more than our Baseline CPU, see Table V). The work does not report the cost to offload queries to the accelerator. QuickNN [45] also accelerates nearest-neighbor search on k-d tree-based point clouds. In their target application, point cloud frames are used as references for new frames. They exploit this behavior in the accelerator architecture, overlapping execution and sharing data of tree build and tree traversal. Moreover, they propose a gather-read and gather-write cache, coalescing accesses to off-chip memory. Like Tigris, their work processes multiple queries in parallel and performs an approximate search. Accelerators for nearest-neighbor search on high-dimensional spaces were also proposed [1], [30], but the problem properties and requirements differ from 3D data used by AD.

The Mesorasi [16] accelerator for point cloud based CNNs (e.g. PointNet++ [47]) proposes delayed-aggregation, allowing neighbor search and feature computation to be overlapped in time, hiding latency. In their proposal, point cloud search time stays roughly the same, and most of the benefits come from faster feature computation. A more recent work [15] improves over Mesorasi by limiting the backtracking step of the k-d tree search to a sub-tree at the cost of accuracy. Additionally, in case of bank conflicts their solution re-uses similar points or completely ignores traversal paths if necessary. This approximate scheme has most of its accuracy corrected during training, restricting it to machine learning scenarios. Similarly to Mesorasi, PointAcc [35] also accelerates point cloud-based CNNs, proposing a ranking-based generic accelerator unit. They also increase the number of mapping operations over Mesorasi [16], e.g., supporting *radius search*, and farthest point sampling to fetch inputs in the point cloud.

Some works exploited GPUs to improve point cloud processing. The Buffer k-d tree [19] proposes nearest-neighbor search using a buffer to delay the processing of queries of the same leaf until enough work is gathered. RTNN [60] 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. [40] focuses 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 clustering performs similarly to an OoO CPU due to the GPU communication overheads [25]. Indeed, Autoware.ai uses the CPU instead of the GPU to run the euclidean cluster by default [6]. The GPU is reserved for image-based object detection CNNs where its benefits are much less debatable [25]. At the same time, adding a GPU just for point cloud processing incurs orders of magnitude more area and power overheads than *K-D Bonsai*, both critical aspects to be minimized in AVs [34].

In summary, the majority of previous works propose the use of accelerators, implying communication overheads, high area costs, and lack of programmability. Moreover, many works introduce approximation to achieve effective solutions. On the other hand, *K-D Bonsai* proposes a small set of new instructions implemented inside a CPU, while also guaranteeing baseline accuracy. This fundamental difference places K-D Bonsai as a programmable and easy-to-be-adopted solution in nowadays systems. Our solution is modest performance-wise, but orders of magnitude cheaper regarding area and power. Nevertheless, our (de)compression scheme aims for better data fetching for leaf processing, an orthogonal concept unexploited by previous works. The techniques presented here, we recall, are also applicable in platforms other than a CPU, as it only depends on the algorithm (*radius search*) and the input (*point clouds*).

## VII. Conclusions

In this work, 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. Differently 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 cycle-accurate simulator in full-system mode. Our solution, K-D Bonsai, proved to be 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.

## VIII. 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 grant PID2020-113172RB-I00, the ICREA Academia program, and a FPI-UPC 2020 grant.

## References

[1] A. M. Abdelhadi, C. S. Bouganis, and G. A. Constantinides, "Accelerated approximate nearest neighbors search through hierarchical product quantization," in *Proceedings - 2019 International Conference on Field-Programmable Technology, ICFPT 2019*, vol. 2019-Decem, 2019, pp. 90–98.

[2] ApolloAuto/Apollo, "An open autonomous driving platform," 2021. [Online]. Available: https://github.com/ApolloAuto/apollo

[3] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using Time-Based Sampling," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2013, pp. 448–459.

[4] ARM, "Understanding the Armv8.x extensions," p. 15, 2019.

[5] ARM, "ARM Architecture Reference Manual - Armv8, for A-profile architecture," pp. 1–1138, 2021.

[6] Autoware, "Autoware.AI · GitHub," 2023. [Online]. Available: https://github.com/Autoware-AI

[7] Baidu, "Apollo." [Online]. Available: http://apollo.auto/

[8] P. H. E. Becker, J. M. Arnau, and A. Gonzalez, "Demystifying Power and Performance Bottlenecks in Autonomous Driving Systems," in *Proceedings - 2020 IEEE International Symposium on Workload Characterization, IISWC 2020*. IEEE, oct 2020, pp. 205–215. [Online]. Available: https://ieeexplore.ieee.org/document/9251251/

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, sep 1975. [Online]. Available: https://dl.acm.org/doi/10.1145/361002.361007

[10] P. J. Besl and N. D. McKay, "A Method for Registration of 3-D Shapes," in *Sensor Fusion IV: Control Paradigms and Data Structures*, P. S. Schenker, Ed., vol. 1611, apr 1992, pp. 586–606. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=981454

[11] P. Biber, "The Normal Distributions Transform: A New Approach to Laser Scan Matching," in *IEEE International Conference on Intelligent Robots and Systems*, vol. 3. IEEE, 2003, pp. 2743–2748. [Online]. Available: https://www.researchgate.net/publication/4045903http://ieeexplore.ieee.org/document/1249285/

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[13] L. Choquin and S. I. Developer, "Arm Custom Instructions: Enabling Innovation and Greater Flexibility on Arm," Tech. Rep. February, 2020. [Online]. Available: https://developer.arm.com/documentation/102891/0100

[14] D. Compiler, "Design Compiler." [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[15] Y. Feng, G. Hammonds, Y. Gan, and Y. Zhu, "Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics," in *Proceedings - International Symposium on Computer Architecture*, vol. 1. ACM, 2022, pp. 962–977. [Online]. Available: https://doi.org/10.1145/3470496.3527395

[16] Y. Feng, B. Tian, T. Xu, P. Whatmough, and Y. Zhu, "Mesorasi: Architecture support for point cloud analytics via delayed-aggregation," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2020-Octob, 2020, pp. 1037–1050. [Online]. Available: https://github.com/horizon-research/efficient-deep-

[17] FLANN, "FLANN - Fast Library for Approximate Nearest Neighbors," 2014. [Online]. Available: http://www.cs.ubc.ca/research/flann/

[18] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, sep 1977. [Online]. Available: https://dl.acm.org/doi/10.1145/355744.355745

[19] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d trees: Processing massive nearest neighbor queries on GPUs," in *31st International Conference on Machine Learning, ICML 2014*, vol. 1, 2014, pp. 312–320.

[20] I. Habib, "Virtualization with KVM," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[21] S. Heinzle, G. Guennebaud, M. Botsch, and M. Gross, "A hardware processing unit for point sets," in *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2008, pp. 21–31.

[22] R. Hussain and S. Zeadally, "Autonomous Cars: Research Results, Issues, and Future Challenges," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1275–1313, 2019.

[23] T. IV, "Autoware Data." [Online]. Available: https://data.tier4.jp/

[24] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[25] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems," *Proceedings - 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018*, no. April, pp. 287–296, 2018.

[26] K. Klasing, D. Wollherr, and M. Buss, "A clustering method for efficient segmentation of 3D laser data," in *2008 IEEE International Conference on Robotics and Automation*. IEEE, may 2008, pp. 4043–4048. [Online]. Available: http://ieeexplore.ieee.org/document/4543832/

[27] P. Koopman and M. Wagner, "Challenges in Autonomous Vehicle Testing and Validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[28] H. H. Labs JUNG AHN, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. ACM Trans," *Archit. Code Optim*, vol. 10, no. 5, 2013. [Online]. Available: http://dx.doi.org/10.1145/2445572.2445577

[29] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, "Pointpillars: Fast encoders for object detection from point clouds," Tech. Rep., 2019. [Online]. Available: https://github.com/nutonomy/second.pytorch

[30] Y. Lee, H. Choi, S. Min, H. Lee, S. Beak, D. Jeong, J. W. Lee, and T. J. Ham, "ANNA: Specialized Architecture for Approximate Nearest Neighbor Search," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, apr 2022, pp. 169–183. [Online]. Available: https://ieeexplore.ieee.org/document/9773206/

[31] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing," *Transactions on Architecture and Code Optimization*, vol. 10, no. 1, 2013.

[32] Y. Li and J. Ibanez-Guzman, "Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems," *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, jul 2020. [Online]. Available: http://www.hesaitech.com/en/autonomous_driving.htmlhttps://ieeexplore.ieee.org/document/9127855/

[33] P. C. 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. [Online]. Available: https://pointclouds.org/

[34] S. C. Lin, Y. Zhang, C. H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 751–766, 2018.

[35] Y. Lin, Z. Zhang, H. Tang, H. Wang, and S. Han, "PointAcc: Efficient point cloud accelerator," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 13. ACM, 2021, pp. 449–461. [Online]. Available: https://doi.org/10.1145/3466752.3480084

[36] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 Simulator: Version 20.0+," Tech. Rep., 2020. [Online]. Available: http://arxiv.org/abs/2007.03152

[37] V. Melikyan, M. Martirosyan, A. Melikyan, and G. Piliposyan, "14nm educational design kit: Capabilities deployment and future," in *Small Systems Simulation Symposium*, 2018.

[38] H. Merten, *The Three-Dimensional Normal-Distributions Transform — an Efficient Representation for Registration, Surface Analysis, and Loop Detection*. Örebro: Örebro universitet, 2008, vol. 10, no. May. [Online]. Available: http://www.aass.oru.se/Research/Learning/publications/2009/Magnusson_2009-Doctoral_Thesis-3D_NDT.pdf

[39] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[40] A. Nguyen, A. M. Cano, M. Edahiro, and S. Kato, "Fast euclidean cluster extraction using GPUS," *Journal of Robotics and Mechatronics*, vol. 32, no. 3, pp. 548–560, 2020.

[41] Nuro, "Delivering Safety: Nuro's Approach," Tech. Rep., 2018.

[42] Nvidia, "NVIDIA DRIVE Overview," 2019. [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/

[43] NVIDIA, "CUDA Math API API Reference Manual," 2022. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Math_API.pdf

[44] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *Journal of Systems Architecture*, vol. 129, p. 102561, aug 2022. [Online]. Available: https://doi.org/10.1016/j.sysarc.2022.102561https://linkinghub.elsevier.com/retrieve/pii/S1383762122001138

[45] R. Pinkham, S. Zeng, and Z. 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*. Institute of Electrical and Electronics Engineers Inc., feb 2020, pp. 180–192.

[46] C. R. Qi, H. Su, K. Mo, and L. 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*, vol. 2017-Janua, 2017, pp. 77–85.

[47] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "PointNet++: Deep hierarchical feature learning on point sets in a metric space," in *Advances in Neural Information Processing Systems*, vol. 2017-Decem, 2017, pp. 5100–5109.

[48] R. B. Rusu, "Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments," *KI - Kunstliche Intelligenz*, vol. 24, no. 4, pp. 345–348, 2010.

[49] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *2011 IEEE International Conference on Robotics and Automation*, vol. 74, no. 3. IEEE, may 2011, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/5980567/

[50] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *Proceedings - 2015 IEEE International Symposium on Workload Characterization, IISWC 2015*. Institute of Electrical and Electronics Engineers Inc., oct 2015, pp. 183–192.

[51] B. Schwarz, "Mapping the world in 3D," *Nature Photonics*, vol. 4, no. 7, pp. 429–430, jul 2010. [Online]. Available: http://www.nature.com/articles/nphoton.2010.148

[52] H. G. Seif and X. Hu, "Autonomous Driving in the iCity—HD Maps as a Key Challenge of the Automotive Industry," *Engineering*, vol. 2, no. 2, pp. 159–162, 2016. [Online]. Available: http://dx.doi.org/10.1016/J.ENG.2016.02.010

[53] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 2016-Decem. IEEE, oct 2016, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/7783751/

[54] I. C. Society, *IEEE Std 754™-2008 (Revision of IEEE Std 754-1985), IEEE Standard for Floating-Point Arithmetic*, 2008, vol. 2008, no. August. [Online]. Available: https://ieeexplore.ieee.org/document/4610935

[55] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, jun 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167926017300755

[56] I. Velodyne Lidar, "Product guide." Tech. Rep.

[57] A. Waterman and K. Asanovic, "The risc-v instruction set manual," *RISC-V Foundation*, vol. I, 2017. [Online]. Available: https://riscv.org/technical/specifications/

[58] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2019, pp. 629–642. [Online]. Available: http://horizon-lab.org

[59] H. Zhao, Y. Zhang, P. Meng, H. Shi, L. E. Li, T. Lou, and J. Zhao, "Driving Scenario Perception-Aware Computing System Design in Autonomous Vehicles," in *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, vol. 2020-Octob, 2020, pp. 88–95.

[60] Y. Zhu, "Rtnn: Accelerating neighbor search using hardware ray tracing," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*. New York, NY, USA: ACM, apr 2022, pp. 76–89. [Online]. Available: https://dl.acm.org/doi/10.1145/3503221.3508409