Alpha Demo: A Hardware-Accelerated Data Model for Ad-Hoc Manipulation of Point Clouds

Sophie Pfister
sophie.pfister@unifr.ch
University of Fribourg
Switzerland

Alberto Lerner
alberto.lerner@unifr.ch
University of Fribourg
Switzerland

Abishek Ramdas
abishek.ramdas@unifr.ch
University of Fribourg
Switzerland

Philippe Cudré-Mauroux
pcm@unifr.ch
University of Fribourg
Switzerland

Abstract

A point cloud is a collection of data points in a three-dimensional space that represents the external surface of an object or environment. This data type is typically generated by spatial sensing devices, like LiDAR (Light Detection and Ranging) systems. While these devices have revolutionized fields such as autonomous vehicles, urban traffic monitoring, and forestry management, the manipulation of point clouds still poses challenges. Applications often involve complex data manipulation and, in some cases, require very low latency. As a result, each application tends to develop its own methods for handling point clouds based on its unique requirements. The common belief is that only through specialized solutions can an application's requirements be met.

In this paper, we demonstrate that complex point cloud manipulations can be developed using generic primitives without compromising latency. To do so, we use a timeless data management approach: We establish a data model for point clouds. The model supports *ad-hoc* querying and transformations simply by combining the powerful operators we introduce. Along with the model, we also present a specially designed FPGA-based hardware unit that is capable of accelerating some of its critical operators. We call our data model Damocles, the hardware that supports it Palicus, and the system built on top of them Alpha. Our system works on an actual LiDAR sensor connected to Alpha and can run even on an affordable edge-sized device with no noticeable latency.

ACM Reference Format:
Sophie Pfister, Alberto Lerner, Abishek Ramdas, and Philippe Cudré-Mauroux. 2025. Alpha Demo: A Hardware-Accelerated Data Model for Ad-Hoc Manipulation of Point Clouds. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725078

1 Background and Motivation

From Lasers to Point Clouds. LiDAR systems are environment sensing devices [3, 6]. They are similar in spirit to radars, but while the latter use echolocation methods, LiDARs use laser sources that bounce off surfaces. A typical arrangement is to place several laser

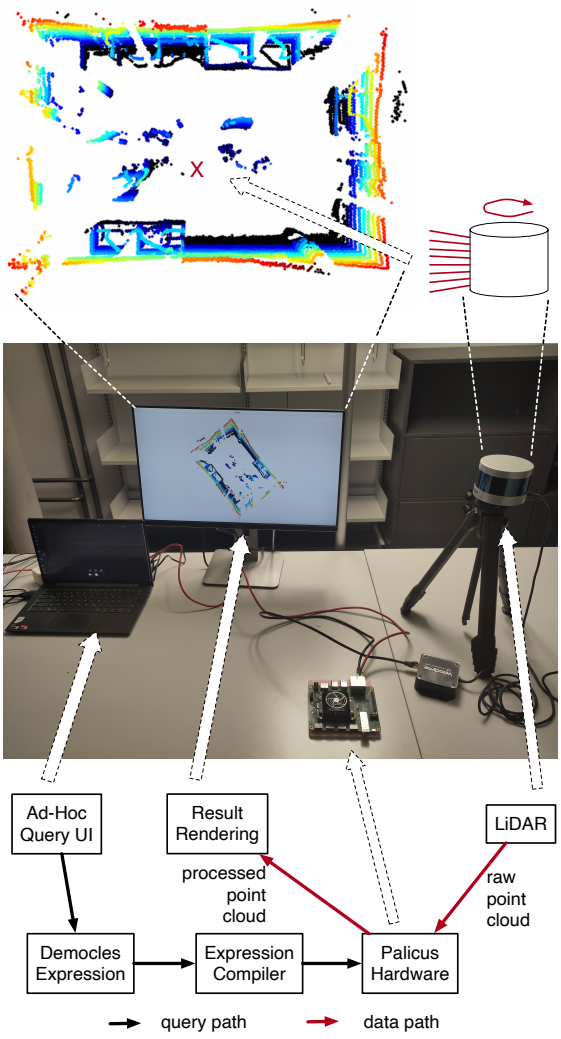

Figure 1: (top) Rendition of Sophie's office point cloud: The manipulation simply changes the vantage point—it renders the room from above—and uses a color scale for the z-axis. Note the 'X' mark where the LiDAR (detailed on the left) is placed. (center) Demonstration setup: We provide a LiDAR sensor, a Palicus prototype, and a running instance of Alpha for data visualization and hardware configuration. (bottom) High-level view of Alpha: The user writes queries in an intuitive way, which are then emitted as Damocles expressions executed by the programmable hardware.

SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725078

sources as an array on the side of a rotating cylinder. Figure 1 (center) depicts such a device and Figure 1 (top) shows a representation of its internal mechanism. The laser sources in this device are not positioned perpendicularly to the surface. Instead, the top sources are tilted upward at increasing angles, while the bottom sources are tilted downward. As a result, the device's laser beams can hit any object or surface within a 360-degree band around it.

Internally, a LiDAR calculates the position of the reflection point based on the round-trip time of the laser pulse and its shooting direction. In our case, the LiDAR also registers the relative intensity of the reflected beam. This information—the 3D coordinates and the reflectance of each point—is output in the same order that lasers were emitted. The resulting stream is known as a dynamic **point cloud**. In practice, it is common to transmit a raw version of a point cloud over a network connection by grouping the points into packets and sending them in a continuous stream.

When an application receives this stream, it first converts the network packets back into a logical representation of the point cloud. After this conversion, the application typically segments the points into static **frames**, where each frame represents a single revolution of the laser array. To accurately represent movements on the scene, the application must perform these conversions with low and consistent latency. Otherwise, the timing of the movements in the rendered image may differ from their actual occurrence. This discrepancy can be problematic for certain applications.

These frames can be rendered using different methods. For example, the image in Figure 1 (top) shows an actual frame depicting the office where the device was located. The 3D view alters the perspective of the point cloud and uses colors to represent the z-axis coordinates of the generated points. Points located below the LiDAR's vantage point are displayed in dark blue, while higher points are represented in red, with gradients indicating intermediate heights. If an object moves, the LiDAR detects it almost instantly due to the high rotational speed of its laser array. In turn, the visualization will show the object's new position, changing the color representing the z-axis, if necessary.

Processing LiDAR-Generated Point Clouds. Besides rendering, applications are often responsible for extracting actionable information from a point cloud, such as detecting obstacles in traffic scenarios or mapping canopy density in forestry management scenarios. However, processing point clouds is challenging due to their sparsity, lack of structure, and high number of points [5]. Therefore, a typical application would start by focusing on a **Region of Interest** (as illustrated in Figure 2, which we will explain shortly) or by creating a more suitable representation of the data (ditto Figure 3).

A skilled programmer can hard-code these data manipulations, but such an approach is time-consuming and cumbersome. Changes in application requirements or upgrades to the used devices can lead to code rewrites, making maintenance and evolution difficult. Moreover, hard-coded applications hinder the ability to share common manipulation techniques across different use-cases because common processing steps are not easily identifiable.

A Point Cloud Data Model and User Experience. In this paper, we present a simpler and more efficient approach to manipulating point clouds. We propose a data model called Damocles, which

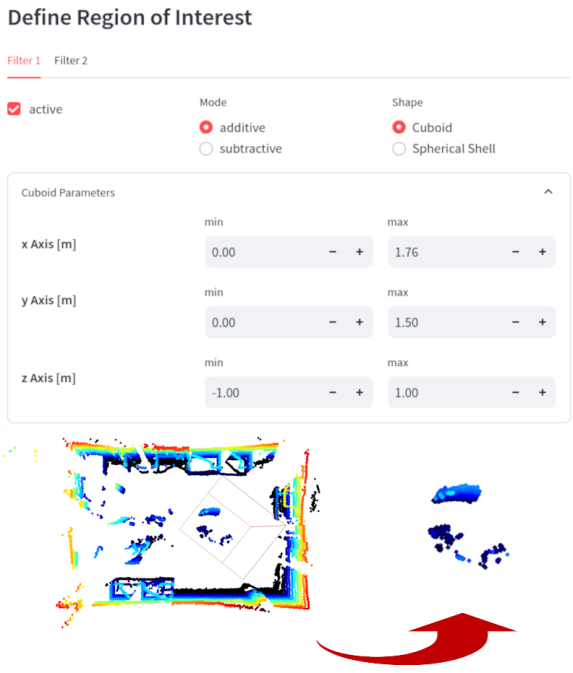

Figure 2: (top) Query-by-example form to express a simple region of interest. The form passes the arguments directly to Alpha, which then filters out the irrelevant points using the Palicus hardware. (bottom) The effect of applying a cuboid RoI filter on the raw point cloud is seen immediately, including if objects move in or out of the RoI.

allows programmers to describe manipulations as expressions over the model rather than programming them. In essence, Damocles does for point clouds what the relational model does for tabular data. It provides a unified representation for point clouds across different processing stages and introduces a set of simple yet powerful operations that can be combined through composition. As a result, Damocles supports complex manipulations that are akin to sophisticated queries over a point cloud model.

A typical setup for our system, Alpha, is depicted in Figure 1 (center) and its internal architecture is shown in Figure 1 (bottom). A user can write Damocles expressions with the help of different intuitive query interfaces, including a visual Query-by-Example UI and manual command inputs. These expressions are mapped into Palicus, the hardware that is specially designed to accelerate Damocles operations, allowing it to execute expressions with no perceptible latency. Palicus is fully implemented on an FPGA platform [1] with low power consumption and small footprint. This allows Alpha to be deployed as an edge device.

In the remainder of this paper, we discuss various manipulations that Alpha can perform by demonstrating some application scenarios (Section 2). We also explain the underlying concepts that enable these manipulations to be expressed clearly and executed efficiently (Section 3). Lastly, we conclude by summarizing the advantages of using a system like Alpha for point cloud manipulation and discuss future research directions (Section 4).

2 Alpha Demonstration Scenarios

In this section, we discuss several practical manipulations that are part of typical point cloud applications. We present the scenarios in order of complexity.

2.1 Region of Interest

In our first scenario, we discuss how Alpha supports extracting a region of interest (RoI) from a point cloud. To show how querying can be made user-friendly, we use a Query-by-Example interface, depicted in Figure 2. The form allows a user to choose the type of shape they wish to use as a filter—*e.g.,* a cuboid or a spherical shell. The form also allows the user to select whether the region of interest is within or outside the shape. In Figure 2, for example, the user picked a cuboid expression that selected only the region surrounding the desk we see in Figure 1 (center).

The selection of a RoI over a point cloud has a similar effect to the execution of a `WHERE` clause in a SQL query in that it filters the size of the point cloud to be processed. Like in the `WHERE` clause, the user can combine several shapes (filtering criteria) to define a more sophisticated RoI. This feature is particularly useful in applications where only a subset of the point cloud must be processed to solve a given task. For example, a fixed LiDAR in a traffic intersection could be configured to ignore the buildings that appear in its field of view and focus only on streets and sidewalks.

2.2 Dimensionality Reduction

After selecting a RoI, some tasks within an application may be better achieved using a 2D projection of the scenario instead of the raw 3D point cloud data. These 2D representations, often called *pseudo-images*, could be extracted from the point cloud through some manipulation that Alpha supports. Take for instance a **frontal view** (FV) image, a pseudo-image used in many self-driving scenarios, shown in Figure 3 (bottom-left). When the distance from the LiDAR to the point is also rendered—*e.g.*, as color range—this FV image is called a **depth map**. Alpha generates this representation by projecting the points to the lateral surface of a cylinder centered at the LiDAR sensor, as Figure 3 (top-left) shows, and retains the distance information for each pixel. Formally, the projection from points to pixels is done as follows:

$$x = \frac{\varphi}{res_\varphi} \qquad y = \frac{\theta - \theta_{min}}{res_\theta}$$

where $C := (r, \theta, \varphi, I)$ denotes the point cloud C extracted from the raw network stream, where each point is described by its spherical coordinates—radius r, polar angle θ, and azimuthal angle φ—and the reflection intensity I. Moreover, let θ_{min} denote the lower bound of the LiDAR's vertical field of view and res_θ, res_φ the angular resolutions. We treat C as a matrix with columns r, θ, φ, and I. In practice, we can calculate $C' := (x, y, r)$ by issuing a Damocles expression with RoI filters and arithmetic operations.

A common scenario in self-driving vehicles, as we mentioned above, is applying a RoI filter that focuses on the area in front of a car and producing a depth map of the results. If both operations are performed on hardware, as is the case with Alpha, the system can quickly check for obstacles by applying simple image processing techniques to the resulting pseudo-image.

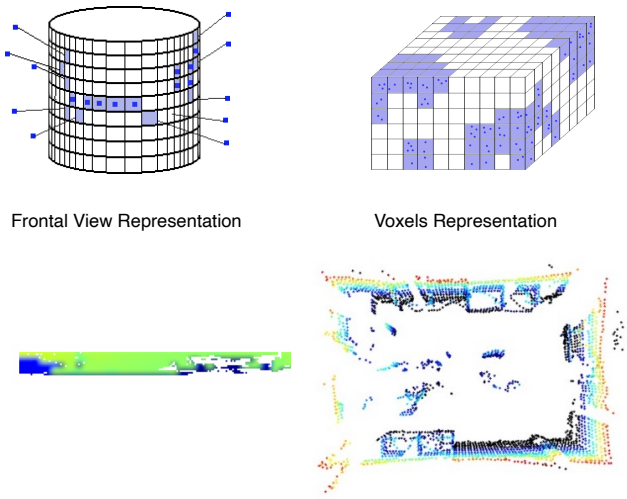

Figure 3: Different representations of the rendition shown on the top of Figure 1: (left) A frontal view projection: the points are projected to the lateral surface of a cylinder centered at the sensor's origin. (right) A projection in which points close to one another are aggregated into voxels (as the 3D version of a pixel is called).

2.3 Downsampling

Besides RoI and arithmetic operations, the Damocles data model also supports summarization over point clouds. Once again, we see similarities with SQL: Summarization resembles the SQL `GROUP BY` clause. We illustrate this feature with a typical forestry application that wishes to determine the density of the canopies in a given region [4]. This application does not require a high-definition rendition of the shapes of tree leaves or grass; it can be achieved by processing a coarser version of the original point cloud. One common approach in this situation is to use a downsampling technique called **voxelization**. Simply put, a voxel is a 3D pixel. Figure 3 (bottom-right) shows a voxelized rendition of the point cloud that appeared in Figure 1 (top) using voxels with a cell width of 10 cm.

In practice, voxelization divides a scene into a regular 3D grid and builds a representation for each cell, as Figure 3 (top-right) shows. In the figure, we represent the non-empty voxels by their centroids, which results in a downsampled point cloud. Like in the dimensionality reduction case, we can use Damocles expressions to capture different voxelization approaches.

3 Damocles and Palicus: A Data Model and its Companion Hardware

The manipulations mentioned above and many others are enabled by Palicus, as we call our LiDAR Processing Unit (LPU). At the heart of the LPU, which Figure 4 (bottom) depicts, rest several specialized processing elements that, combined, can efficiently execute a variety of expressions built with Damocles. We will describe these processing elements next, but we first note that Palicus is programmable. This is made possible by the ability to pipeline any number—up to a limit—of processing elements in an arbitrary order and configure them independently.

At a high level, the Palicus LPU is composed of three types of processing elements:

- **Data transmission and formatting elements.** These elements, which appear in Figure 4 as **Network** and **De-/Parsing** boxes, are responsible for (a) connecting the LPU and the LiDAR via networking [2], (b) transforming the raw point cloud stream into the matrix C we describe in Section 2.2, (c) delivering the resulting logical point cloud into the pipeline for which the LPU is programmed, (d) reformatting the result of this computation for transmission, and (e) transmitting the result.
- **Relational elements.** These elements are closed on matrices, that is, they take a matrix C as input and output a matrix C' that, due to side-effects of a computation, can have different dimensions. The change in dimensions does not prevent the chaining of processing elements—as we will discuss shortly—because the latter are agnostic of dimensions—also up to a certain limit. Figure 4 shows **Arithmetic** and **Filter** (selection) elements. For presentation simplicity, the figure depicts an example expression that uses arithmetic elements only, but the other elements can be part of a computation just as well.
- **Aggregation elements.** These elements implement relational aggregation (GROUP BY) or specific summarization operations tailored specifically to point clouds. In other words, they can aggregate point clouds either using relational methods, such as the voxel representation (Section 2.3), or via more specific reduction methods, such as clustering. The latter require special implementations, which the Palicus LPU carries. Figure 4 refers to this element as **Aggregation**.

Palicus also carries a special component referred in Figure 4 as **Interconnect**. This component is central to the programmability of Palicus because it allows a program to tie the output of an arbitrary processing element to the input of another arbitrary element.

The LPU is programmed as a result of a Damocles expressions compilation, which reduces the latter into smaller calculations and assigns them to individual processing elements. Figure 4 shows how this process unfolds using the expression we described in Section 2.2 to produce the depth map of Figure 3 (bottom-left). The complete expression, which appears on top of Figure 4, receives a point cloud represented as a matrix with 4 numbered columns (radius, polar angle, azimuth, and intensity), and calculates the pixel coordinates x and y. The coordinate x involves a trivial division, as the 6^{th} term of the figure shows. The coordinate y requires a subtraction and a division, represented by the 4^{th} and 5^{th} terms.

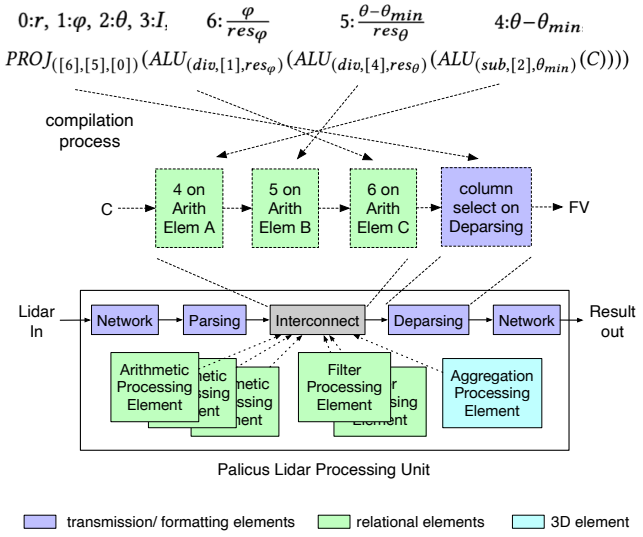

Figure 4: The programmable LPU's high level architecture and usage: (top) A logical expression made of a composition of operator instances. (middle) The mapping of the expression's operators into the available processing elements. Note that the interconnect element can pipeline the processing elements required by the expression by connecting the elements' inputs and outputs, effectively implementing the data flow underlying an expression. (bottom) The hardware can ingest, parse, process, deparse, and emit raw point clouds thanks to data transmission and formatting elements.

4 Conclusion

In this paper, we demonstrated that raw point clouds can be effectively processed using a new set of simple operations—an algebra specifically designed to express a wide range of common manipulations in point clouds. We also showed that the co-designed hardware used to evaluate these expressions can be compact and cost-effective, making it suitable as an edge device. By combining this data model with specialized hardware, our system achieves low latency without sacrificing the support for ad-hoc manipulations.

We hope that the scenarios we presented above will instill in the database community the sense of opportunity that adopting point cloud algebras can create. We aimed to present the point cloud research field in a manner that resonates with this community and left several questions unanswered. This includes potential extensions to the data model, such as defining additional operators and implementing them in hardware. Additionally, data model optimization techniques, such as operation reordering or fusion, could potentially be used to reduce the latency even further or minimize the space required for intermediate computations.

Acknowledgments

This work has received funding from the Swiss State Secretariat for Education (SERI) in the context of the SmartEdge EU project (grant agreement No. 101092908).

References

[1] AMD. [n. d.]. Kria K26 SOM Optimized for vision AI and robotics applications. https://www.amd.com/en/products/system-on-modules/kria/k26.html.
[2] Alex Forencich et al. 2020. Corundum: An Open-Source 100-Gbps Nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. doi:10.1109/FCCM48280.2020.00015
[3] Gary G Gimmestad and David W Roberts. 2023. *Lidar engineering: introduction to basic principles*. Cambridge University Press. doi:10.1017/9781139014106
[4] Qinghua Guo et al. 2023. *LiDAR Principles, Processing and Applications in Forest Ecology*. Elsevier. doi:10.1016/C2020-0-00694-8
[5] Songbin Liu, Min Zhang, Pranav Kadam, and C.-C. Jay Kuo. 2021. *3D point cloud analysis : traditional, deep learning, and explainable machine learning methods*. Springer.
[6] C. Wang et al. 2024. *Introduction to LiDAR Remote Sensing*. CRC Press. doi:10.1201/9781032671512