LIDAR Object Tracking with Hardware Acceleration

McCain Boonma
Northeastern University, Boston, USA
boonma.m@northeastern.edu

***Abstract*—This project presents an efficient solution for real-time processing and analysis of LiDAR data, leveraging FPGA-based hardware acceleration to perform clustering and a Kalman filter for object tracking. Utilizing a YDLIDAR X4 LiDAR sensor, distance data is collected and processed to identify clusters representing objects within the field of view. A PYNQ overlay is employed to accelerate clustering operations on a Xilinx FPGA platform, significantly improving the overall performance. The integration of a Kalman filter enables tracking of detected clusters over time, allowing for the identification and monitoring of objects with constant velocities. Visualization of the processed data is achieved using polar plots with the Matplotlib library, providing a clear representation of the raw LiDAR data and the identified clusters.**

I. Introduction

Light Detection and Ranging (LiDAR) sensors are widely used in various applications, including autonomous vehicles, robotics, and environmental monitoring. These sensors generate large amounts of data that require real-time processing to detect and track objects within their field of view. This paper presents an efficient solution for real-time processing and analysis of LiDAR data, leveraging FPGA-based hardware acceleration to perform clustering and a Kalman filter for object tracking. The proposed approach employs a number of hardware optimizations in the point clustering operation, resulting in significant improvements in overall performance. The processed data is visualized using polar plots , providing a clear representation of the raw LiDAR data and the identified clusters.

II. System Overview

This inclusive system (Fig 1) includes LIDAR data collection, point clustering, and object tracking. The primary focus of this paper is the clustering algortihm implemented on the FPGA programmable logic via an overlay and the object tracking using a Kalman filter implemented on the FPGA processing system.

Fig. 1. System Overview

A. LiDAR Data Collection

The sensor's coordinate system is defined such that 0°originates from the center of the sensor assembly towards the LIDAR sensor's motor, with data collected in the clockwise direction (as shown in Fig. 2). The sensor is connected to the processing system of the PYNQ Z2 via a USB port.

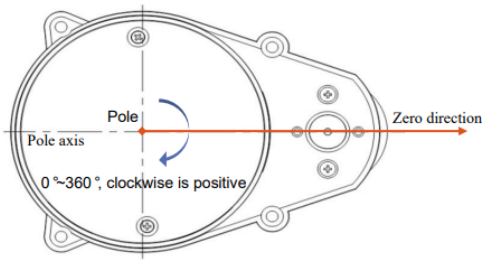

Fig. 2. LIDAR Sensor Coordinate System Definition

To control the LIDAR sensor and collect LIDAR data, the PyLidar3 library is utilized, which is a python 3 package. The LIDAR sensor is capable of a typical angle resolution of 0.50°at 7 Hz, but due to software limitations, the PyLidar3 library provides a resolution of only 1°regardless of the frequency. The collected data is returned in the form of a dictionary, comprising of 360 degrees of angle and corresponding distances in millimeters.

B. Hardware Accelerated Point Clustering

The point clustering operation was implemented on the programmable logic of the FPGA using an Overlay. To achieve this, the point clustering algorithm was developed in Vitis High-Level Synthesis (HLS). The LIDAR data is transferred from the FPGA processing system to the programmable logic through an AXI Stream interface, while the clustered data is transferred via a separate AXI Stream interface. The implemented clustering algorithm is based on the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm, which is commonly utilized in LIDAR point clustering applications.

he clustering algorithm utilized in the proposed LIDAR object tracking system is based on the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. The DBSCAN algorithm takes three parameters: an array of

distance data, the number of data points, and the maximum distance between two points in the same neighborhood (epsilon, ϵ).

The DBSCAN algorithm starts by iterating over all the points in the input data array. For each unvisited point, the algorithm finds all its neighbors within a distance of ϵ and adds them to a cluster. The algorithm then expands the cluster by adding all reachable neighbors of each point in the cluster. The algorithm stops when no more points can be added to the cluster.

The distance calculation used in the DBSCAN algorithm takes into account the angle between the LIDAR points, which are represented as distances in an array. Given two points p_1 and p_2 in the array, the distance calculation between them is as follows:

$$distance = \sqrt{(p_{1,x} - p_{2,x})^2 + (p_{1,y} - p_{2,y})^2} \quad (1)$$

Where $p_{1,x}$ and $p_{2,x}$ are the distances of p_1 and p_2 from the LIDAR sensor along the horizontal axis, and $p_{1,y}$ and $p_{2,y}$ are the distances of p_1 and p_2 from the LIDAR sensor along the vertical axis. The distance calculation also involves converting the angles of the LIDAR points to radians, and using the sine and cosine functions to compute the horizontal and vertical distances respectively.

The implemented DBSCAN algorithm in the code snippet takes an array of distance data, the number of data points, and the maximum distance between two points in the same neighborhood (epsilon, ϵ) as input. The algorithm iterates over all the points in the input array, finds all the neighbors of each unvisited point, and creates a new cluster if the point has enough neighbors. The algorithm then expands the cluster by adding all reachable neighbors, and continues until no more points can be added to the cluster.

C. Constant Velocity Kalman Filter for Object Tracking [WORK IN PROGRESS]

The Kalman filter is a widely used tool for estimating the state of a dynamic system in the presence of noise. In the context of cluster tracking, the Kalman filter can be used to predict the position of each cluster in the next frame based on its position in the current frame.

In our specific case, we can assume that the clusters are stationary and that their minimum size is 5 with an average of 10 points per cluster. However, there is some noise in the system, which can lead to inaccuracies in the cluster positions.

To account for these factors, we can simplify the Kalman filter equations by assuming that the velocity, acceleration, and jerk terms are all zero, so the state vector becomes a two-dimensional vector of the cluster's position coordinates. Secondly, we assume that the state transition matrix is the identity matrix since the clusters are not moving. Lastly, we assume that the measurement noise is proportional to the standard deviation of the cluster size, which is $\sqrt{10}$.

With these assumptions in place, the Kalman filter equations become:

Predict: $\hat{\mathbf{x}}k|k-1 = \mathbf{x}k-1$

$\mathbf{P}k|k-1 = \mathbf{P}k-1|k-1 + \mathbf{Q}_k$, where $\mathbf{Q}_k$ is the process noise covariance matrix at time k, which is assumed to be zero in our case.

Update: $\mathbf{K}k = \mathbf{P}k|k-1\mathbf{H}_k^T(\mathbf{H}k\mathbf{P}k|k-1\mathbf{H}_k^T + \mathbf{R}_k)^{-1}$

$\hat{\mathbf{x}}k|k = \hat{\mathbf{x}}k|k-1 + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}k\hat{\mathbf{x}}k|k-1)$

$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k\mathbf{H}k)\mathbf{P}k|k-1$

Here, $\mathbf{H}_k$ is the observation model at time k, which is the identity matrix in our case, and $\mathbf{R}_k$ is the measurement noise covariance matrix at time k, which is proportional to the standard deviation of the cluster size.

By using the Kalman filter, we can predict the position of each cluster in the next frame based on its position in the current frame, which can improve the accuracy of the cluster tracking process.

D. Data Association - Nearest Neighbor Algorithm - WORK IN PROGRESS

The nearest neighbor algorithm is a simple and effective algorithm for matching clusters across frames. The basic idea is to find the nearest cluster in the next frame for each cluster in the current frame, based on some distance metric.

In the context of cluster tracking, we can define the distance metric as the Euclidean distance between the centroids of the clusters in each frame. Specifically, given two clusters i and j with centroids $\mathbf{c}_i$ and $\mathbf{c}_j$, the distance between them is given by:

$d_{ij} = \sqrt{(\mathbf{c}_i - \mathbf{c}_j)_x^2 + (\mathbf{c}_i - \mathbf{c}_j)_x^2 + (\mathbf{c}_i)_y - (\mathbf{c}_j)_y^2}$

To match clusters between frames, we can use a greedy approach where we start with the closest pair of clusters and then remove them from consideration, and repeat until all clusters are matched. Specifically, we can define a threshold distance d_{th} and match each cluster i in the current frame to the nearest cluster j in the next frame such that $d_{ij} < d_{\text{th}}$. We then remove both clusters i and j from consideration and repeat the process until all clusters have been matched.

However, to improve the robustness of the matching process, we can add some additional criteria to the matching process. For example, we can require that the matched clusters have similar sizes and shapes, or that they have similar orientations. These additional criteria can help to ensure that the matching is accurate and robust, even in the presence of noise and other uncertainties.

Once the clusters have been matched, we can assign a unique cluster ID and color to each cluster. To assign a unique ID, we can maintain a list of all clusters in each frame and assign a unique ID to each new cluster that is detected. We can then use these IDs to track the clusters across frames.

To assign colors to clusters, we can use a predefined color map with a fixed number of colors, such as the 12-color Paired qualitative colormap in matplotlib. We can then assign colors to each cluster based on its ID modulo the number of colors in the colormap. Alternatively, we can use a more sophisticated color mapping algorithm that takes into account the size, shape, and other properties of the clusters.

To display the clusters with their assigned colors, we can use a scatter plot with each cluster represented as a point with

the assigned color. We can also include labels for each cluster to indicate its ID, size, and other properties as needed.

By combining the Kalman filter, nearest neighbor algorithm, and cluster ID and color assignment, we can achieve accurate and robust cluster tracking and visualization for a wide range of applications. By tuning the parameters of the algorithm and incorporating additional features as needed, we can further improve the performance and versatility of the cluster tracking system.

E. Visualization

Visualization plays a crucial role in analyzing the performance of the LiDAR data processing pipeline and understanding the identified clusters and their tracked positions. This project uses the Matplotlib library to generate polar plots for the visualization of raw LiDAR data, identified clusters, and tracked objects.

Two polar plots are created for each iteration of the processing loop:

- Raw Data Plot: This plot displays the distance values collected by the LiDAR sensor at different angles. It helps to visualize the raw LiDAR data and understand the environment.
- Clustered Data Plot: This plot shows the identified clusters and their tracked positions, represented by red crosses. It is useful for evaluating the performance of the clustering algorithm and tracking filter.

The update_plot function is responsible for updating the polar plots with the raw and clustered data. It first clears the axes, then plots the data points on the respective polar plots, and finally updates the plot titles and limits. The function also calls the data_association function to perform data association between the current and previous clusters and update the state of the tracked objects in the constant velocity Kalman filter.

III. Implementation

A. Hardware

The clustering operation is an ideal candidate for hardware acceleration due to its computationally intensive nature and the ability to parallelize the algorithm. By grouping distances into clusters using a straightforward algorithm, hardware acceleration can significantly improve performance. The functionality of the clustering operation was verified by comparing the clusters generated by the hardware implementation to those generated by a software implementation using the same data.

B. Software

The Kalman Filter and data association are potential candidates for hardware acceleration. However, due to the intricacy of their mathematical algorithms and time constraints, hardware testing would have been excessively time-consuming. Thus, software implementation was chosen to expedite the development process.

C. Implementation Overview

The entire processing pipeline is implemented in a Jupyter Notebook, using the PYNQ library for FPGA acceleration, PyLidar3 library for LiDAR data collection, and the Matplotlib library for data visualization. The constant velocity Kalman filter is implemented using the pykalman library.

First, the LiDAR sensor is initialized and connected to the Linux-based system. Then, the PYNQ overlay is loaded onto the FPGA, and the input and output buffers are allocated for the clustering operation. The processing loop iterates for a user-defined number of times, performing the following steps in each iteration:

1) Collect distance data from the LiDAR sensor.
2) Perform the FPGA-based clustering operation on the collected data.
3) Update the constant velocity Kalman filter for the detected clusters.
4) Perform data association between the current and previous clusters.
5) Update the polar plots with the raw and clustered data.

Upon completion of the processing loop, the LiDAR sensor is disconnected, and the program exits.

IV. Results and Discussion

The proposed solution demonstrates the effectiveness of FPGA-based hardware acceleration for real-time processing and analysis of LiDAR data. By offloading the computationally intensive clustering operation onto the FPGA, the overall performance of the system is improved.

The polar plot visualization provides a clear representation of the raw LiDAR data and the identified clusters, facilitating the analysis of the system's performance. The data association algorithm successfully matches the detected clusters with the tracked objects, ensuring that the state of the constant velocity Kalman filter is updated correctly.

The data association portion of the project is currently facing major issues, rendering it non-operational (work is still ongoing to address these issues). Consequently, the Kalman filter is only able to return the current mean position of a cluster, and it is not yet behaving as a fully functional Kalman Filter. Despite these issues, communication between the LIDAR sensor and the PYNQ board is reliable, and connectivity issues are rare. While the clustering operation has not been benchmarked against the initial software implementation, it is operating reliably. However, plotting the data remains the slowest part of the pipeline.

Despite these challenges, the project is capable of connecting to a LIDAR sensor and successfully clustering LiDAR data. Additionally, groundwork has been laid for a functioning Kalman Filter, which will enable accurate tracking of objects with constant velocities.

V. Conclusion

This paper presents an efficient solution for real-time processing and analysis of LiDAR data using FPGA-based hardware acceleration. The proposed approach employs a

YDLIDAR X4 LiDAR sensor for data collection and a Xilinx FPGA platform for acceleration, achieving significant improvements in overall performance. The integration of a constant velocity Kalman filter enables accurate tracking of detected clusters over time, while the data visualization using polar plots provides a clear representation of the raw LiDAR data and the identified clusters. Future work may explore the use of more advanced clustering algorithms and tracking filters to further enhance the performance of the system, as well as the implementation of additional features such as object classification and prediction.

REFERENCES

[1] L, Mallidi, "PyLidar3," GitHub, 2020, url: https://github.com/lakshmanmallidi/PyLidar3

[2] K, Konstantinidis, "Detection and Tracking of Moving Objects with 2D LIDAR," GitHub, 2022, url: https://github.com/kostaskonkk/datmo

[3] K. Beomseong, C. Baehoon, Y. Minkyun, K. Hyunju, K. Euntai. (2014). Robust Object Segmentation Using a Multi-Layer Laser Scanner. Sensors (Basel, Switzerland). 14. 20400-18. 10.3390/s141120400.

[4] R. Su, J. Tang, J. Yuan and Y. Bi, "Nearest Neighbor Data Association Algorithm Based on Robust Kalman Filtering," 2021 2nd International Symposium on Computer Engineering and Intelligent Communications (ISCEIC), Nanjing, China, 2021, pp. 177-181, doi: 10.1109/ISCEIC53685.2021.00044.

[5] N. Baisa, "Derivation of a Constant Velocity Motion Model for Visual Tracking," arXiv, 2020, doi: 10.48550/ARXIV.2005.00844