# An Efficient Accelerator for Deep Learning-based Point Cloud Registration on FPGAs

Keisuke Sugiura and Hiroki Matsutani
Dept. of ICS, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan 223-8522
Email: {sugiura,matutani}@arc.ics.keio.ac.jp

***Abstract*—Point cloud registration is the basis for many robotic applications such as odometry and Simultaneous Localization And Mapping (SLAM), which are increasingly important for autonomous mobile robots. The limitation of computational resources and power budgets on such robots motivates us to study the resource-efficient registration method on low-cost edge devices. In this paper, we propose an FPGA-based novel pipeline for 3D point cloud registration built upon a recent deep learning-based method, PointNetLK. Based on the profiling results, we focus on the PointNet feature extraction as it becomes a major bottleneck; we improve its scalability and memory-efficiency by consuming each input point one-by-one in a pipelined manner instead of processing the whole point cloud at once. We then design a fully-parallelized and pipelined accelerator consisting of a custom PointNet IP core, which fits within both low-cost and mid-range FPGAs (e.g., Avnet Ultra96v2 and Xilinx ZCU104). Experimental results show that our proposed pipeline achieves up to 21.34x and 69.60x faster registration speed than the vanilla PointNetLK and ICP, respectively, while only consuming 722mW and maintaining the same level of accuracy.**

***Index Terms*—Point Cloud Registration, PointNet, FPGA**

## I. Introduction

Point cloud registration is the key component for 3D reconstruction and robotic applications, e.g., LiDAR (Light Detection and Ranging) SLAM. It is the process of estimating a 3D rigid-body transform that best aligns a pair of point clouds. In LiDAR SLAM, a current robot pose is updated by matching two consecutive point clouds acquired by a 3D LiDAR sensor, and computing a relative rigid motion of the robot. Apparently, the choice of registration method directly affects the performance and accuracy of SLAM. Considering that such robotic applications have soft real-time constraints and are deployed on mobile robots (e.g., micro UAVs) with limited resources and power, it is of a crucial importance to develop a lightweight point cloud registration pipeline targeting low-power mobile devices.

A myriad of registration methods have been proposed over the past 30 years for improved accuracy and robustness. ICP (Iterative Closest Point) [1] is the most widely-known method; it alternates between finding point correspondences and computing a rigid transform that minimizes Euclidean distances between matched point pairs. Since the former involves a nearest-neighbor search, ICP has a computational complexity of at least $O(N \log N)$ (or $O(N^2)$), where $N$ is the number of points. Another weakness is that ICP is sensitive to initial estimates and can converge to spurious local minima unless the initial estimate is sufficiently close to the global optimum.

Inspired by the tremendous success of deep learning, the integration of deep learning into registration tasks has emerged as a new research topic. Similar to ICP, some methods firstly predict point correspondences by assigning weights (scores) to all the point pairs using DNNs, and then use SVD (Singular Value Decomposition) to find a rigid transform in closed form; such correspondence search incurs a high computational cost of $O(N^2)$. Other methods employ DNNs to directly estimate transformation parameters from input point clouds; they are still computationally intensive due to large number of parameters and complex network architectures, requiring GPUs and desktop-grade PCs for real-time performance.

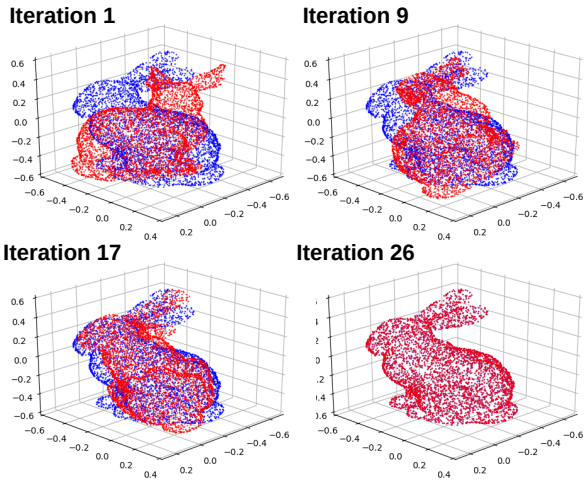


Fig. 1. Examples of outputs from our proposed FPGA-accelerated registration pipeline (Stanford bunny). PointNetLK, a backbone DNN for our registration pipeline, was trained on the 20 categories (airplane to lamp) in ModelNet40 dataset. The pipeline successfully aligns a pair of bunny models which is distinct from the training set, showing a generalization ability. A template (blue) was rotated $90^\circ$ around a random axis to generate a source (red).

PointNetLK [2] is a recently proposed method based on an entirely new approach, which combines the Lucas-Kanade (LK)-based iterative optimization and PointNet feature embedding. As PointNetLK does not rely on point correspondences and uses a simple DNN (i.e., PointNet) for feature extraction, it has $O(N)$ computational cost and requires $O(1)$ memory space. This brings a better scalability and a significant performance advantage compared to the other learning-based methods, making it well-suited for real-time applications on edge devices.

In this paper, we propose a fast and highly-efficient 3D point cloud registration pipeline based on PointNetLK for edge devices. We focus on a PointNet feature extraction as it presents a major performance bottleneck, and improve its scalability and memory-efficiency by optimizing the scheduling of computations inside PointNet. Then, we design an FPGA-based dedicated accelerator incorporating a fully-pipelined PointNet

IP core, and implement it on both low-cost and mid-range FPGA SoCs. Experimental results demonstrate that our proposed pipeline achieves significantly faster registration compared to both vanilla PointNetLK and ICP, without degrading the generalization ability and accuracy.

The rest of the paper is organized as follows: Section II overviews related works. Section III formulates the registration problem and describes PointNetLK algorithm. Section IV illustrates the design of our FPGA-based registration pipeline. The evaluation in terms of speed, accuracy, resource utilization, and power consumption is presented in Section VI. Section VII concludes the paper.

## II. Related Works

### A. Deep Learning-based Point Cloud Registration

In the past few years, deep learning techniques have been successfully applied to point cloud registration [2]–[5], outperforming traditional geometry-based methods such as Iterative Closest Point (ICP) [1], Normal Distributions Transform (NDT) [6], and ICP variants [7], [8].

*1) End-to-end approach:* The common learning-based approach is to solve registration tasks in an end-to-end fashion [9]–[13]. For instance, PCRNet [12] directly regresses a 3D rigid transform from a pair of point sets using PointNet in a Siamese architecture. 3DRegNet [13] uses a stack of ResNet blocks to compute a feature for each given point correspondence, which is passed on to the CNN-based model for pose estimation. The authors of [11] propose an unsupervised method for jointly aligning a sequence of 2D point clouds and building an occupancy grid map. Such an end-to-end strategy comes with a major drawback; it usually requires a larger dataset and a sophisticated network with millions of parameters to accurately estimate the 3D transformation parameters directly from noisy point clouds, hence making the training difficult and time-consuming. In addition, they are not suitable for deployment on the edge devices due to their high resource requirements.

*2) Correspondence-based approach:* Another approach combines a deep feature extraction and a conventional closed-form pose estimation; a number of methods [4], [14]–[20] predict point correspondences and then perform SVD to compute a rigid transform. DCP [14] combines a graph convolution and an attention mechanism to extract point-wise features which take into account both intra and inter point cloud information, and computes a set of correspondences by matching these features. DeepVCP [15] builds a DNN model based on PointNet and CNNs for per-point feature extraction and matching. PRNet [16] deals with missing corresponding points and achieve partial-to-partial registration. These correspondence-based methods suffer from the lack of scalability; they consider all the possible point pairs (i.e., soft correspondences) to make DNNs fully-differentiable and trainable with a standard backpropagation, leading to $O(N^2)$ computational complexity and memory space. Besides, they are generally prone to missing or incorrect correspondences, which frequently occur due to shape ambiguity, noises, outliers, and occlusions in 3D point clouds.

*3) Lucas-Kanade (direct feature alignment) approach:* Aside from the above approaches, some methods perform the iterative registration in the framework of LK algorithm. PointNetLK [2] first extracts global features from the input point clouds using PointNet, and then iteratively updates a 3D rigid transform by directly aligning these features. To find an update to the transform, it computes a Jacobian of the PointNet feature with respect to transformation parameters using a finite-difference approximation. Sekikawa *et al.* [21] replaces MLPs in PointNet with look-up tables to eliminate the vector-matrix operations, and accelerates PointNet feature embedding and Jacobian computation in PointNetLK. Li *et al.* [5] proposes to analytically compute a Jacobian by decomposing it into two terms (feature gradient and warp Jacobian) instead of approximating it to avoid numerical instabilities and improve generalization capability. Importantly, PointNetLK and its variants do not depend on point correspondences; they directly minimize the difference between global features of point clouds, leading to lower computational complexity and memory space ($O(N)$ and $O(1)$ as described in Sec. III). We opt to use PointNetLK as a backbone for a fast, highly-efficient point cloud registration pipeline in this paper.

### B. FPGA-based Acceleration of Point Cloud Registration

Only a few works studied the FPGA-based acceleration of 3D point cloud registration. Kosuge *et al.* [22] develop an accelerator for the ICP-based object pose estimation, which is a critical process in picking robots. They focus on the $k$-nearest neighbor ($k$-NN) search, which constitutes a major bottleneck in ICP due to its $O(N \log N)$ (or $O(N^2)$) complexity, and devise a novel hierarchical graph data structure for improved search efficiency. The proposed accelerator combines a parallelized distance computation unit and a dedicated sorter unit to speed up the graph construction and NN search. Deng *et al.* [23] present an FPGA-based accelerator for NDT. NDT [6] models point clouds as a set of voxels, each of which represents the Gaussian distribution of points. They introduce a new hierarchical data structure to accelerate the voxel search operations. Eisoldt *et al.* [24] implement Truncated Signed Distance Function (TSDF)-based registration method and TSDF map update process onto the FPGA logic circuit for efficient 3D LiDAR SLAM. These works successfully demonstrate the effectiveness of FPGA acceleration for well-established or geometry-based registration methods. This paper is the first to explore the FPGA-based registration pipeline built upon a deep learning method.

## III. Background

### A. PointNetLK Algorithm

In this section, we briefly describe the PointNetLK algorithm. We summarize the algorithm in Alg. 1 (refer to [2], [5] for more details).

As shown in Fig. 1, the aim of point registration is to align two 3D point clouds, referred to as a **template** $\mathcal{P}_T$ and **source** $\mathcal{P}_S$, by estimating a 3D rigid transform $\mathbf{G} \in \mathrm{SE}(3)$ from $\mathcal{P}_S$ to $\mathcal{P}_T$. One classical and straightforward approach is to minimize the Euclidean distances between corresponding point pairs, which however requires a costly nearest neighbor search with at least $O(N \log N)$ complexity. To avoid this, PointNetLK takes

another approach: it finds an optimal transform $\mathbf{G}$ such that global features extracted from two point clouds are close to each other, i.e., $\phi(\mathbf{G} \cdot \mathcal{P}_S) = \phi(\mathcal{P}_T)$.

$\phi : \mathbb{R}^{N\times 3} \to \mathbb{R}^K$ denotes a PointNet that maps a point cloud of $N$ points into a $K$-D global feature[1]. The transform $\mathbf{G}(\boldsymbol{\xi}) = \exp(\boldsymbol{\xi}^\wedge)$ is computed from a 6D twist $\boldsymbol{\xi} \in \mathbb{R}^6$ via the exponential map. The definition of wedge operator ($^\wedge$) is found in [25]. From the above notations, we can formulate the registration problem as a minimization of the squared difference between two PointNet features $\phi(\mathbf{G}(\boldsymbol{\xi}) \cdot \mathcal{P}_S)$, $\phi(\mathcal{P}_T)$ with respect to the 6D twist parameters $\boldsymbol{\xi}$:

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}} \left|\phi(\mathbf{G}(\boldsymbol{\xi}) \cdot \mathcal{P}_S) - \phi(\mathcal{P}_T)\right|^2. \tag{1}$$

For efficiency, PointNetLK uses an inverse-compositional (IC) formulation and swaps the roles of template and source. Instead of following Eq. 1, it computes a twist $\boldsymbol{\xi}$ such that the rigid transform $\mathbf{G}(\boldsymbol{\xi})^{-1} = \exp(-\boldsymbol{\xi}^\wedge)$ from $\mathcal{P}_T$ to $\mathcal{P}_S$ minimizes the difference between $\phi(\mathcal{P}_S)$ and $\phi(\mathbf{G}(\boldsymbol{\xi})^{-1} \cdot \mathcal{P}_T)$:

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}} \left|\phi(\mathcal{P}_S) - \phi(\mathbf{G}(\boldsymbol{\xi})^{-1} \cdot \mathcal{P}_T)\right|^2. \tag{2}$$

By applying the first-order Taylor expansion, we linearize the term $\phi(\mathbf{G}(\boldsymbol{\xi})^{-1} \cdot \mathcal{P}_T)$:

$$\phi(\mathbf{G}(\boldsymbol{\xi})^{-1} \cdot \mathcal{P}_T) \simeq \phi(\mathcal{P}_T) + \mathbf{J}\boldsymbol{\xi}, \tag{3}$$

where $\mathbf{J} \in \mathbb{R}^{K\times 6}$ is a Jacobian of the PointNet feature $\phi(\mathcal{P}_T)$ with respect to the twist parameters $\boldsymbol{\xi}$:

$$\mathbf{J} = \frac{\partial}{\partial \boldsymbol{\xi}} \phi(\mathbf{G}(\boldsymbol{\xi})^{-1} \cdot \mathcal{P}_T). \tag{4}$$

Each column vector $\mathbf{J}_i \in \mathbb{R}^K$ $(i = 1, \ldots, 6)$ of $\mathbf{J}$ is computed by numerical gradient approximation as follows (line 4):

$$\mathbf{J}_i \simeq \frac{\phi(\exp(-t_i \boldsymbol{e}_i) \cdot \mathcal{P}_T) - \phi(\mathcal{P}_T)}{t_i}, \tag{5}$$

where $t_i$ is an infinitesimal perturbation to the twist (e.g., 0.01) and $\boldsymbol{e}_i \in \mathbb{R}^6$ is a unit vector whose $i$-th element is 1 and the others are 0. By substituting Eq. 3 into Eq. 2, we can solve for the optimal twist $\boldsymbol{\xi}$ as follows (line 8):

$$\boldsymbol{\xi} = \mathbf{J}^\dagger \left(\phi(\mathcal{P}_S) - \phi(\mathcal{P}_T)\right), \tag{6}$$

where $\mathbf{J}^\dagger = (\mathbf{J}^\top \mathbf{J})^{-1} \mathbf{J}^\top \in \mathbb{R}^{6\times 6}$ is a pseudo-inverse of $\mathbf{J}$ (line 5). We transform the source $\mathcal{P}_S$ using $\Delta\mathbf{G} = \exp(\boldsymbol{\xi}^\wedge)$ (lines 9-10) and proceed to the next iteration until convergence. The final solution $\mathbf{G}$ is obtained as a product of all incremental transforms, i.e., $\mathbf{G} = \Delta\mathbf{G}_n \cdots \Delta\mathbf{G}_2 \Delta\mathbf{G}_1$, where $\Delta\mathbf{G}_k$ is the estimate at the $k$-th iteration, and $n$ is the number of iterations.

As seen in Eq. 5, the computation of $\mathbf{J}$ is expensive, as we need to perturb a point cloud and compute PointNet features six times in total. In the original formulation (Eq. 1), $\mathbf{J}$ is computed by perturbing $\mathcal{P}_S$ instead of $\mathcal{P}_T$, meaning that $\mathbf{J}$ should be recomputed after transforming $\mathcal{P}_S$ (line 10) at every iteration. With the IC formulation (Eq. 2), $\mathbf{J}$ is computed only once at the initialization phase (lines 3-5) and remains constant during the optimization, hence greatly improving the computational efficiency of PointNetLK.

[1]For ease of explanation, we assume that both point clouds contain the same number of points $N$, i.e., $\mathcal{P}_S, \mathcal{P}_T \in \mathbb{R}^{N\times 3}$.

**Algorithm 1** Point cloud registration using PointNetLK (The colored part is accelerated using FPGA.)

**Require:** $\mathcal{P}_S, \mathcal{P}_T \in \mathbb{R}^{N\times 3}$, initial estimate $\mathbf{G}_0 = \mathbf{I}$ (optional), PointNet encoder $\phi : \mathbb{R}^{N\times 3} \to \mathbb{R}^K$

**Ensure:** Rigid transform $\mathbf{G} \in \mathrm{SE}(3)$ from $\mathcal{P}_S$ to $\mathcal{P}_T$

▷ **Initialization step**

1: Apply an initial transform: $\mathcal{P}_S \leftarrow \mathbf{G}_0 \cdot \mathcal{P}_S$
2: Compute a PointNet encoding of template: $\phi(\mathcal{P}_T) \in \mathbb{R}^K$
3: Perturb a template: $\phi(\exp(-t_i \boldsymbol{e}_i) \cdot \mathcal{P}_T),\ i = 1, \ldots, 6$
4: Compute a Jacobian: $\mathbf{J} \in \mathbb{R}^{K\times 6}$ (Eq. 5)
5: Compute a pseudo-inverse: $\mathbf{J}^\dagger \leftarrow (\mathbf{J}^\top \mathbf{J})^{-1} \mathbf{J}^\top$

▷ **Optimization step**

6: **for** $i = 1, 2, \ldots, i_{\max}$ **do**
7: Compute a PointNet encoding of source: $\phi(\mathcal{P}_S)$
8: Compute an optimal twist: $\boldsymbol{\xi}_i \leftarrow \mathbf{J}^\dagger \left(\phi(\mathcal{P}_S) - \phi(\mathcal{P}_T)\right)$
9: Compute an update: $\Delta\mathbf{G}_i \leftarrow \exp(\boldsymbol{\xi}_i^\wedge)$
10: Transform the source: $\mathcal{P}_S \leftarrow \Delta\mathbf{G}_i \cdot \mathcal{P}_S$
11: **if** $|\boldsymbol{\xi}_i| < \varepsilon$ **then** ▷ Check convergence
12: **break**
13: **return** $\mathbf{G} = \Delta\mathbf{G}_i \cdots \Delta\mathbf{G}_2 \Delta\mathbf{G}_1 \mathbf{G}_0$

PointNet [26] is a simple yet powerful network for point cloud processing. The network consists of five fully-connected layers (Fig. 3), each of which is followed by batch normalization and ReLU activation, to extract $K$-D point-wise local features from input points. The max-pooling layer is placed at the end of network to aggregate point-wise features and compute a global feature. One notable feature is that the computation for each point is independent except the last max-pooling layer, i.e., PointNet offers a massive data parallelism and are suitable for FPGA acceleration. PointNet does not entail convolutions, attention mechanisms or skip connections, leading to the ease of implementation, and the number of network parameters is fairly low (i.e., 0.15M) compared to the end-to-end methods (Sec. II-A1). As PointNet directly takes in raw 3D point coordinates, the preprocessing such as normal estimation are not required. Despite the sparsity and irregularity of point clouds, PointNet does not involve random accesses to input data and is suitable for pipelining (Fig. 4). The computation and memory cost of PointNet is $O(N)$ and $O(1)$ (so does PointNetLK), leading to a significant advantage compared to the correspondence-based methods (Sec. II-A2).

## IV. Method

### A. Overview of the Registration Pipeline

In this section, we first present a design of our registration pipeline based on PointNetLK for FPGA SoCs. Our design consists of a fully-pipelined and parallelized PointNet IP core, since PointNet feature extraction (lines 2-3, 7 in Alg. 1) constitutes a large portion of the execution time as shown in Fig. 9. We then describe design optimizations to exploit the intra- and inter-layer parallelism in PointNet, and improve scalability and memory-efficiency of the feature extraction.

Fig. 2 depicts a block diagram of our board-level design for Xilinx Zynq UltraScale+ MPSoC family, which consists of the processing system (PS) part and programmable logic

(PL) part. The proposed PointNet IP core and a Direct Memory Access (DMA) controller are placed inside the PL part, which compute a global feature $\boldsymbol{\phi}(\mathcal{P})$ from an input point cloud $\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N\} \in \mathbb{R}^{N\times 3}$ upon a request from the PS part (lines 2-3, 7 in Alg. 1 are offloaded to the PL part). The PS part is in charge of setting up the IP core and triggering a DMA controller. Other steps such as Jacobian computation (e.g., lines 4-5 and 9-10 in Alg. 1) are also performed on the PS part. For the high-speed data transfer, the DMA controller is connected to a 32-bit wide high-performance slave port (HPC port) and utilizes AXI4-Stream protocol (red lines in Fig. 2). The control registers are accessible through the AXI4-Lite interface connected to a 32-bit wide high-performance master port (HPM port, blue lines in Fig. 2).

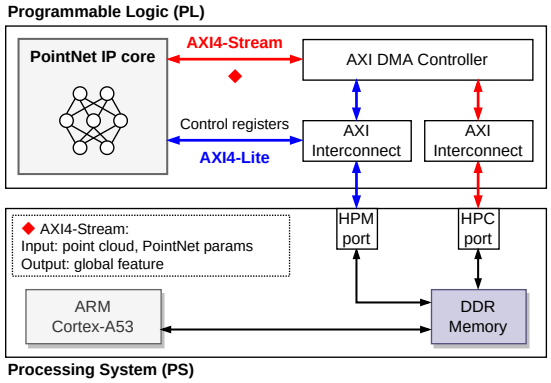


Fig. 2. Board-level design (Xilinx Zynq UltraScale+ MPSoC)

Our PointNet IP core has two operation modes: **weight initialization** and **feature extraction**. In the **weight initialization** mode, the IP core receives PointNet model parameters (e.g., weight and bias) through the AXI4-Stream interface and stores them to the on-chip BRAM buffer. The IP core returns a nonzero 32-bit value as an acknowledgement message to notify that the initialization is complete and the core is in ready state. In the **feature extraction** mode, a 1024D global feature $\boldsymbol{\phi}(\mathcal{P}) \in \mathbb{R}^{1024}$ is firstly initialized with zeros. Then, as illustrated in Fig. 3, the IP core receives a 3D coordinate $\boldsymbol{p}_i \in \mathbb{R}^3$, $i = 1, \ldots, N$ for each point and computes a 1024D point-wise local feature $\boldsymbol{\psi}(\boldsymbol{p}_i) \in \mathbb{R}^{1024}$ by propagating through five consecutive MLP layers. The global feature $\boldsymbol{\phi}(\mathcal{P})$ is updated by taking the element-wise maximum of $\boldsymbol{\phi}(\mathcal{P})$ and $\boldsymbol{\psi}(\boldsymbol{p}_i)$ (Eq. 8). In this way, point-wise local features $\{\boldsymbol{\psi}(\boldsymbol{p}_1), \ldots, \boldsymbol{\psi}(\boldsymbol{p}_N)\}$ are aggregated into one global feature $\boldsymbol{\phi}(\mathcal{P})$. After the computation is done for all points, the current $\boldsymbol{\phi}(\mathcal{P})$ is returned to PS as a final result.

The most straightforward way for extracting a global feature $\boldsymbol{\phi}(\mathcal{P})$ is that, we first transfer the whole point cloud $\mathcal{P}$ from PS to PL, compute local features $\boldsymbol{V} = \{\boldsymbol{\phi}(\boldsymbol{p}_1), \ldots, \boldsymbol{\phi}(\boldsymbol{p}_N)\} \in \mathbb{R}^{N\times K}$ for all points at once, and aggregate them using max-pooling at once ($\boldsymbol{\phi}(\mathcal{P}) = \mathrm{MaxPool}(\boldsymbol{V})$). Despite its simplicity, this three-step approach has a clear drawback regarding the FPGA implementation. It would consume a large part of the scarce on-chip memory, as it requires $O(N)$ memory space to store intermediate layer outputs and local features $\boldsymbol{V}$ for all points. As a result, the number of input points is limited by the amount of available on-chip memory resources, which degrades the scalability and memory-efficiency.

As described above, our PointNet IP core consumes each point $\boldsymbol{p}_i$ one-by-one sequentially, and thus requires a buffer for storing layer outputs and a local feature $\boldsymbol{\psi}(\boldsymbol{p})$ for a single point $\boldsymbol{p}$. This substantially improves the memory-efficiency, as the memory consumption is reduced from $O(N)$ to just $O(1)$. The design is also flexible and scalable in a sense that it accepts point clouds of any size (i.e., does not limit the number of input points). Compared to the three-step approach, our design also hides the data transfer overhead (Sec. IV-D).

### B. Modules in the PointNet IP core

As shown in Fig. 3, the IP core is composed of three types of layer modules: **FC**, **BN-ReLU**, and **MaxPool**. $\mathbf{FC}(K, L)$ corresponds to a standard affine layer; it computes a $L$-D output $\boldsymbol{y} = \mathbf{W}\boldsymbol{x} + \boldsymbol{b}$ from a $K$-D input $\boldsymbol{x} \in \mathbb{R}^K$, where $\mathbf{W} \in \mathbb{R}^{L\times K}$ is a weight and $\boldsymbol{b} \in \mathbb{R}^L$ is a bias term. $\mathbf{BN\text{-}ReLU}(K)$ combines a batch normalization and a ReLU activation: given an input $\boldsymbol{x} = [x_1, \ldots, x_N]^\top \in \mathbb{R}^K$, its output $\boldsymbol{y} = [y_1, \ldots, y_N]^\top \in \mathbb{R}^K$ is obtained as follows:

$$y_i = \max\left(0, \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} \cdot w_i + b_i\right) \quad (1 \le i \le K), \tag{7}$$

where $\mu_i, \sigma_i$ are the mean and standard deviation estimated from the training data, $w_i, b_i$ denote the learnable weight and bias, and $\varepsilon$ is a small positive number (e.g., $10^{-5}$) to avoid division by zero, respectively. In the actual implementation, we precompute reciprocals of $\sqrt{\sigma_i^2 + \varepsilon}$ on the CPU, to replace division with multiplication and eliminate square root operations. $\mathbf{MaxPool}(K)$ updates a global feature $\boldsymbol{\phi}(\mathcal{P}) = [\phi_1, \ldots, \phi_K]^\top \in \mathbb{R}^K$ using a point-wise local feature $\boldsymbol{\psi}(\boldsymbol{p}) = [\psi_1, \ldots, \psi_K]^\top \in \mathbb{R}^K$ as follows:

$$\phi_i \leftarrow \max(\phi_i, \psi_i) \quad (1 \le i \le K). \tag{8}$$

As discussed in the following sections, we exploit both intra- and inter-layer parallelism to minimize the latency.

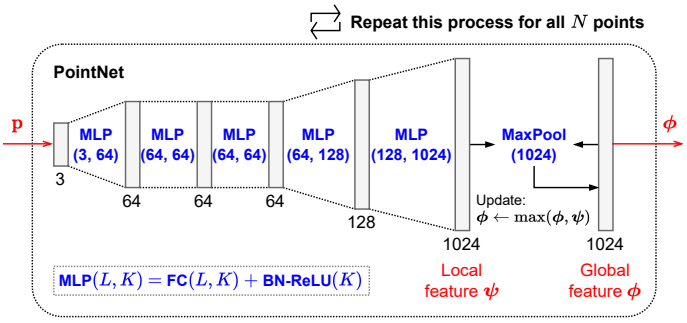


Fig. 3. Computation flow inside the PointNet IP core

### C. Exploiting the Intra-layer Parallelism

$\mathbf{FC}(K, L)$ involves a matrix-vector multiplication $\boldsymbol{y} = \mathbf{W}\boldsymbol{x} + \boldsymbol{b}$ between a weight $\mathbf{W} \in \mathbb{R}^{L\times K}$ and an input $\boldsymbol{x} \in \mathbb{R}^K$, represented by two nested loops over $K$ and $L$. We unroll the inner loop over $K$ by setting an unrolling factor to $B \ge 1$ to parallelize the multiplication between weights $w_{i,j}, \ldots, w_{i,j+B-1}$ and inputs $x_j, \ldots, x_{j+B-1}$. The

intermediate values $w_{i,k}x_k$ ($j \leq k \leq j+B-1$) are then accumulated using an adder tree, which takes $\log B$ iterations. In this way, the number of iterations is reduced from $KL$ to $L(K/B+\log B)$, which amounts to roughly $B$x speedup. This approach requires $B$ DSP blocks and the array partitioning of $\mathbf{W}, \boldsymbol{x}$ to increase the number of read operations per clock cycle. We further reduce the latency by pipelining the inner loop. For instance, **FC**$(64, 64)$ modules for the second and third fully-connected layers (see Fig. 3) are pipelined and parallelized by setting $B = 16$, which results in the 16.3x speedup (i.e., latency is reduced from 8.39$\mu$s to 513ns). **BN-ReLU**$(K)$ and **MaxPool**$(K)$ are easily parallelizable, as the computation for each output element ($y_i$ or $\phi_i$) is independent as seen in Eqs. 7-8. We set the unrolling factor $B \geq 1$ to compute multiple output elements ($y_i, \ldots, y_{i+B}$ or $\phi_i, \ldots, \phi_{i+B}$) and obtain $B$x performance improvement.

### D. Exploiting the Inter-layer Parallelism

We also exploit the coarse-grained task-level parallelism to further improve the performance. As depicted in Fig. 4, the layer modules work in a pipelined manner: this allows to overlap the computations for multiple input points and hide the data transfer overhead. For instance, the fifth MLP layer (MLP5) computes a 1024D local feature of the first point $\boldsymbol{p}_1$, while the fourth MLP layer (MLP4) computes a 128D local feature of the second point $\boldsymbol{p}_2$. We carefully choose a loop unrolling factor $B$ for each module to make the latency of all modules as even as possible and maximize the effectiveness of pipelining. Table I lists the unrolling factors and latencies ($B$ and $T$) for modules inside the core. As expected, **FC**$(128, 1024)$ module for the last fully-connected layer is a bottleneck of the pipeline: we use the maximum possible value $B = 128$ to fully unroll the loop. For the other modules, we adjust the unrolling factor $B$ such that their latencies do not exceed the one of **FC**$(128, 1024)$.

TABLE I
UNROLLING FACTORS AND LATENCIES FOR LAYER MODULES

| Module | $B$ | $T$ ($\mu$s) | Module | $B$ | $T$ ($\mu$s) |
|---|---|---|---|---|---|
| **FC**$(3, 64)$ | 1 | 5.77 | **BN-ReLU**$(64)$ | 1 | 0.68 |
| **FC**$(64, 64)$ | 16 | 5.13 | **BN-ReLU**$(128)$ | 1 | 1.32 |
| **FC**$(64, 128)$ | 32 | 7.69 | **BN-ReLU**$(1024)$ | 2 | 5.16 |
| **FC**$(128, 1024)$ | 128 | **10.28** | **MaxPool**$(1024)$ | 2 | 5.14 |

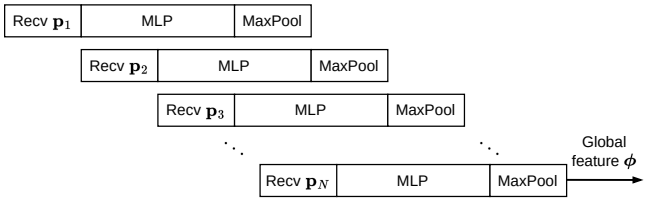


Fig. 4. Pipelined execution inside the PointNet IP core

## V. IMPLEMENTATION DETAILS

We developed a custom PointNet IP core using Xilinx Vitis HLS 2020.2, and used Xilinx Vivado 2020.2 for synthesis and place-and-route. We chose Xilinx Zynq UltraScale+ MPSoC devices, namely, Xilinx ZCU104 Evaluation Kit (XCZU7EV-2FFVC1156) and Avnet Ultra96v2 (ZU3EG A484) as target FPGA SoCs (Fig. 5), which integrate an FPGA and a mobile CPU on the same board. The specifications of these FPGAs are listed in Table II. They both run Ubuntu 20.04-based Pynq Linux 2.7 on a quad-core ARM Cortex-A53 CPU running at 1.2GHz and have a 2GB of DRAM. We set the operation frequency of our accelerator to 100MHz, which is a default setting in the Vivado toolchain. To prevent the accuracy loss, our design uses a 32-bit fixed-point format with 16-bit integer part and 16-bit fraction part for both layer outputs and PointNet parameters.

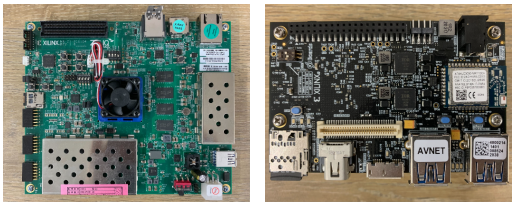


Fig. 5. FPGA boards (left: Xilinx ZCU104, right: Avnet Ultra96v2)

TABLE II
FPGA SPECIFICATIONS OF XILINX ZCU104 AND AVNET ULTRA96V2

| Board | BRAM | DSP | FF | LUT |
|---|---|---|---|---|
| ZCU104 | 312 | 1728 | 460800 | 230400 |
| Ultra96v2 | 216 | 360 | 141120 | 70560 |

We took the PointNetLK source code used in the original paper [2], and modified it to offload PointNet feature extraction (lines 2-3, 7 in Alg. 1) to our FPGA accelerator. The code is implemented using Python 3.8.2 with PyTorch 1.10.2. For both ZCU104 and Ultra96v2, PyTorch was compiled from source using GCC 9.3.0 with ARM Neon intrinsics enabled to fully take advantage of the quad-core CPU. We used the same setting of hyperparameters as in the original code. The number of training epochs is set to 250, and an infinitesimal perturbation $t_i$ to 0.01 (Eq. 5). As an optimizer, Adam with a learning rate of 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$ is adopted, and the batch size is set to 32. The convergence criterion is $|\boldsymbol{\xi}| < 10^{-7}$ (line 11 in Alg. 1), where $\boldsymbol{\xi}$ is an update to the rigid transform as described in Section III-A. PointNetLK model is trained on a workstation equipped with Intel Core i9-9900X at 3.5GHz, 64GB DRAM, and GeForce GTX 2080 Ti.

Given an estimated and a ground-truth rigid transform $\mathbf{G}, \mathbf{G}^* \in \mathrm{SE}(3)$ from source $\mathcal{P}_S$ to template $\mathcal{P}_T$, and a pair of PointNet features $\boldsymbol{\phi}(\mathbf{G} \cdot \mathcal{P}_S), \boldsymbol{\phi}(\mathcal{P}_T)$, the training loss $\mathcal{L}(\mathbf{G})$ is computed as follows:

$$\mathcal{L}(\mathbf{G}) = \left|\mathbf{G}\mathbf{G}^{*-1} - \mathbf{I}\right|_2^2 + \lambda \left|\boldsymbol{\phi}(\mathbf{G} \cdot \mathcal{P}_S) - \boldsymbol{\phi}(\mathcal{P}_T)\right|_2^2, \quad (9)$$

where the first term represents the registration error, and the second one is a regularization term, which encourages the PointNet model to produce similar features for the similar (i.e., aligned) point clouds $\mathbf{G} \cdot \mathcal{P}_S, \mathcal{P}_T$. $\lambda$ is a weighting factor (set to 1 in this paper). $|\mathbf{A}|_2^2$ denotes a sum of the squares of the elements in a matrix $\mathbf{A}$.

## VI. EVALUATION

In this section, we quantitatively and qualitatively evaluate the performance of our proposed FPGA-based registration

pipeline in terms of accuracy (Sec. VI-A), computation time (Sec. VI-B), resource utilization (Sec. VI-C and VI-D), and power consumption (Sec. VI-E).

### A. Accuracy

We first compare the registration accuracy of our FPGA-based registration pipeline in comparison with the PyTorch implementation of PointNetLK and ICP [1]. Note that PyTorch uses the single-precision floating-point format. As done in the original paper [2], we trained PointNetLK on the training sets of the first 20 object classes (airplane to lamp) in ModelNet40 [27] consisting of 5144 CAD models, and tested on the test sets of the same 20 classes consisting of 1202 CAD models.

For each CAD model, we extracted a template point cloud $\mathcal{P}_T$ from the vertices, and normalized the point coordinates to fit inside a unit cube. We rotated $\mathcal{P}_T$ around a random axis by a constant angle $0° \leq \theta \leq 90°$, and then translated it by a random vector with each element uniformly distributed on $[0, 0.3)$ to generate a source $\mathcal{P}_S$. From a difference $\mathbf{\Delta G} = \mathbf{G}\mathbf{G}^{*-1}$ between a ground-truth transform $\mathbf{G}^*$ and an output $\mathbf{G}$, we computed (isotropic) rotational and translational errors. A translational error $\varepsilon_t$ is obtained as a norm of the translation part in $\mathbf{\Delta G}$, i.e., $\varepsilon_t = |(\mathbf{\Delta G})_{0:3,3}|_2$ (Fig. 6 left). By taking a norm of a rotation part in a 6D twist $\mathbf{\Delta \xi} = \log(\mathbf{\Delta G})^\vee \in \mathbb{R}^6$, a rotational error $\varepsilon_r$ (Fig. 6 right) is obtained as $\varepsilon_r = |\mathbf{\Delta \xi}_{0:3}|_2$ ($^\vee$ and log are the inverse of $^\wedge$ and exp in Sec. III-A).

We downsampled (or upsampled) the input point clouds as necessary to fix the number of points $N$ to 1024 for all data samples. In both ICP and PointNetLK, the maximum number of iterations was set to 20 for a fair comparison. Fig. 6 shows the results with varying initial angles.

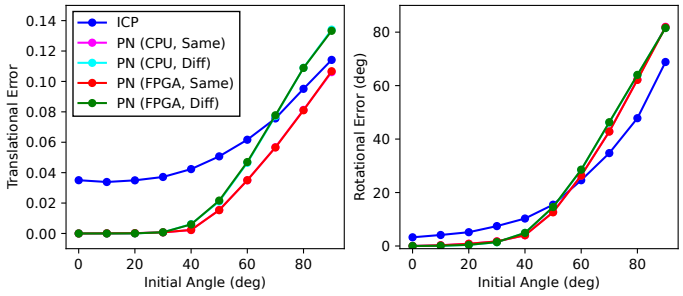


Fig. 6. Registration errors of our registration pipeline (FPGA) in comparison with vanilla PointNetLK and ICP

Our proposed registration pipeline coupled with an FPGA accelerator (magenta) achieves almost the same accuracy as the software implementation (red), and provides better accuracy than ICP for $\theta \leq 50°$. For $\theta \geq 60°$, PointNetLK did not converge to correct solutions and showed larger rotational errors than ICP. This is an expected behavior; during training, we created a rigid transform $\exp(\boldsymbol{\xi}^\wedge) \in \mathrm{SE}(3)$ between point clouds from a 6D twist vector $\boldsymbol{\xi}$ with norm less than 0.8. In other words, PointNetLK was never trained on point cloud pairs with initial angles larger than 0.8 radians (45.9°).

We also trained PointNetLK on the training sets of the first 20 classes (airplane to lamp) and tested on the test sets of the last 20 classes (laptop to xbox) containing 1266 CAD models. While it (cyan, green) shows a larger translational error than PointNetLK trained and tested with the same classes (magenta, red) for $\theta \geq 60°$, it still achieves the same level of accuracy especially in the rotation estimation. Besides, for $\theta \leq 50°$, the registration error is lower than ICP and closer to that of PointNetLK in the previous setting. This indicates that PointNetLK has a generalization ability to align objects which are unseen during training. As shown in Fig. 6, our registration pipeline (green) has almost the same accuracy as the software counterpart (cyan), meaning that it yields faster computation time without compromising the accuracy. For qualitative analysis, we visualize the registration results obtained from our registration pipeline for ModelNet40 and Stanford bunny [28] in Figs. 1 and 7, respectively.

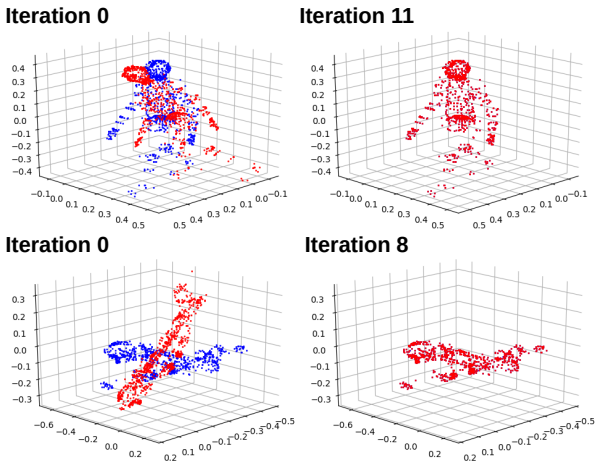


Fig. 7. Examples of outputs from our proposed FPGA-accelerated registration pipeline (ModelNet40). To show the generalization ability, PointNetLK trained on the first 20 object classes (airplane to lamp) was employed for the person model (top), whereas the one trained on the last 20 categories (laptop to xbox) was used for the airplane model (bottom). Red and blue points represent the source and template, respectively. The initial rotation angle was set to 60°.

### B. Computation Time

Our FPGA-based registration pipeline is evaluated in terms of the computation time to highlight its significant advantage over ICP. Fig. 8 shows the results with the varying number of input points $N$ from 128 to 4096. We used the table category in ModelNet40 and plotted an average wall-clock time for registration. We included the data transfer overhead between PS–PL for a fair comparison. The initial angle $\theta$ is set to 30°. We also note that PointNetLK was trained on the first 20 categories in ModelNet40, which do not include the table category. The wall-clock time increases linearly in PointNetLK and quadratically in ICP, which reflects the $O(N)$ and $O(N^2)$ computational complexity of PointNetLK and ICP; PointNetLK provides a better performance advantage over ICP especially with a larger $N$. For $N = 1024$, the CPU version of PointNetLK was 1.36x slower than ICP (5.47s and 4.04s). Our FPGA-based pipeline (red) took only 366ms per input, which was 14.98x faster compared to the CPU version (green), and eventually lead to 11.04x speedup than ICP (blue). As shown in Fig. 8, we obtained better results for $N = 4096$: compared to ICP, the CPU version was 3.26x faster (71.16s and 21.82s), and

our FPGA-based pipeline was 69.60x faster, which is attributed to the 21.34x speedup (21.82s to 1.02s).

Fig. 9 shows the breakdown of processing time for our registration pipeline and the PyTorch implementation of PointNetLK. We set the initial angles $\theta$ to 30° (first two rows) and 60° (last two rows). PointNet feature extraction (red + green, red refers to line 7, and green refers to lines 2-3 in Alg. 1) is a major bottleneck, accounting for 91.90% ($\theta = 30°$) and 93.29% ($\theta = 60°$), which were reduced to 58.01% and 57.96% by FPGA acceleration.

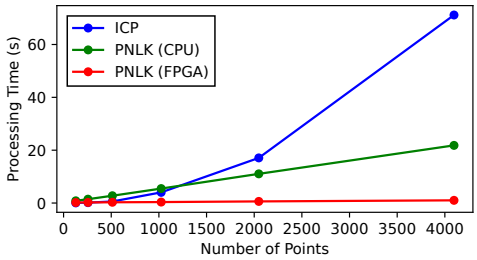


Fig. 8. Processing time of our registration pipeline (FPGA) in comparison with vanilla PointNetLK and ICP

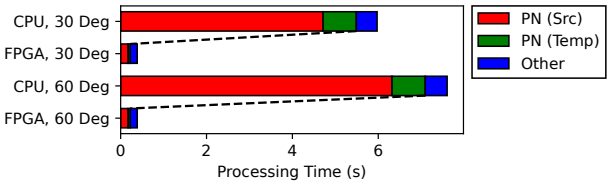


Fig. 9. Breakdown of the processing time for our registration pipeline (FPGA) in comparison with vanilla PointNetLK

### C. Effects of Quantization

This section analyzes a relationship between the number of quantization bits and accuracy. Fig. 10 shows the registration errors evaluated under five different numbers of quantization bits from 16 to 32. Table III summarizes the FPGA resource utilization for Xilinx ZCU104 board. Each design uses the $2n$-bit fixed-point format with $n$-bit integer part and $n$-bit fraction part ($n = 8, 10, 12, 14, 16$). We trained PointNetLK on the first 20 object classes and tested with the table class in ModelNet40.

As apparent in Fig. 10, the 16-bit quantized version exhibited larger errors than the others. In contrast, for $0° \leq \theta \leq 50°$, the 20-bit version produced nearly the same results as the 32-bit version. Even for $\theta \geq 60°$, the reduction from 32 to 20-bit only introduced a slight accuracy loss. Notably, the DSP usage was halved by reducing from 32 to 24-bit (Table III). The reduction from 24 to 20-bit further halved the DSP usage (24.07% to 12.56%) and increased the LUT usage (9.91% to 12.87%). The results indicate that the 20-bit version strikes the best balance between accuracy and resource consumption. As seen in Table IV, the 20-bit version fits within a low-cost FPGA, Avnet Ultra96v2, whereas the 32-bit version cannot be implemented due to the shortage of DSP and LUT resources.

### D. Effects of Design Optimization

Here, we discuss the effects of design optimizations described in Section IV on the performance and FPGA resource

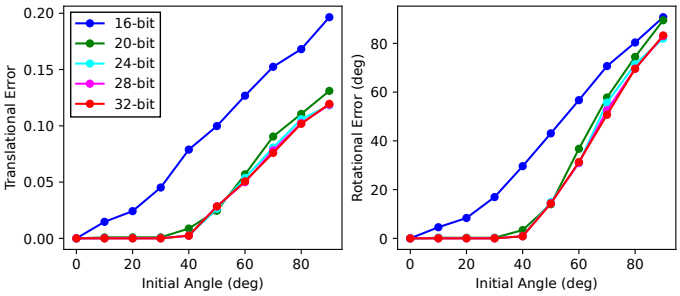


Fig. 10. Registration errors and the number of quantization bits

TABLE III
FPGA RESOURCE UTILIZATION AND QUANTIZATION (XILINX ZCU104)

| # of Bits | BRAM (%) | DSP (%) | FF (%) | LUT (%) |
|---|---|---|---|---|
| 32 | 55.13 | 48.50 | 5.46 | 16.31 |
| 28 | 55.13 | 48.09 | 4.73 | 13.70 |
| 24 | 44.87 | 24.07 | 4.05 | 9.91 |
| 20 | 44.23 | 12.56 | 3.85 | 12.87 |
| 16 | 27.40 | 12.21 | 2.83 | 7.74 |

utilization for Xilinx ZCU104 board. In addition to the final design, we also consider a design without inter-layer pipelining and a naive design with no optimization as a baseline. Fig. 11 plots the processing time with varying point cloud sizes from $N = 128$ to $N = 16384$, and Table V compares the resource utilization. In Fig. 11, we observe a linear increase of the processing time, and the naive design (blue) is 3.49x slower than the CPU (black) for $N = 1024$ (363.49ms and 1267.08ms). By exploiting the intra-layer parallelism, the design (green) attains a speedup of 34.46x ($N = 1024$) compared to the unoptimized version (1267.08ms to 36.77ms) at the expense of 14.38x increase in the DSP usage (3.07% to 44.16%). The intra-layer pipelining allows a further speedup of 4.29x (green and blue, 36.77ms to 8.58ms) with a few additional resources, by overlapping data transfer and computation. This leads to a total performance improvement of 147.68x and 42.36x compared to the unoptimized version and CPU counterpart for $N = 1024$, respectively.

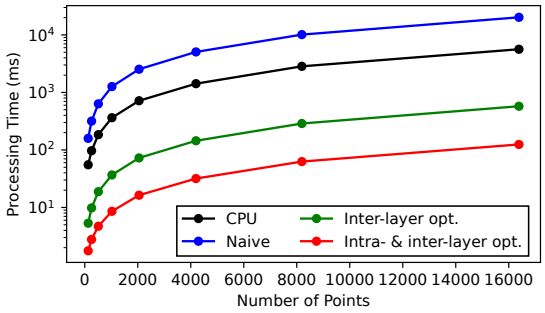


Fig. 11. Comparison of inference time of PointNet

### E. Power Consumption

According to the report from Xilinx Vivado 2020.2, the power consumption of our accelerator on Xilinx ZCU104 was 722mW, which is a quarter of the CPU one.

TABLE IV
FPGA RESOURCE UTILIZATION (AVNET ULTRA96V2)

| # of Bits | BRAM (%) | DSP (%) | FF (%) | LUT (%) |
|---|---|---|---|---|
| 20 | 64.81 | 60.28 | 12.81 | 43.52 |
| 32 | 79.63 | 100.00 | 22.35 | **238.77** |

TABLE V
FPGA RESOURCE UTILIZATION AND DESIGN OPTIMIZATION (ZCU104)

| Design | BRAM (%) | DSP (%) | FF (%) | LUT (%) |
|---|---|---|---|---|
| Naive | 45.99 | 3.07 | 0.82 | 3.73 |
| Intra-layer | 54.17 | 44.16 | 4.56 | 10.66 |
| Inter- & intra-layer | 55.13 | 48.50 | 5.46 | 16.31 |

## VII. CONCLUSION

We presented the first study of the resource-efficient FPGA-accelerated pipeline for 3D point cloud registration based on PointNetLK. We focused on a PointNet-based feature extraction part as it constitutes the major bottleneck, and designed a custom accelerator for both low-cost and mid-range FPGAs (Avnet Ultra96v2 and Xilinx ZCU104). We exploited both intra- and inter-layer parallelism in PointNet to fully optimize the IP core design, and also achieved $O(N)$ computational complexity and $O(1)$ memory consumption by processing a point one-by-one in a pipelined fashion. Experimental results demonstrated that our FPGA-based registration pipeline achieved up to 21.34x and 69.60x speedup compared to the software implementation of vanilla PointNetLK and ICP, respectively, while consuming only 722mW and maintaining the accuracy. Besides, our registration pipeline showed a better scalability than ICP and a generalization ability to unseen object categories. As a future work, it is necessary to compare with the other edge computing platforms, e.g., NVidia Jetson Nano and Raspberry Pi Zero, to investigate the effectiveness of our FPGA-based approach and make evaluation results more comprehensive. The registration speed could be further improved by optimizing the neural architecture (e.g., reducing the number of MLP layers from five to three) instead of using the vanilla PointNet as-is.

**Acknowledgments** This work was supported by JST CREST Grant Number JPMJCR20F2 and JSPS KAKENHI Grant Number JP22J21699.

## REFERENCES

[1] P. J. Besl and N. D. McKay, "A Method for Registration of 3-D Shapes," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 14, no. 2, pp. 239–256, Feb. 1992.

[2] Y. Aoki, H. Goforth, R. A. Srivatsan, and S. Lucey, "PointNetLK: Robust & Efficient Point Cloud Registration using PointNet," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019, pp. 7156–7165.

[3] G. Elbaz, T. Avraham, and A. Fischer, "3D Point Cloud Registration for Localization Using a Deep Neural Network Auto-Encoder," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 4631–4640.

[4] A. Kurobe, Y. Sekikawa, K. Ishikawa, and H. Saito, "CorsNet: 3D Point Cloud Registration by Deep Neural Network," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 3960–3966, Feb. 2020.

[5] X. Li, J. K. Pontes, and S. Lucey, "PointNetLK Revisited," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 12 763–12 772.

[6] P. Biber and W. Straßer, "The Normal Distributions Transform: A New Approach to Laser Scan Matching," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2003, pp. 2743–2748.

[7] A. V. Segal, D. Haehnel, and S. Thrun, "Generalized-ICP," in *Proceedings of the Robotics: Science and Systems Conference (RSS)*, June 2009.

[8] J. Yang, H. Li, D. Campbell, and Y. Jia, "Go-ICP: A Globally Optimal Solution to 3D ICP Point-Set Registration," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 38, no. 11, pp. 2241–2254, Nov. 2016.

[9] J. Li, H. Zhan, B. M. Chen, I. Reid, and G. H. Lee, "Deep Learning for 2D Scan Matching and Loop Closure," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept. 2017, pp. 763–768.

[10] M. Valente, C. Joly, and A. de La Fortelle, "An LSTM Network for Real-Time Odometry Estimation," in *Proceedings of the IEEE Intelligent Vehicles Symposium (IV)*, June 2019, pp. 1434–1440.

[11] L. Ding and C. Feng, "DeepMapping: Unsupervised Map Estimation From Multiple Point Clouds," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019, pp. 8642–8651.

[12] V. Sarode, X. Li, H. Goforth, Y. Aoki, R. A. Srivatsan, S. Lucey, and H. Choset, "PCRNet: Point Cloud Registration Network using PointNet Encoding," arXiv Preprint 1908.07906, Aug. 2019.

[13] G. D. Pais, S. Ramalingam, V. M. Govindu, J. C. Nascimento, R. Chellappa, and P. Miraldo, "3DRegNet: A Deep Neural Network for 3D Point Registration," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020, pp. 7193–7203.

[14] Y. Wang and J. M. Solomon, "Deep Closest Point: Learning Representations for Point Cloud Registration," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2019, pp. 3523–3532.

[15] W. Lu, G. Wan, Y. Zhou, X. Fu, P. Yuan, and S. Song, "DeepVCP: An End-to-End Deep Neural Network for Point Cloud Registration," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Feb. 2019, pp. 12–21.

[16] Y. Wang and J. M. Solomon, "PRNet: Self-Supervised Learning for Partial-to-Partial Registration," in *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, Dec. 2019, pp. 8814–8826.

[17] Z. J. Yew and G. H. Lee, "RPM-Net: Robust Point Matching Using Learned Features," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020, pp. 11 824–11 833.

[18] C. Choy, W. Dong, and V. Koltun, "Deep Global Registration," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020, pp. 2514–2523.

[19] K. Fu, S. Liu, X. Luo, and M. Wang, "Robust Point Cloud Registration Framework Based on Deep Graph Matching," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 8893–8902.

[20] T. Min, E. Kim, and I. Shim, "Geometry Guided Network for Point Cloud Registration," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 7270–7277, Oct. 2021.

[21] Y. Sekikawa and T. Suzuki, "Tabulated MLP for Fast Point Feature Embedding," arXiv Preprint 1912.00790, Dec. 2019.

[22] A. Kosuge, K. Yamamoto, Y. Akamine, and T. Oshima, "An SoC-FPGA-Based Iterative-Closest-Point Accelerator Enabling Faster Picking Robots," *IEEE Transactions on Industrial Electronics*, vol. 68, no. 4, pp. 3567–3576, Mar. 2020.

[23] Q. Deng, H. Sun, F. Chen, Y. Shu, H. Wang, and Y. Ha, "An Optimized FPGA-Based Real-Time NDT for 3D-LiDAR Localization in Smart Vehicles," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 9, pp. 3167–3171, July 2021.

[24] M. Eisoldt, M. Flottmann, J. Gaal, P. Buschermöhle, S. Hinderink, M. Hillmann, A. Nitschmann, P. Hoffmann, T. Wiemann, and M. Porrmann, "HATSDF SLAM – Hardware-accelerated TSDF SLAM for Reconfigurable SoCs," in *Proceedings of the European Conference on Mobile Robots (ECMR)*, Aug. 2021.

[25] T. D. Barfoot, *State Estimation for Robotics*. Cambridge University Press, 2017.

[26] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 652–660.

[27] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3D ShapeNets: A Deep Representation for Volumetric Shapes," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[28] G. Turk and M. Levoy, "The Stanford 3D Scanning Repository," http://graphics.stanford.edu/data/3Dscanrep/.