2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

Memory Efficient Point Cloud Registration Accelerator on FPGA

Qiong Chang[1], Dongqi Cai[2], Ran Dong[3] and Junpei Zhong[4]

Abstract— **Point cloud registration, which aligns multiple datasets into a unified coordinate system, is critical for mobile applications such as 3D SLAM and autonomous driving. Among existing methods, Iterative Closest Point (ICP) remains a widely used method for rigid registration due to its robustness and simplicity. However, its performance on mobile platforms is hindered by iterative computations and limited memory resources. This paper proposes a high-performance ICP registration framework implemented on FPGA. Building upon an efficient GPU-based method named VAN-ICP, our FPGA-based ICP accelerator achieves greater memory efficiency and faster processing speed, making it ideal for resource-constrained mobile platforms. Experimental results demonstrate a speedup of over 1.5× compared to mobile GPU-based implementations and a 99% reduction in memory usage, validating the effectiveness of the proposed approach for real-world point cloud registration on edge platforms. Beyond these improvements, the proposed framework also facilitates advancements in robotic vision technologies by enabling more accurate and efficient perception under stringent hardware constraints.**

I. INTRODUCTION

In robotic systems, understanding the 3D structure of the environment using point clouds plays a crucial role. Compared to 2D images, 3D point clouds provide richer spatial information, enabling robots to accurately recognize object shapes, estimate distances, and navigate complex scenes. As a fundamental technique, the *point cloud registration* aligns multiple point cloud datasets captured from different viewpoints into a unified coordinate system. This process supports robots to build consistent and comprehensive 3D maps, fuse sensor data over time, and model their surroundings more effectively.

The Iterative Closest Point (ICP) algorithm is a representative method for rigid registration of two 3D point clouds. It iteratively estimates the optimal transformation matrix that minimizes the distance between corresponding points in the source and target datasets. The computational complexity of ICP is largely dominated by the nearest neighbor search (NNS) step in each iteration. For a source point cloud with N points and a target with M points, a brute-force NNS requires $O(MN)$ time per iteration, although this can be reduced to $O(N \log M)$ using spatial data structures such as kd-trees. Recently, Wang et al. [1] proposed a GPU-based strategy to accelerate the ICP operation by introducing an approximate NNS method. Their approach voxelizes the target point cloud and fills voxels around the source points, transforming the costly global search into a more efficient, parallelizable local search. This method achieves significant speedups over existing techniques while preserving high registration accuracy. However, it is designed for general-purpose GPUs and has high memory demands, consuming approximately 2GB of GPU memory for a single frame-to-frame registration. This restricts its use on memory-limited edge devices and presents scalability challenges when processing large-scale point cloud data in real-world applications. In contrast to GPU, FPGA has the advantages of high reconfigurability and low power consumption. It allows for more precise control of hardware resources, resulting in providing flexibility and efficiency in circuits design to the specific needs of algorithms.

In this paper, we present an FPGA-based implementation of the VAN-ICP algorithm for point cloud registration. Our design incorporates a voxelization-based pre-traversal technique, which significantly enhances memory utilization and data management efficiency. Leveraging a mid-range embedded-class FPGA, we demonstrate real-time registration performance for point clouds containing tens of thousands of points, achieving throughput that surpasses a desktop GPU. These results highlight the potential of FPGA platforms to deliver high-performance, resource-efficient solutions for computationally intensive 3D vision tasks. The contributions of this paper can be outlined as follows:

- We present a real-time FPGA-based framework for ICP registration, enabling high-throughput alignment of 3D point clouds under resource-constrained environments.
- We propose a pre-traversal technique that drives the voxel dilation strategy of ICP, ensuring effective utilization and management of on-chip resources.
- We further design and integrate an FPGA-optimized singular value decomposition (SVD) module to accelerate the rigid body transformation step in ICP, achieving low-latency and high-precision computation.

The remainder of this paper is structured as follows. Section II provides an overview of point cloud registration and outlines the computational steps of the ICP algorithm. Section III reviews existing approaches to accelerating ICP. Section IV details the proposed design and hardware architecture. Section V describes the experimental setup and reports the results. Finally, Section VI concludes the paper with a summary of the key contributions and directions for

Qiong Chang is with the School of Computing, Institute of Science Tokyo, Tokyo, 152-8550, Japan.

Dongqi Cai is with the School of Integrated Circuit, Nanjing University, Soochow, 215163, China.

Ran Dong is with the School of Engineering, Chukyo University, Aichi, 470-0393, Japan.

Junpei Zhong is with Department of Digital Innovation and Technology, Technological and Higher Education Institute of Hong Kong, Hong Kong, China. Junpei Zhong is the corresponding author. jonizhong@gmail.com

Qiong Chang and Dongqi Cai contributed equally to this work.

future work.

II. Related Work

In recent years, numerous studies have focused on accelerating the processing of the ICP algorithm through various approaches to promote the practical applications of point cloud technology.

Zhang et al. treated the classic ICP algorithm as a Maximization-Minimization (MM) algorithm [3], using Anderson acceleration to speed up the algorithm's convergence. This approach reduces the number of iterations to improve the algorithm's processing speed and minimizes the error parameters based on the Welsch function introduced in the method.

Li et al. used a K-D tree [4] data structure to organize scattered point cloud data, improving the efficiency of nearest neighbor search in the algorithm. However, the K-D tree-based search process requires a large number of non-continuous memory accesses, placing high demands on memory bandwidth.

Koide et al. [5] proposed VGICP, which extends the Generalized ICP (GICP) algorithm by dividing three-dimensional space into multiple voxel grids to improve the efficiency of nearest neighbor search in the algorithm. Also, they combined VGICP with GPUs to further enhance the algorithm's execution speed [6]. However, when the point cloud size is large or when there is little overlap between the two point clouds to be registered, the effectiveness of the above acceleration methods decreases.

Chang et al. [1], [2] proposed an expansion strategy based on voxelization, which transmits the index of voxel blocks containing target points to voxel blocks without target points, transforming the nearest neighbor search into local searches, thus further improving the processing speed of the algorithm. However, this method is based on GPU implementation, which results in higher power consumption and is less suitable for use in small mobile devices.

Atsutake et al. [7] proposed a hierarchical graph-based KNN search method and the corresponding hardware sorting network to accelerate the ICP algorithm. This method organizes data points into a graph structure with multiple hierarchical levels, achieving faster and more efficient searches by progressively narrowing the search space, which offers a significant advantage over traditional K-D tree search methods. Li, Zheng, and Xiao, based on an approximate K-D tree structure, proposed an efficient parallel sorting circuit [8] to accelerate the sorting of large amounts of data during tree construction, while simultaneously computing the Euclidean distances of all points in the subspace. However, this approximate K-D tree-based method is prone to finding incorrect nearest neighbor target points, which leads to the ICP algorithm requiring more iterations.

Wang et al. [9] proposed an improved Local Sensitive Hashing (LSH) method, which can assign similar points to the same hash bucket (low-dimensional space), thereby reducing the search space and accelerating kNN search speed. This method requires the selection of an appropriate hash function to ensure that similar points are assigned to the same bucket.

Shi et al. [10] designed multiple units to store point cloud data. These units perform parallel nearest neighbor searches, and the sum of the absolute differences in the coordinates of the source and target points in three dimensions replaces the traditional Euclidean distance calculation. This method of using the sum of coordinate differences instead of Euclidean distance calculation may lead to larger errors and increase the number of iterations of the algorithm.

III. Background

A. Basic ICP

The Iterative Closest Point (ICP) algorithm [11] is one of the most widely used methods for rigid registration of 3D point clouds due to its simplicity and robustness. Its goal is to align a source point cloud with a target point cloud through iterative estimation of the rigid body transformation. The algorithm proceeds in four main stages:

Nearest Neighbor Search For each point the source point cloud $P = \{p_1, p_2, ..., p_m\}$, the algorithm identifies the closest corresponding point in the target point cloud $Q = \{q_1, q_2, ..., q_n\}$. This results in a matched point set $Q' = \{q'_1, q'_2, ..., q'_m\}$, where each q'_i is the nearest neighbor of p_i. These correspondences serve as the foundation for estimating the transformation between the two point clouds.

Computation of the Rigid Transformation To align the source and target point clouds, the rigid transformation consisting of a rotation matrix R and a translation vector T is computed. First, the center of mass p_s of the source cloud and its matched target point q_t are calculated. Then, the covariance matrix H is constructed:

$$H = \sum_{i=1}^{m} (p_i - p_s)(q'_i - q_t)^T \tag{1}$$

Singular value Decomposition (SVD) is applied to H, yielding $H = U\Sigma V^T$. The optimal rotation and translation are then obtained as:

$$R = VU^T, T = q_t - Rp_s. \tag{2}$$

Rigid Transformation of the Source Cloud Using the computed transformation, the source point cloud is updated as:

$$P' = RP + T \tag{3}$$

This moves the source point cloud closer to alignment with the target.

Iteration and Convergence The above steps are repeated iteratively until either the maximum iteration count is reached or the alignment error converges below a specified threshold. At convergence, the source and target point clouds are considered registered within the rigid body transformation model

B. VAN-ICP

The VAN-ICP [1] serves as the foundation of our accelerator, as it has been shown to achieve significantly higher processing speed on GPUs compared with current registration methods [12], [13], [14].

Voxelization The target point cloud is first voxelized into a 3D grid. Each voxel stores the points that fall within it, effectively transforming the continuous global search into a structured grid representation.

Voxel Dilation At the early stages of ICP iterations, only a limited number of voxels overlap. To address this, VAN-ICP dilates the occupied voxels outward to their neighboring voxels. This process expands the overlap region, allowing more source points to locate corresponding target voxels even in the initial iterations, and efficiently propagates voxel information across adjacent regions.

Local Approximate Nearest Neighbor Search After voxelization and dilation, each source point only needs to search within its assigned voxel or its dilated neighbors, rather than across the entire target point cloud. If the voxel is non-empty, a local search is performed (O(n) complexity within the voxel). If the voxel is empty, the algorithm falls back to a global search to ensure correctness, though such cases become increasingly rare as iterations proceed.

GPU Parallelism The nearest neighbor search of each source point is mapped to an individual GPU thread. Leveraging voxel-based locality, these approximate nearest neighbor searches are highly parallelizable, avoiding the global kd-tree traversals or brute-force distance computations that dominate traditional ICP.

However, to meet the parallel execution requirements of GPU threads, VAN-ICP must pre-allocate ample memory for each voxel to support dilation. This tight coupling between point cloud size and memory capacity reduces the flexibility of VAN-ICP in practical applications. Furthermore, for extremely large-scale point clouds, the inability to accommodate dilation may lead to registration failures.

IV. Hardware Implementation

We present a memory-efficient FPGA accelerator for ICP, designed on top of the VAN-ICP strategy. In our design, memory is allocated dynamically according to the number of points within each voxel block, while dedicated hardware modules are developed to accelerate different stages of the ICP algorithm. This approach not only reduces memory consumption but also accelerates computation. Each module in the accelerator communicates through pulse signals: when a module completes its computation, it issues a valid pulse signal that triggers the subsequent module to begin execution. For the dividers, matrix multipliers, and sine/cosine generators required in the pipeline, we implemented customized designs tuned to the required precision. This customization minimizes logic resource usage and reduces computation latency. Our accelerator particularly focuses on the computationally intensive nearest neighbor search, while also optimizing the SVD decomposition for solving the rigid transformation matrix. Furthermore, point coordinates across different dimensions are stored in separate RAM blocks, enabling full exploitation of FPGA parallelism.

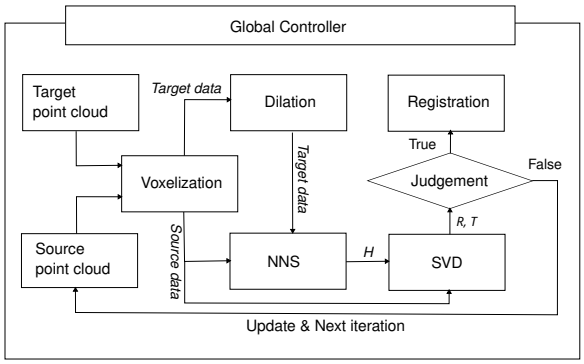

Fig. 1. Overall design of the ICP accelerator.

The overall design of the proposed system is illustrated in Figure 1. The process begins by voxelizing both the target and source point clouds. The voxelized target point cloud then undergoes a dilation operation to increase the overlap region. Subsequently, nearest neighbor search is performed between the expanded target voxels and the voxelized source points. Based on the correspondences obtained, a matrix H is constructed and decomposed via singular value decomposition (SVD) to estimate the rigid transformation matrices R and T. The source point cloud is then transformed using R and T to align it with the target. Finally, the algorithm checks whether the maximum number of iterations has been reached or the error has converged. If either condition is satisfied, the registration process terminates; otherwise, the source point cloud is updated and the next iteration begins.

A. Voxelization and Memory Management

Voxelization: The three-dimensional space is partitioned into 2^{3N} voxel blocks, indexed from 0 to $2^{3N} - 1$. Each voxel block has a unit side length, and the total side length of the voxel space along each dimension is 2^N. As the voxel resolution increases, the number of points contained in each block decreases, thereby reducing the computational cost of nearest-neighbor search for source points. However, excessive partitioning may lead to local optima. In addition, the limited parallel resources of the FPGA restrict the number of voxel blocks that can be simultaneously processed. To balance efficiency and hardware feasibility, N is set to 3. The hardware architecture for voxelization is illustrated in Figure 2. A counter-driven state machine generates the read address signal (16-bit unsigned) and the read-enable signal (1-bit) to access point coordinates stored in RAM. The coordinates x, y, z of each point are read sequentially, with each dimension stored in a separate RAM to enable parallel access. Each coordinate is represented using a 19-bit fixed-point format, consisting of 1 sign bit, 2 integer bits, and 16 fractional bits. In this circuit design, an adder module is first employed to guarantee that each coordinate dimension of the input point remains strictly positive. The adjusted values are then processed through a shift-register mapping

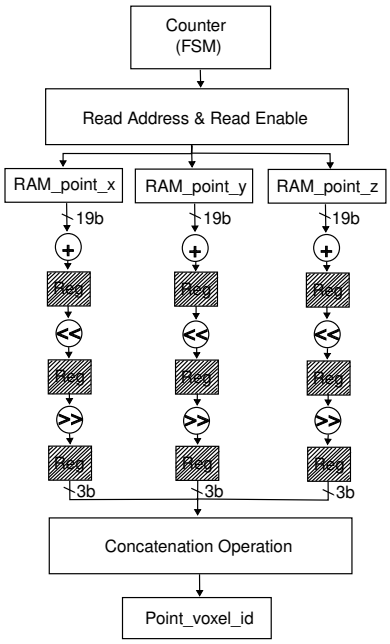

Fig. 2. Implementation for point cloud voxelization.

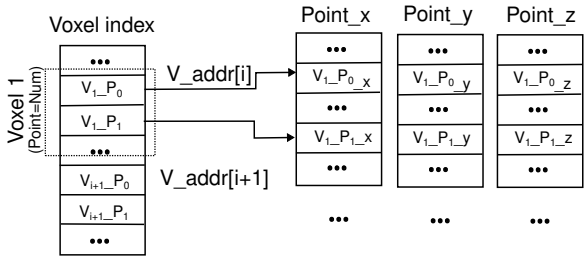

Fig. 3. Memory assignment.

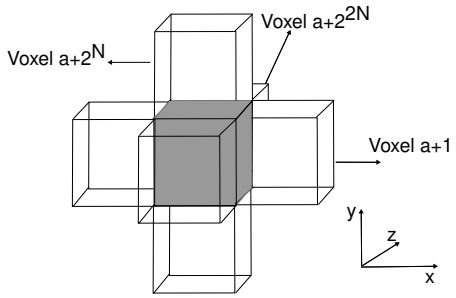

Fig. 4. Voxel dilation in six directions.

unit, which normalizes them into the range $(0, 2^N)$. Each dimension is represented in hardware using 16-bit fixed-point fractional precision. To obtain compact integer indices, a secondary shift register removes the 16 fractional bits, producing x_{id}, y_{id} and z_{id} as 3-bit outputs. Finally, a bitwise concatenation unit combines these three fields to form a 9-bit voxel block index corresponding to the current point. The actual computational relationship is shown in Equation (4):

$$Point\_voxel\_id = x_{id} \cdot 2^{2N} + y_{id} \cdot 2^N + z_{\mathrm{id}} \qquad (4)$$

Memory Management: To maximize the efficiency of FPGA on-chip memory, we propose a pre-traversal technique that optimizes memory allocation across voxel blocks. The design employs two register sets, *Point* and *V_addr*, together with a memory block, *Voxel_index*. Each register set contains as many registers as voxel blocks, with one register assigned to each block. The *Point* registers record the number of points in each voxel block. During pre-traversal, point coordinates stored in RAM are read, voxelized, and mapped to their corresponding voxel blocks. When a point is assigned to a voxel, the corresponding *Point* entry is incremented. After voxelization, the total number of points in each voxel block can be directly obtained from the *Point* set. The *V_addr* registers hold the head address of each voxel block. The *Voxel_index* memory stores the access addresses of the point coordinates within each voxel. As illustrated in Figure 3, *V_addr[i]* stores the head address of voxel i, from which the access addresses of all points in voxel i can be retrieved from *Voxel_index*. For instance, $V_i\_P_0\_index$ denotes the access address of the first point in voxel i, determined by traversing the point coordinates. The head address of voxel $i+1$ is computed as the sum of the head address of voxel i and the number of points in voxel i (i.e., *Point[i]*). The head address of voxel 0 is initialized to 0.

By integrating *Point*, *V_addr*, and *Voxel_index*, the system can directly access the coordinates of any point within any voxel block. This mechanism enables memory allocation for voxel blocks according to the actual number of points they contain, thereby minimizing memory overhead and ensuring efficient utilization.

B. Dilation

After voxelizing the source and target point clouds, certain voxel blocks may contain source points but no corresponding target points due to the incomplete overlap between the two clouds. In such cases, nearest neighbor search cannot be directly applied within those blocks. To address this issue and ensure that each source point can identify a corresponding nearest neighbor within its voxel block, a dilation operation is applied to the voxelized target point cloud.

The dilation operation involves encoding the indices of root voxels (voxel blocks that contain target points) and propagating them into adjacent empty voxels (voxel blocks without target points). As illustrated in Figure 4, for a center voxel a, dilation is performed along six directions: x, y, z, $-x$, $-y$, $-z$. The index of the voxel block adjacent to voxel a is computed as follows: in the positive x-direction it is $a+1$, in the positive y-direction it is $a+2^N$, and in the positive z-direction it is $a+2^{2N}$.

The circuit diagram for dilating voxel a in the positive x-direction is shown in Figure 5. The process begins with a comparator that checks whether *Point[a]* (the register storing the number of target points within voxel a) is zero. If the value is zero, voxel a is empty and no dilation is performed; the value of *Point[a+1]*, corresponding to voxel $a+1$, remains unchanged. Otherwise, if voxel a is non-empty, an additional comparator evaluates *Point[a+1]*. If this value is

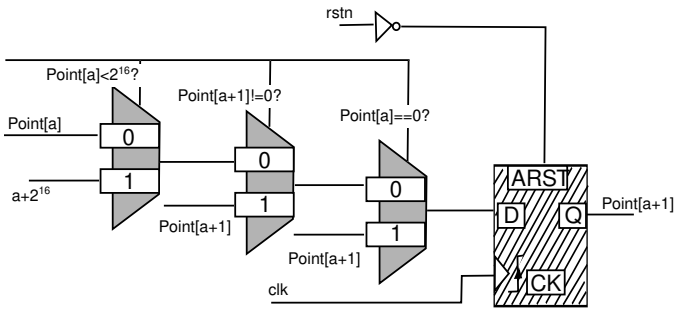

Fig. 5. Implementation of voxel block dilation along the positive x direction.

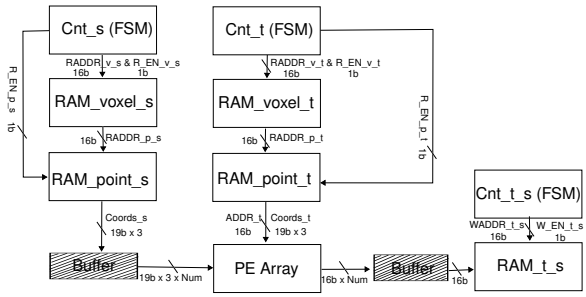

Fig. 6. Implementation of nearest neighbor search. *19b* denotes the bit width of each coordinate dimension. Since each point is three-dimensional, the total storage requirement is 19×3. *Num* indicates the number of source points processed in parallel during the nearest neighbor search. Note that in some cases, the number of source points requiring search may be smaller than the number of available PEs.

non-zero, voxel $a+1$ is also non-empty, and no dilation is required. If it is zero, the dilation operation is applied. For dilation of voxel a, it is further necessary to check whether $Point[a] < 2^{16}$. If so, voxel a is identified as a root voxel. In this case, its index a is offset by 2^{16} and the result is written to *Point[a+1]*. Otherwise, voxel a is classified as a dilated voxel derived from another root, and its value is directly propagated to *Point[a+1]*.

In this process, the root voxel is dilated across multiple directions to ensure that each empty voxel in the voxel space is associated with a neighboring root voxel. Consequently, each source point can restrict its nearest neighbor search to the corresponding voxel block, effectively transforming the original global search into a localized one.

Within the GPU-based VAN-ICP strategy, dilation is performed by storing the encoded root voxel index in the memory regions allocated to empty voxel blocks. However, since empty voxel blocks are assigned the same memory space as non-empty ones, a substantial portion of memory remains unused, leading to excessive consumption. In contrast, our approach leverages the *Point* register set, originally used during voxelization, to execute dilation. Each empty voxel block requires only a single register from the *Point* set along with a small lookup table, thereby significantly reducing memory usage.

C. Nearest Neighbor Search

After dilation of the target point cloud, the nearest neighbor search for the source points is performed. As illustrated in Figure 6, a PE Array comprising 200 processing elements (PEs) is employed to carry out the search for multiple source points within the same voxel block in parallel. Each PE is responsible for computing the nearest target point for a single source point within the voxel.

The read addresses (16-bit) and read-enable signals (1-bit) for the source point coordinates are pre-generated by the counter *Cnt_s*, which operates as a state machine. These addresses (16-bit, unsigned, integer only) are then used to access the source point coordinates stored in *RAM_voxel_s*. The retrieved coordinates are buffered and subsequently distributed to the individual PEs. For the target points within the same voxel block, the 16-bit read addresses are similarly obtained. Unlike the source points, the coordinates of the target points are not buffered. Instead, once the PE trigger signal is asserted, the buffered source point coordinates are simultaneously fed into the PEs, while the coordinates of all target points are sequentially broadcast. Specifically, in each clock cycle, one target point's coordinates (19-bit per dimension) along with its access address (16-bit) are delivered and shared across the entire PE Array.

Next, each PE computes the squared Euclidean distance between a source point and its candidate target points, and retains the access address corresponding to the target point with the smallest distance. The access address of the nearest neighbor target point is then output. When multiple PEs produce results simultaneously, their output addresses are buffered. These addresses are subsequently retrieved in sequence and stored in *RAM_t_s*, which records the access addresses of the nearest neighbor target points for all source points. Through this procedure, the nearest neighbor target addresses for each source point in the voxel blocks are obtained. By storing them in *RAM_t_s*, the coordinates of the nearest neighbor pairs can later be retrieved directly using the access addresses. Finally, the input matrix H for SVD decomposition is constructed according to Equation (1).

D. SVD Decomposition

After deriving the 3×3 matrix H from Equation (1), Singular Value Decomposition (SVD) is performed. In this work, the Jacobi method [15] is employed, which iteratively applies plane rotations to eliminate off-diagonal elements and thereby diagonalize the matrix. This approach is particularly efficient and reliable for small-scale matrices such as H.

The steps are as follows:

a) Calculate the rotation angles: Calculate the left rotation angle θ and the right rotation angle ϕ based on the elements of matrix H and Equation (5):

$$H = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{5}$$

$$\theta = \left(\tan^{-1}\left(\frac{a_{qp}+a_{pq}}{a_{qq}-a_{pp}}\right) - \tan^{-1}\left(\frac{a_{qp}-a_{pq}}{a_{qq}+a_{pp}}\right) \right)/2$$

$$\phi = \left(\tan^{-1}\left(\frac{a_{qp}-a_{pq}}{a_{qq}+a_{pp}}\right) + \tan^{-1}\left(\frac{a_{qp}+a_{pq}}{a_{qq}-a_{pp}}\right) \right)/2$$

Since the matrix of size 3x3 has three pairs of off-diagonal elements, it is necessary to calculate three left rotation angles and three right rotation angles. To eliminate the matrix elements a_{12} and a_{21}, when solving for θ_1 and φ_1, $q = 2$ and $p = 1$ in the formula; to eliminate the matrix elements a_{13} and a_{31}, when solving for θ_2 and φ_2, $q = 3$ and $p = 1$ in the formula; to eliminate the matrix elements a_{23} and a_{32}, when solving for θ_3 and φ_3, $q = 3$ and $p = 2$ in the equation.

b) Constructing the rotation matrix: Using the rotation angles obtained in the previous step and following equation, the Jacobi rotation matrix is constructed:

$$R = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cos\theta & \cdots & \sin\theta & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\sin\theta & \cdots & \cos\theta & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \tag{6}$$

The rotation matrix shares the same dimensionality as matrix H. For the Jacobi rotation matrix J_{ij} associated with the elements a_{ij} and a_{ji} (with $i < j$), the off-diagonal entries at positions (i, j) and (j, i) are set to $\sin\theta$ and $-\sin\theta$, respectively. The diagonal entries at (i, i) and (j, j) are assigned the value $\cos\theta$. All other diagonal elements are set to 1, and the remaining off-diagonal elements are zero.

We implement the arctangent calculator required for Equation (5), as well as the sine and cosine wave generators for Equation (6), using the classical *CORDIC* algorithm in hardware. The corresponding hardware architecture is illustrated in Figure 7. The *CORDIC* design relies exclusively on shift registers, adders (or subtractors), and look-up tables, making it highly resource-efficient. Its core principle is to iteratively approximate the desired rotation angle, where each incremental rotation is defined by a tangent value equal to a positive power of two. This property enables the multiplication operations in the iterative formula to be replaced by simple shift operations, efficiently realized with shift registers. Furthermore, the required angles corresponding to tangent values of powers of two are precomputed in software and stored in a look-up table, which is accessed during each iteration to guide the rotation process.

c) Obtain the singular matrix: The Jacobi rotation is applied to matrix H as follows, yielding the updated matrix H':

$$H' = J_3^{L^T} J_2^{L^T} J_1^{L^T} H J_1^R J_2^R J_3^R \tag{7}$$

From this transformation, the left singular matrix U_1 and the right singular matrix V_1 are obtained as:

$$\begin{aligned} U_1 &= J_1^L J_2^L J_3^L \\ V_1 &= J_1^R J_2^R J_3^R \end{aligned} \tag{8}$$

A single Jacobi rotation is typically insufficient to fully eliminate the off-diagonal elements of the matrix H. Therefore, the matrix is updated to H' after each rotation, and the procedure is iteratively applied until convergence. Upon completion, the singular value decomposition (SVD) yields the matrices U and V according to the following relation:

$$\begin{aligned} U &= U_1 U_2 \cdots U_k \\ V &= V_1 V_2 \cdots V_k \end{aligned} \tag{9}$$

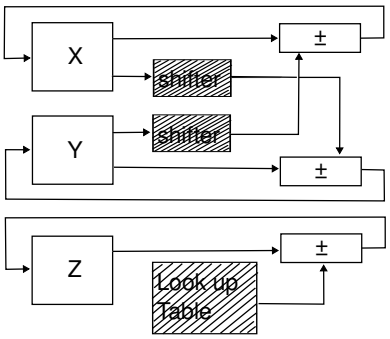

Fig. 7. Hardware architecture of the CORDIC algorithm.

d) Completed Registration: First, the rotation matrix R and translation matrix T are computed from the SVD-derived matrices U and V using Equation (2). Next, the source point cloud is transformed according to Equation (3) by applying R and T, and the updated coordinates are written back to the corresponding RAM. Subsequently, the mean squared error between the transformed source point cloud and the target point cloud is evaluated. If the error converges or the maximum number of ICP iterations is reached, the process terminates, signifying that registration is complete. Otherwise, the iteration proceeds until one of the stopping criteria is satisfied.

V. Experimental Setup and Results

This section presents the development platform and experimental setup, followed by a comparative evaluation of the proposed design against CPU, GPU, and FPGA-based implementations to demonstrate its effectiveness.

A. Experimental Setup

We employed the Xilinx Zynq UltraScale+ MPSoCs EV series AXU4EVB-E development board as the hardware acceleration platform. This board integrates a heterogeneous architecture comprising a Processing System (PS) and Programmable Logic (PL). The PS incorporates four ARM Cortex-A53 processors running at up to 1.2GHz with a two-level cache hierarchy. The PL provides 88k lookup tables, 176k flip-flops, and 4.5MB of on-chip BRAM. In our experiments, the FPGA was configured to operate at a clock frequency of 100MHz. The design was synthesized and implemented using the Vivado 2018.3 Integrated Development Environment, while functional verification of each hardware module was conducted in the MATLAB simulation environment. The error convergence threshold was set to $1e^{-4}$, and the maximum iteration limit was fixed at 50. To assess performance under realistic conditions, we used the Stanford Bunny point cloud dataset [16], consisting of

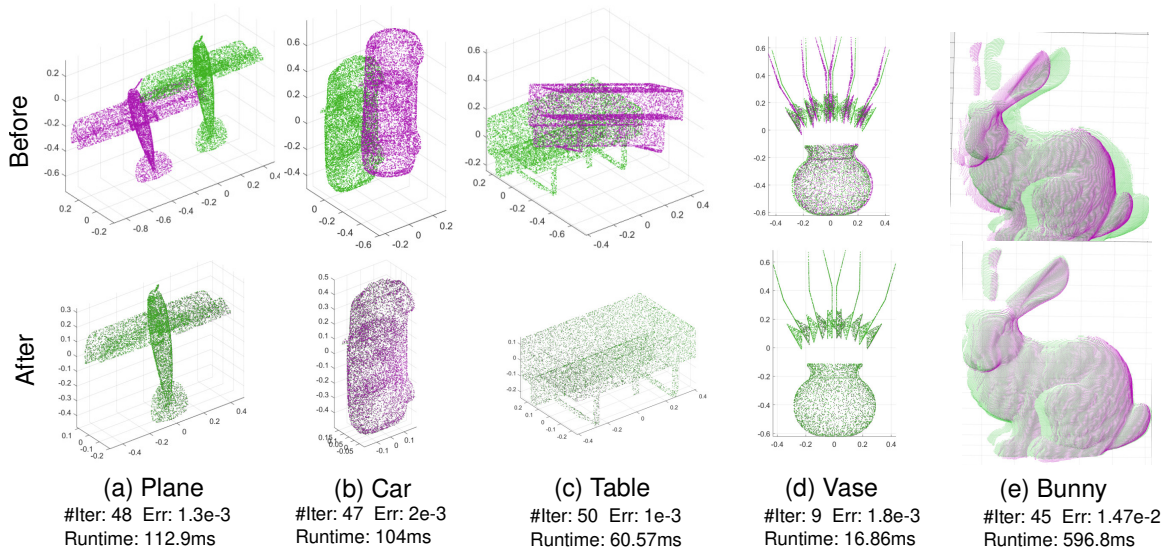

Fig. 8. Registration performance on various datasets.

approximately 40k points acquired by a 3D laser scanner. To further evaluate the generality of the design, we also tested on sampled point clouds from the ModelNet40 dataset [17], containing approximately 10k points. All point clouds were normalized to fit within a unit sphere. For testing, we applied random perturbations, including rotations of up to 20° and translations of up to 0.4 units along each axis, to generate the input sets.

B. Experimental Results

In this section, we evaluate the proposed method against several baseline designs, including the CPU and GPU based ICP implementations, and multiple FPGA-based implementations. Experiments are conducted on the standard ModelNet40 (10k) dataset and the Stanford Bunny dataset (40k), measuring both registration time and registration error. The comparison results shown in Table I, demonstrating that our method achieves speedups of up to 15x and 12x over the baseline PCL library on two datasets, substantially surpassing existing approaches. It is noteworthy that Open3D [20], one of the most widely used point cloud processing libraries, achieves performance comparable to ours on certain datasets. This can be attributed to the fact that, for simple data models, CPU-based algorithms often provide more accurate registration, thereby significantly reducing the number of required iterations. On the Bunny dataset, our method demonstrates a substantial acceleration effect. Importantly, since VAN-ICP is currently the fastest published ICP registration solution, our approach not only achieves additional speed gains but also significantly reduces memory consumption. Specifically, whereas their method requires approximately 2GB of memory, our design occupies only a few megabytes.

Figure 8 illustrates the registration results obtained on five representative models: *Plane*, *Car*, *Table*, *Vase*, and *Bunny*. Each subfigure provides a *before-and-after* comparison, where the top line shows the initial misaligned state

TABLE I

PERFORMANCE COMPARISON ACROSS MULTIPLE PLATFORMS

Dataset	Methods	Hardware	Runtime(s)	Speedup	Error
ModelNet (10k)	KD-PCL[18]	CPU	1.86	1×	1e-6
	Brute	CPU	6.25	0.3x	1.97e-8
	Open3D[20]	CPU	**0.12**	**15x**	2.3e-8
	PQT[21]	CPU	1.29	1.4x	8.44e-4
	FRICP[3]	CPU	0.23	8x	2.08e-6
	KD-tree[19]	GPU	1.43	1.3x	9.9e-5
	Brute	GPU	2.36	0.8x	5.08e-7
	VAN-ICP[1]	GPU	0.7	2.7x	1.94e-5
	Ours	FPGA	**0.12**	**15x**	1.38e-3
Bunny (40k)	KD-PCL[18]	CPU	7.2	1×	8e-3
	Brute	CPU	189	0.04x	1.33e-5
	KD-tree[19]	GPU	2.01	3.6x	1.28e-4
	Brute	GPU	10.1	0.7x	8e-3
	VAN-ICP[1]	GPU	0.9	8x	9e-3
	Ours	FPGA	**0.6**	**12x**	1.47e-2

of the source and target point clouds, and the bottom line shows the alignment achieved by the proposed method. For each case, the number of iterations, the final registration error, and the total runtime on FPGA are reported to quantify both efficiency and accuracy. Except for the *Bunny* model, all datasets consist of between 8K and 15K points. In general, the number of iterations required for convergence is determined by both the geometric complexity of the model and the extent of the initial misalignment. In our FPGA implementation, convergence behavior is further influenced by the limited precision of fixed-point arithmetic. Despite these constraints, the proposed method achieves millisecond-level registration even for models containing up to 40K points, while maintaining final alignment errors within $1e{-}2$. These results highlight both the efficiency and the practical applicability of the FPGA-accelerated VAN-ICP framework for real-time 3D registration.

The comparison with existing FPGA-based implementations is summarized in Table II. Although our design operates at a relatively modest frequency of 100MHz, this choice reflects a deliberate trade-off to support large-scale datasets. In

TABLE II
COMPARISON WITH FPGA-BASED IMPLEMENTATIONS

Works	[8]	[7]	[9]	[10]	Our Work
Clock (MHz)	200	250	300	200	100
LUTs[1]	39908	N/A	65843	87976	86311
FFs[1]	54112	N/A	11710	145458	38594
BRAMs[1]	114.5	N/A	86	134.5	86
DSPs[1]	131	N/A	96	552	130
Power(w)	N/A	4.2	4.54	N/A	0.84
Dataset points	32k	30k	30k	30k	40k
NNS Latency(ms)	2.3	N/A	0.64	N/A	8.3
ICP - IL(ms)	23.4	N/A	N/A	N/A	13.2
Total Latency(ms)	N/A	720	N/A	1550	597

[1] Utilization: LUT: 97.51%, FF: 22.18%, BRAM: 17.97%, DSP: 17.86%.

terms of resource utilization, our method demonstrates competitive performance, with LUT and DSP usage comparable to prior works, while achieving the lowest reported power consumption (0.84W). More importantly, our architecture can scale to datasets of up to 40k points, substantially extending the processing capacity for complex inputs. Despite the larger dataset size, the ICP iteration latency remains only 13.2ms, which is significantly faster than the 23.4ms reported in [8], thereby demonstrating the efficiency of our design for registration tasks. While [9] focuses on accelerating the nearest neighbor search, it does not address the entire ICP pipeline. In contrast, our complete implementation reduces the total latency to 597ms, which is less than half of the 1550ms reported in [10] and substantially lower than the 720ms observed in [7], thereby demonstrating the significant acceleration benefits of our design.

VI. CONCLUSION

In this work, we presented an FPGA-based memory-efficient accelerator for point cloud registration. By leveraging a pre-traversal technique to dynamically allocate memory according to the distribution of points within each voxel block, our approach effectively reduces memory overhead and, importantly, enables the system to flexibly process point clouds of varying scales without requiring reconfiguration. This adaptability ensures robust operation across datasets that differ significantly in size and density. Furthermore, the incorporation of a voxel dilation strategy transforms global nearest-neighbor searches into localized operations, thereby lowering computational cost while preserving registration accuracy. Once the search space is localized, the algorithm becomes not only more efficient but also highly compatible with resource-constrained and mobile platforms, where limited memory and compute budgets demand compact yet scalable solutions. Together, these features highlight the potential of the proposed design as a versatile and practical accelerator for real-time 3D perception in embedded and mobile applications.

Future work will focus on further improving the operating frequency and accuracy of the proposed accelerator, as well as investigating its adaptability and performance across diverse mobile platforms. These efforts will advance the practicality and scalability of FPGA-based solutions, paving the way for efficient real-time 3D reconstruction in resource-constrained environments.

ACKNOWLEDGMENT

This work was partly supported by SCAT, Grant No. KJ2550022n, and Telecommunications Advancement Foundation, Grant No. KJ25030011.

REFERENCES

[1] Wang, Weimin, and Qiong Chang."VAN-ICP: GPU-accelerated approximate nearest neighbor search for ICP registration via voxel dilation." ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2023.

[2] Chang, Qiong, Weimin Wang, and Jun Miyazaki. "Accelerating Nearest Neighbor Search in 3D Point Cloud Registration on GPUs." ACM Transactions on Architecture and Code Optimization 22.1 (2025): 1-24.

[3] Zhang, Juyong, Yuxin Yao, and Bailin Deng. "Fast and robust iterative closest point." IEEE Transactions on Pattern Analysis and Machine Intelligence 44.7 (2021): 3450-3466.

[4] Li, Shihua, et al. "Tree point clouds registration using an improved ICP algorithm based on kd-tree." 2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS). IEEE, 2016.

[5] Koide, Kenji, et al. "Voxelized GICP for fast and accurate 3D point cloud registration." 2021 IEEE international conference on robotics and automation (ICRA). IEEE, 2021.

[6] Koide, Kenji, et al. "Globally consistent 3D LiDAR mapping with GPU-accelerated GICP matching cost factors." IEEE Robotics and Automation Letters 6.4 (2021): 8591-8598.

[7] Kosuge, Atsutake, et al. "An SoC-FPGA-based iterative-closest-point accelerator enabling faster picking robots." IEEE Transactions on Industrial Electronics 68.4 (2020): 3567-3576.

[8] Li, Yiming, Kailei Zheng, and Hao Xiao. "A knn accelerator based on approximate kd tree for icp." 2022 International Conference on Image Processing and Media Computing (ICIPMC). IEEE, 2022.

[9] Wang, Chengliang, et al. "An FPGA-based kNN Seach Accelerator for point cloud registration." 2024 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2024.

[10] Feng-Yuan, Shi, et al. "Fast point cloud registration algorithm using parallel computing strategy." Journal of Physics: Conference Series. Vol. 2235. No. 1. IOP Publishing, 2022.

[11] Wang, Fang, and Zijian Zhao. "A survey of iterative closest point algorithm." 2017 Chinese Automation Congress (CAC). IEEE, 2017.

[12] Raettig, Ryan M., et al. "Accelerated point set registration method." The Journal of Defense Modeling and Simulation 21.4 (2024): 421-440.

[13] Zeng, Chao, et al. "A Structure-Based Iterative Closest Point Using Anderson Acceleration for Point Clouds with Low Overlap." Sensors 23.4 (2023): 2049.

[14] Hexsel, Bruno, Heethesh Vhavle, and Yi Chen. "DICP: Doppler iterative closest point algorithm." arXiv preprint arXiv:2201.11944 (2022).

[15] Shiri, Aidin, and Ghader Karimian Khosroshahi. "An FPGA implementation of singular value decomposition." 2019 27th Iranian Conference on Electrical Engineering (ICEE). IEEE, 2019.

[16] Turk, Greg, and Marc Levoy. "Zippered polygon meshes from range images." Proceedings of the 21st annual conference on Computer graphics and interactive techniques. 1994.

[17] Wu, Zhirong, et al. "3d shapenets: A deep representation for volumetric shapes." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[18] Rusu, Radu Bogdan, and Steve Cousins. "3d is here: Point cloud library (pcl)." 2011 IEEE international conference on robotics and automation. IEEE, 2011.

[19] C. Sharma. (2019) Project4-cuda-icp. [Online]. Available: https://github.com/chhavisharma/Project4-CUDA-ICP

[20] Zhou, Qian-Yi, Jaesik Park, and Vladlen Koltun. "Open3D: A modern library for 3D data processing." arXiv preprint arXiv:1801.09847 (2018).

[21] Wieschollek, Patrick, et al. "Efficient large-scale approximate nearest neighbor search on the gpu." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.