2017

Evaluation of a SoC for Real-time 3D SLAM

Benjamin Williams
Iowa State University

Evaluation of a SoC for Real-time 3D SLAM

by

Benjamin Wendell Williams

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Iowa State University

Ames, Iowa

2017

DEDICATION

I would like to dedicate this thesis to my parents, Paul and Gail, for their untiring support of my academics, passion for music, and professional career throughout my lifetime, and especially these last few months.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to Dr. Zambreno for his patience, motivation, and support through the selection, research, and writing process.

ABSTRACT

SLAM, or Simultaneous Localization and Mapping, is the combined problem of constructing a map of an agent's environment while localizing, or tracking that same agent's pose in tandem. It is among the most challenging and fundamental tasks in computer vision, with applications ranging from augmented reality to robotic navigation. With the increasing capability and ubiquity of mobile computers such as cell phones, portable 3D SLAM systems are becoming feasible for widespread use. The Microsoft Hololens, Google Project Tango, and other 3D aware devices are modern day examples of the potential of SLAM and the challenges it has yet to face. The ICP, or Iterative Closest Point Algorithm, is a popular solution for retrieving the relative transformation between two scans of the same object. It has gained a resurgence in popularity due to the rise of affordable depth sensors such as the Kinect in robotics and augmented reality research. ICP, while providing a high certainty of correctness given similar point clouds, is challenging to implement in real time due to its computational complexity. In this thesis, a basic 3D SLAM algorithm is implemented and evaluated, and two proposed FPGA architectures to accelerate the Nearest Neighbor component of ICP for use in a mobile ARM-based System-on-Chip (SoC) are presented. These architectures are predicted to achieve speedups of up to 7.89x and 17.22x over a naïve embedded software implementation.

CHAPTER 1. INTRODUCTION

Simultaneous Localization and Mapping (SLAM), is the task of creating the map of an unknown environment while simultaneously estimating an agent's position and pose within that environment. SLAM is still an unsolved problem, with a wide array of applications such as autonomous navigation, augmented reality, and robotics. With a capable and affordable mobile SLAM system, digital and physical worlds can be integrated in exciting new ways for gaming, productivity, and communication.

SLAM uses a diverse set of approaches that are presented in the academic literature. Global maps can choose to store the environment as 2D [9] or 3D [6] information, and with varying degrees of system memory requirements and granularity. Data can be received using various methods, from a single 2D laser scanner [9], to RGB cameras [8], to modern day usage of RGB-Depth cameras such as the Kinect and Intel RealSense [10]. Alignment techniques for localization vary depending on the computational constraints of the system researched. These range from relatively low computational cost techniques such as sparse landmark extraction and association [10] all the way to dense all-to-all comparisons [28], generally a CPU intensive task.

In this thesis, a computationally expensive Depth Frame SLAM technique, normally constrained to high end desktop computers, is brought to an embedded System-on-Chip (SoC) with an ARM CPU and FPGA coprocessor. A scan matching SLAM algorithm using the Iterative Closest Point (ICP) [3] algorithm is implemented and evaluated on a desktop, and cross compiled for the Xilinx ZedBoard platform. The embedded application is profiled, and two different hardware coprocessor architectures for accelerating the computationally expensive Nearest Neighbor Search algorithm employed in ICP are designed, synthesized, and evaluated using simulation.

The main contributions of this project are as follows:

- A novel SLAM algorithm using ICP to align depth frames from the Kinect v2 sensor
- A baseline hardware coprocessor architecture accelerating the Nearest Neighbor Search step of ICP
- A highly parallel coprocessor architecture, using 110 Nearest Neighbor Processing Elements, aiming to achieve maximum system throughput

The SLAM algorithm is implemented in C++ and uses the OpenCV library for image processing. The evaluations show that it can map the 3D environments in the recorded SLAM scenarios given slow rotational and translational motions. The baseline architecture is calculated to achieve theoretical speedups of up to 7.89x over a pure software implementation. The parallel architecture has estimated speedups of 17.22x over a pure software implementation, a 2.19x speedup over the baseline architecture, and successfully meets the real-time requirements of 2 depth frames processed per second.

The remainder of this thesis is structed as follows. Chapter 2 discusses SLAM in detail and gives an overview of visual SLAM systems as well as hardware acceleration techniques. Chapter 3 describes landmark SLAM systems developed in the last 15 years, and provides an overview of embedded SLAM systems presented in the academic literature. Chapter 4 introduces ICP-SLAM. Chapter 5 discusses the baseline and parallel architectures for acceleration of ICP-SLAM. Chapter 6 presents an analysis of the accuracy and performance of the ICP-SLAM algorithm using the CoRBS dataset. Chapter 7 concludes the project and discusses future work for improving ICP-SLAM.

CHAPTER 2. BACKGROUND

2.1 Simultaneous Localization and Mapping

SLAM, short for Simultaneous Localization and Mapping, is the high-level problem of automated map generation without any ground truth knowledge of pose. Localization is the task of estimating an agent's pose using discrete sensor readings. While the map is constructed, the agent must correctly localize itself in the map that it has constructed up to time t, and expand the map with new sensor information at time $t + 1$. An abstracted example of visual SLAM is depicted in 2.1. The relationship between map construction and localization is essential in SLAM. If the localization is faulty, new data added to any existing map will be inconsistent, therefore not resembling the environment. In addition, if the map is not properly constructed, new sensor information will fail to properly localize the agent. All pose estimations by the SLAM agent will become subject to error. This can lead to problems such as a "death spiral", where erroneous map measurements lead to further errors, causing the entire map to become unusable.

Solving the SLAM problem requires a diverse set of approaches with respect to the type of sensor data available, timing requirements, and computational resources available. Many SLAM research projects are focused on batch processing of a sequence of frames. These are often focused on validating the accuracy of a new algorithm. Others focus on real-time performance and attempt to be feasible for use with commodity sensors and computational hardware. The dimensionality of the mapping technique (2D vs 3D) also has a great impact on the overall approach.

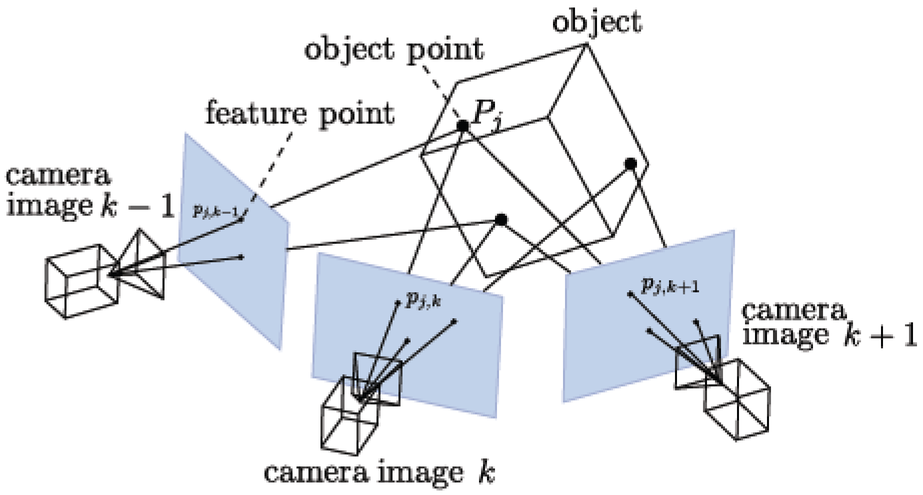


Figure 2.1 SLAM of a fixed object.

[22]

2.1.1 Sensors

SLAM approaches are often grouped by the type of sensor, and contrasted by the differences used to make sense of similar sensory input. For example, a robot often needs a 2D map of an indoor environment to successfully navigate. These applications are becoming increasingly feasible with the rise of popular LiDAR and high FoV range sensors. These sensors are commonly used in tandem with IMU and wheel encoder odometry for accurate sensor fusion. In [29], a Kalman Filter is used to create a probabilistic map of an office space.

Monocular or stereo view color cameras have been used as sensory input for dozens of SLAM systems. Using a combination of visual features, distortion correction, and semi-dense reconstruction techniques, single camera SLAM has been proven to be an effective medium for camera tracking. The first true visual SLAM system, MonoSLAM [8], used a commodity web camera and PC hardware to construct and track a sparse feature map. PTAM [14], another key innovation, introduced the concept of parallel map refinement alongside real time monocular tracking.

Stereo Vision brings estimated depth information into the equation, with both color imagery and a dense distance frame available for environmental perception. Stereo vision SLAM has captured the interest of researchers in recent years due to the similarity between that approach and how we as humans perceive our surroundings. In [25], a system for large scale position tracking over the course of kilometers of travel was proposed using stereo cameras as input. In 2011, [26] presented a dense structure model for use with input from a stereo camera sensor, and demonstrated the used of disparity maps to localize and track true structure through the presence of noise. S-PTAM is a modification of the monocular PTAM algorithm with data from stereo vision [31].

Kinect is a 3D depth and RGB sensor developed by Microsoft, intended as a peripheral for the Xbox video game consoles and Windows PC's [5]. It tracks the skeletons of the players and allows them to control game avatars using physical motions without the presence of a traditional controller. Although the Kinect was intended solely for use in an entertainment context, it has gained popularity in the robotics and computer vision communities.

The Kinect was one of the first commercial products to provide researchers with a reliable, inexpensive 3D depth sensor. It allowed for dense depth information to be combined with an RGB camera for new sensor fusion fueled scenarios. The Kinect uses a wide-angle Time of Flight sensor to collect dense 3D information. In the later iteration of the sensor, the Kinect v2, the depth resolution and precision were improved. The resolution of the RGB camera was increased from 480p to 1080p. Low light performance and the number of skeleton models that can be tracked simultaneously were also increased. The Kinect v2 is the sensor used to obtain the datasets in this thesis.

2.1.2 Iterative Closest Point

Iterative Closest Point (ICP) is an algorithm used to transform a point cloud to align itself with a reference shape. It can be adapted to align 3D or 2D shapes, in a variety of data formats. In this thesis, ICP is used for the purposes of localization in the context of the SLAM problem. It is a common technique employed to solve the problem of scan matching. Over the years, several variants of ICP have been presented to improve certain performance characteristics. Adaptations using different data structures and point dimensionality have been demonstrated as well [18].

ICP finds the relative transformation between point clouds by iteratively associating points and recovering the transformation that minimizes a user defined distance-cost metric. The methods for selecting point associations and cost metrics vary with different ICP implementations, but all follow the same high-level algorithm.

The ICP algorithm is defined by the following steps [3]:

1. Define the minimum error e and the maximum number of iterations i that ICP will use for refining the transformation;
2. Given two point clouds, *data* and *source*, center each around a common origin;
3. For each point p in *data*, find the nearest neighbor point q in *source*;
4. Using these correspondences, construct a covariance matrix;
5. Perform SVD (Singular Value Decomposition) on the covariance matrix;
6. Compute the rotational matrix R from the results of SVD;
7. Rotate *source* by R (or *data* by the inverse of R);
8. (Optional) Calculate an offset between all point associations, and translate *data* accordingly;
9. Calculate the error between *data* and *source*;
10. If the computed error is larger than e and the number of iterations is less than i, repeat steps 2-9.

2.1.2.1 Distance Metrics

ICP implementations have used several metrics for determining the correspondences between two point clouds. These methods are combined with a cost function to quantify the similarity between these clouds. The original metric is the Mean Squared Error of Point-to-Point distance [3]. The distance between points a and b in three dimensions is calculated using the Euclidean distance equation:

$$d(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}. \tag{2.1}$$

.

The mean squared error, or MSE, of an estimation measures the average of the squared errors found in measurements. Mean squared error is a function of risk corresponding to the expected value of the squared error loss. With n samples, set of predictions P, and recorded measurements R, MSE is computed with the equation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(P_i - R_i)^2. \tag{2.2}$$

Point-to-Point is a metric with moderate resilience to noise, but fails given a large enough difference between point sets. It assumes that the scanned model is a discrete set of points, with no filled space between or behind. It also fails to take advantage of any additional information given about the points such as surface normals or color.

If point normal data is available, a point-to-plane distance metric can be calculated. Given a map containing surface information, reverse-projecting the scan point normals onto a stored surface has been shown to improve alignment [36]. This is also known as "ray shooting."

In an indoor environment, usually containing an abundance of planar surfaces, point-to-plane error metrics have been shown to greatly increase convergence rates and reduce noise. This method is highly robust to translations, and works well for small to medium rotations. When transformations have altered the scanned point set to the point that surface normals no longer face the correct plane, the quality of associations quickly degrades.

In an alternate ray shooting scheme known as"reverse calibration", the direction of the ray is calculated by projecting a line from the center of the camera origin through a scanned point [36]. This corrects for situations where detected surface normals are noisy, inaccurate, or large translations between surfaces occurs. A hybrid ray shooting approach was presented by Blais and Levine in 1995 [4], where a point was projected onto a surface and a radius-bound point-to-point distance metric was used to search for the nearest neighbor on the target surface. Point-to-ray distance, and compatibility of intensity were also evaluated. Beyond a simple Euclidean distance metric, the relative angle between surface normals was presented as a robust method to find point similarities [18].

2.1.2.2 Nearest Neighbor Search

To find the nearest neighbor of a point, otherwise known as a Nearest Neighbor Search, several methods have been presented. The most basic approach is to iterate over every point in the target point cloud, compute the distance-cost between the scanned and source points, and compare this value to the stored minimum [3]. This method guarantees that the nearest point is found, but has an undesirable runtime of $O(N * M)$, where N is the number of points in the stored point cloud, and M is the number of points in the latest scan. This is referred to as a brute force search, specified in Algorithm 1. A visualization of the nearest neighbor associations step can be seen in Figure 2.2.

```
Input: Scan Point p
Output: The Nearest Neighbor
min distance = MAX DISTANCE;
nearest neighbor = null;
foreach Point q in Map do
    distance = euclidian_distance(p, q);
    if distance ¡ min distance then
        min distance = distance;
        nearest neighbor = q;
    end
end
return nearest neighbor
```

Algorithm 1: The brute force nearest neighbor search

Although brute force nearest neighbor has a poor worst case runtime, it can easily be parallelized due to the independence of each scan point's search. The runtime is also predictable and repeatable given a certain size of the *source* and *data* point clouds. Structuring the point data as arrays allows for consistent size and shape of information, as opposed to other approaches using advanced data structures. These properties make it desirable and straightforward for parallel implementations [18].

A k-depth tree is a binary tree data structure which organizes points using subdividing hyperplanes. k-depth trees allow for $O(log(N))$ multidimensional searches such as the Nearest

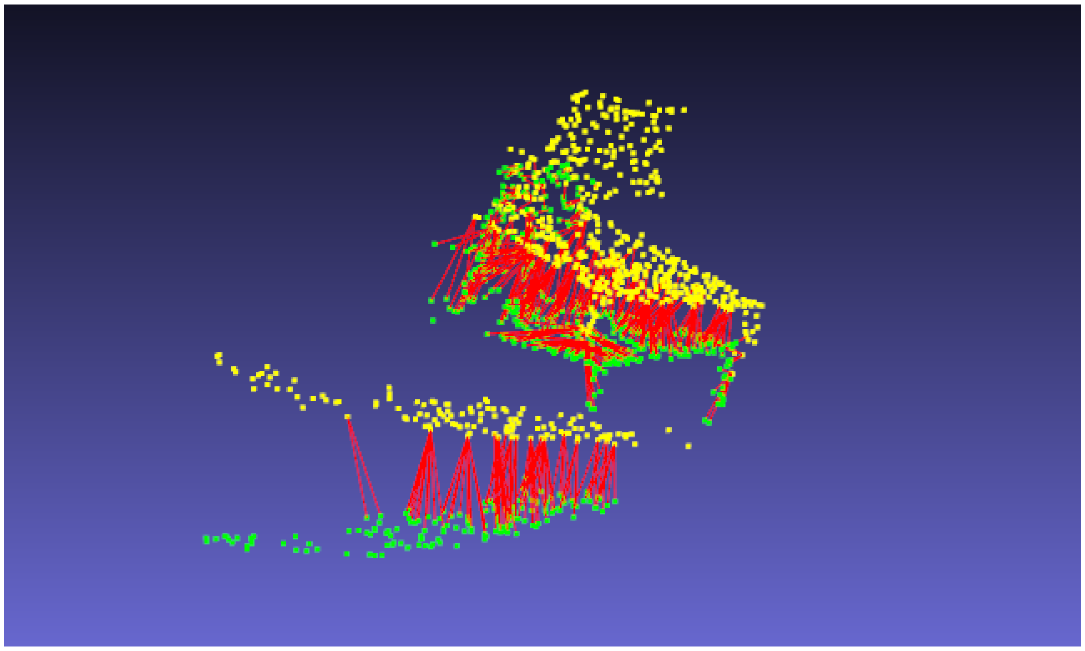


Figure 2.2 Visualization of the Nearest Neighbors Associations step.

Neighbor search present in ICP [39]. They are popular for use in ICP due to their speed. Although k-depth tree searches are considered faster than the brute force alternative, tree imbalance as well as poor hyperplane subdivision can cause reductions in performance. This data structure performs best when points are evenly distributed across all k dimensions.

Although the methods mentioned above are the two most basic and well known nearest neighbor searches, there are hundreds of papers written on the topic. k-depth trees are effective at lower dimensions, but as k increases, runtime savings decrease. Arya et al. proposed a $k-d$ tree search with radius bound approximations to accelerate runtime [1]. This approach considers n approximate nearest neighbors, points whose distance d between the searched point p will never exceed a predefined margin of error. A priority queue was used to reduce search latency. This error-bound method of approximating the nearest neighbor will guarantee that a nearest neighbor will be selected within a certain proximity of the true nearest neighbor, and was demonstrated to improve runtime by orders of magnitude in certain situations. In [35], a graph of nearest neighbors is presented in which vertices represent points and edges connect points to respective nearest neighbors. The authors select evenly distributed vertices in the graph as "seeds" and use a greedy best-first algorithm to quickly find the nearest neighbor.

2.1.2.3 Mapping

Many different mapping techniques have been presented for use in SLAM applications, each with their own strengths and weaknesses. Grid maps [27] are human readable and logical in terms of how we think about the world, but take up large amounts of memory. Because empty space is filled with redundant data, total mapping volume can be limited. Some attempts to solve this problem have been presented, such as voxel hashing [20] and octrees [24]. Voxel hashing compresses spatial information into a lookup table, where the only information stored is filled space. It combines dense volumetric information with constant time lookups and the ability to stream data from a voxel lookup table to other 3D representations.

An Octree is a tree data structure which subdivides space into 8 octants. In each octant, a single three-dimensional point is stored as the center for the recursive subtree. This center point now defines the center of a smaller octree that occupies the volume of its parent octant. Octrees greatly reduce the size of map when a small portion has been populated, without any loss in precision. Octrees are different than k-depth trees, as octrees split the space into 8 sections around a single center point while k-depth trees split it along a dimension.

2.2 Hardware Acceleration

Hardware Acceleration is the act of moving a functional component of a software system from a general-purpose processor to a dedicated piece of hardware. Hardware acceleration is used for performance critical tasks such as image processing, graphics, audio, and packet routing in internet routers. CPU's traditionally support a sequential path of execution for machine instructions, and therefore are limited by the size of the data that can be operated on in a given time frame. Hardware accelerators can greatly increase the size of the dataset operated on by improving concurrency [7], storing information in close locality to processing elements [15], and employing fixed length datapaths. Although modern processors have increased throughput through use of multiple cores and dedicated SIMD units [12], hardware acceleration is still relevant to performance critical tasks that use large sets of information.

Creating dedicated hardware is an expensive and time consuming process. Reconfigurable computing is a paradigm that combines the flexibility of software defined functionality with the high performance of a dedicated hardware solution. General purpose computing on GPU (GPGPU) [19] and FPGAs fill the need for many resource intensive tasks at a fraction of the cost. While they may never reach the performance of a traditional hardware accelerator, they commonly meet the needs for all but the most rigorous tasks.

A field-programmable gate array (FPGA) is a chip intended to be reconfigured after it has been manufactured. FPGA's employ hundreds or even thousands of basic logic cells, which are configured in a process known as synthesis to form a custom digital circuit. They are programmed using hardware design languages, visual system design tools, or modern high level synthesis that attempts to define a digital circuit from imperative programming paradigms.

A major paradigm in hardware acceleration is to compute independent values in parallel with separate processing elements. As FPGA's can be configured to stream data through multiple paths, large performance gains can be realized using multiple independent channels of execution. FPGA's can implement fixed-point and integer arithmetic using abundant lookup tables, but floating-point operations require scarce digital signal processor resources. FPGA's have limited I/O channels to communicate with external memory. If a design requires random access to off-chip memory, performance gains could be eliminated by memory port starvation.

Pipelining is another method to increase computational throughput [15]. When a calculation is pipelined, intermediary values along the path of execution are stored to increase the total computing throughput. It is akin to dynamic programming in that excess cycles computing the same value are eliminated, in exchange for the added cost of caching data at each stage of the pipeline.

A systolic array is a homogenous network of processing elements, or PE's [17]. Each processing element independently computes a subset of a larger operation on data received from upstream nodes. Systolic arrays use pipelining and temporal locality of data being operated on to increase throughput. Systolic arrays have been used for image processing [11], [21] priority queue algorithms, matrix multiplication [16], and many other applications. A systolic array architecture was chosen for the high throughput design presented in this thesis.

CHAPTER 3. RELATED RESEARCH

3.1 Visual SLAM

In 1992, Paul Besl [3] published the first paper to dramatically change the way we think about point clouds and their applications. This paper introduced ICP, or the Iterative Closest Point algorithm. It applied 3D point cloud registration to the task of recovering a 6DoF transformation between a source shape and an incomplete data shape. This algorithm has revolutionized scan and point cloud alignment techniques for a variety of tasks related to object tracking [2] and SLAM [45].

In 2003, Andrew Davison published one of the first groundbreaking papers in visual SLAM history, Real-Time Simultaneous Localisation and Mapping with a Single Camera, otherwise known as MonoSLAM [8]. It was the first successful application of robotic SLAM techniques to cross the gap to a camera only computer vision domain. It created a live map of sparse, persistent landmarks with probablistic motion estimation. A dynamic approach to mapping allowed for the algorithm to track motion with some resilience to misaligned or lost features. MonoSLAM achieved real time performance using a desktop CPU.

In 2007, Klein and Murry created the next big leap in mobile SLAM by splitting the tasks of camera tracking and map refinement into parallel threads [14]. One thread deals exclusively with tracking camera features, while the other produces a joint 3D map of features captured in previous frames. This allowed for map optimization techniques normally relegated to offline structure from motion approaches, and that greatly increased map consistency and precision. Far more landmarks can be present in the map than in previous, single threaded techniques. PTAM was applied to an augmented reality use case, where a virtual workspace was rendered on a flat plane over the tracked area.

In 2008, Belshaw and Greenspan presented the paper that was one of the biggest inspirations for this work, A High Speed Iterative Closest Point Tracker on an FPGA Platform. It took the computationally intensive problem of scan alignment and applied hardware acceleration techniques with a high-end FPGA to track a single 3D object at 200 FPS [2]. This project pioneered the concept of a nearest neighbor unit, or a single PE present in hardware that can quickly find the nearest neighbor to a single point. Using 16 of these nearest neighbor units in parallel, each of which can compute the distance between a point pair in a single clock cycle, the 200 FPS performance requirements were met.

Several important distinctions between ICP Tracker and the system presented in this thesis must be made clear. This tracker used heavy subsampling and ROI masking to cut down the number of points processed to around 200, and was evaluated using small 3D models, usually containing fewer than 1000 points. The size of both the scan and model are roughly an order of magnitude smaller than the data used for this thesis, and assumptions made for single object tracking do not hold for dense scene reconstruction.

One of the first projects to utilize the Microsoft Kinect sensor, RGB-D SLAM [26], is a dense 3D mapping system that uses an advanced joint optimization algorithm employing both shape-based alignment and visual features. It used a form of key frame based loop closure detection, which allowed for globally consistent mapping and improved pose optimization. The algorithm was evaluated using data from two large indoor environments. It was shown to create accurate, consistent maps, but was too slow to run in real time.

Kinect Fusion was a revolutionary dense SLAM system presented by [28]. It used all available depth information from the Kinect sensor to fuse depth data into a model of surfaces in real time. The pose of the Kinect is tracked using a version of the Iterative Closest Point algorithm, matching scan data against the global geometric model. This growing global model was proven to be a far more stable anchor for depth frame alignment than previous frame-to-frame approaches [8]. Using GPGPU, localization used all available scan points to track the camera in real time. Limited drift and high accuracy were recorded with room sized scenes. It was the most viable real time dense reconstruction system to be presented at the time, and it is still relevant for robotics and augmented reality to this day.

3.2 SLAM on a SoC

As mobile computing capabilities increase, limited attempts at implementing visual SLAM on a system-on-chip have been attempted in recent years. In 2014, Rodriguez-Araujo et. al presented a distributed image processing system used to localize unmanned ground vehicles in indoor spaces [33]. A network of connected FPGA processing nodes perform image preprocessing, reducing the amount of data transmitted through wireless channels. Dynamic responses to guidance and control problems allowed for an inexpensive and resilient system.

In [30], a visual-inertial sensor unit aimed at Micro Aerial Vehicles providing robust real-time SLAM capabilities is presented. Up to four cameras and an Inertial Measurement Unit (IMU) are connected to an SoC containing an ARM CPU and FPGA. This system allows for a novel fusion of visual and inertial sensor data. A high degree of robustness and accuracy is reported. The FPGA preprocesses visual keypoints to reduce the SLAM computational overhead and lowers the barrier to entry on computationally limited platforms. Hardware design, evaluation, selection of sensors, and firmware are covered in this work.

CHAPTER 4. ICP-SLAM

This section presents a software-only SLAM algorithm that uses a voxel based certainty map and ICP frame to global alignment. Depth data from the Kinect v2 is preprocessed and aligned with a growing 3D global model. The global map is updated with each new scan.

4.1 Tools

Several open source tools commonly used for computer vision were used in this project for image processing, visualizations, and build support. OpenCV [42], short for Open Source Computer Vision Library, is an open source library that contains implementations of many standard computer vision algorithms. It has universal support across the open source community as well as industry support. It was selected for this project for image processing, the 3D Viz library, and cross platform support. The C++ bindings for OpenCV 3.2.0 were used.

The Visualization Toolkit (VTK) [37] is an open source library for visualizing 3D information, image processing, and computer graphics. VTK is a comprehensive package that can visualize information from point clouds, textured 3D models, and volumetric rendering to vector graphics. It supports integrations with OpenCV, several open source GUI toolkits such as Qt and Tk, and the Microsoft Windows windowing system. The Viz bindings to call VTK functions from OpenCV were used in this project.

CMake is another open-source, cross-platform toolset used to build this project across both Linux and Windows based platforms. CMake simplifies the compilation process using simple configuration files, and can automate the generation of native makefiles and workspaces.

Microsoft Visual Studio is an integrated development environment (IDE) used in this project for its ease of build configuration, run configuration, and Intellisense syntax completion. The bundled Visual Studio performance profiler was indispensable for collecting high level performance information.

4.2 Algorithm

The high-level algorithm for ICP-SLAM is shown in Figure 4.1. In the Initialization step, the map is seeded with data from the very first frame in the dataset. Camera position and rotation are initialized to world origin and the identity rotation matrix, respectively. Then, a depth frame is loaded from the filesystem and preprocessed using OpenCV. A point cloud data structure is constructed using the preprocessed frame, and then ICP returns a relative transform between the global map and the newest scan. Finally, the estimated transformation is applied and new points are added to the map. This process is repeated until all depth frames have been successfully processed.

4.2.1 Preprocessing

When an image is first read from the Kinect sensor, noise and distortion are present and must be accounted for. Before depth information is suitable for scan matching, the following three-step process is implemented to preprocess the depth frame for transform recovery:

1. Undistortion, or correction for the pinhole camera model;
2. Distance filtering;
3. Denoising.

The preprocessing pipeline is depicted in Figure 4.2.

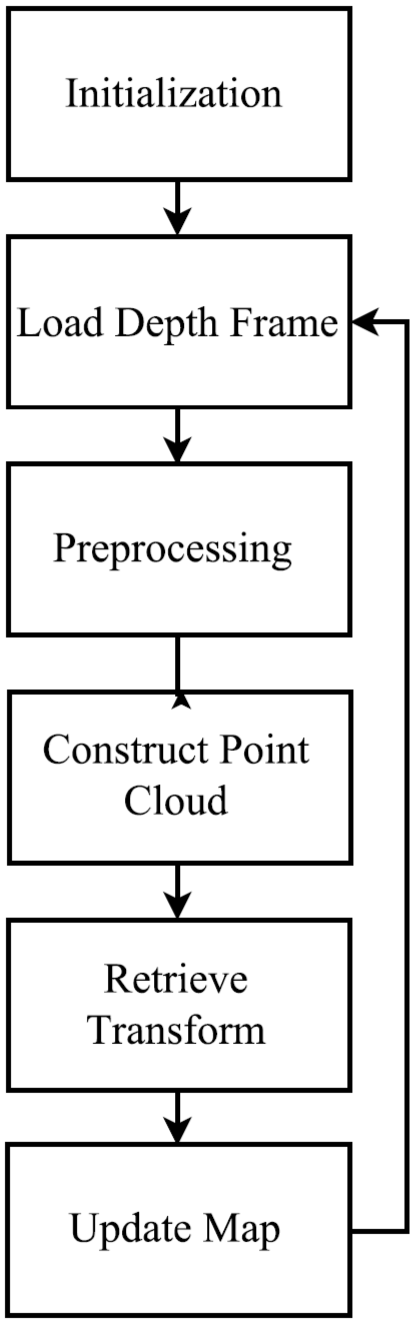


Figure 4.1 High Level Algorithm.

4.2.1.1 Undistortion

The Kinect depth sensor returns depth images consistent with a pinhole camera model. Pinhole cameras, however, distort the images they capture. Thus, the algorithm must account for all distortion that would hinder correct depth alignment. The pinhole mathematical model relates a point in 3D space with its projection onto a captured image from an ideal pinhole camera. Using OpenCV's undistort function, a combination of distortion correction and remapping of pixels to their "true" image position, the downsides from pinhole cameras can be alleviated.

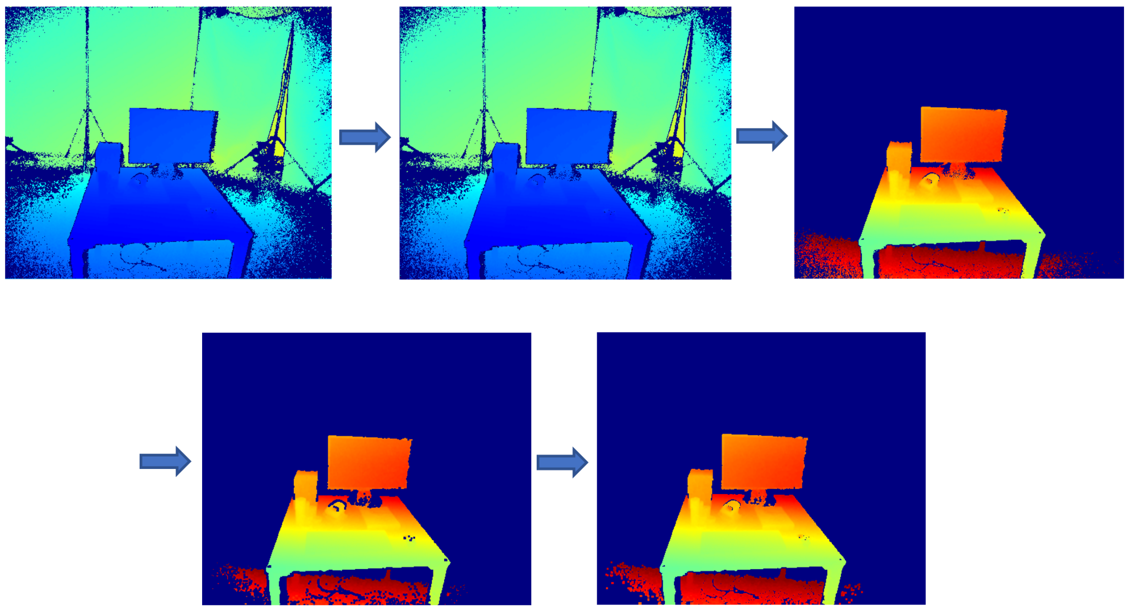


Figure 4.2 The Preprocessing Pipeline. A raw depth frame is loaded (top left), corrected for distortion, filtered by distance, and denoised using morphological operations.

Radial and tangential distortion affect image misshaping in different ways. Radial distortion, otherwise known as "barrel" or "fish-eye" distortion, occurs because of the concaveness of the lens, and can be expressed with the following formula [34]:

$$X_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6), \tag{4.1}$$

$$Y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6). \tag{4.2}$$

For true pixel coordinates (x, y), with given distortion coefficients $k1$, $k2$, $k3$, and r, a pixel's distorted position on an image should be $(X_{corrected}, Y_{corrected})$.

Tangential distortion occurs when the lens is not truly parallel to the image capture sensor. It is also known as "decentering distortion." Tangential distortion can be represented with the following formula:

$$X_{corrected} = x + 2p_1xy + p_2(r^2 + 2x^2), \tag{4.3}$$

$$Y_{corrected} = y + p_2(r^2 + 2y^2) + 2p_2xy. \tag{4.4}$$

The distortion coefficients of a camera can be combined into a single matrix known as a camera matrix. A camera matrix is a 3 by 4 matrix that describes the mapping from physical objects in the world to 2D pixels in an image. A camera matrix F is defined as follows:

$$F = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

. Once the camera intrinsics are known, reversing the distortion becomes a problem of solving for the original (x, y) given $(X_{corrected}, Y_{corrected})$. Given corrected coordinates (u, v) for pixel coordinates (x, y), the reverse mappings M_x and M_y are calculated as follows:

$$M_x = \left(\frac{u - c_x}{f_x}\right) 1 + k_1r^2 + k_2r^4 + k_3r^6) + 2p_1 \left(\frac{u - c_x}{f_x}\right)\left(\frac{v - c_y}{f_y}\right) + p_2 \left(r^2 + 2\left(\frac{u - c_x}{f_x}\right)^2\right), \tag{4.5}$$

$$M_y = \left(\frac{v - c_y}{f_y}\right) (1 + k_1r^2 + k_2r^4 + k_3r^6) + p_1 \left(r^2 + 2\left(\frac{v - c_y}{f_y}\right)^2\right) + 2p_2 \left(\frac{u - c_x}{f_x}\right)\left(\frac{v - c_y}{f_y}\right). \tag{4.6}$$

Although undistortion must happen every frame, given the same camera parameters remapping pixels to an undistorted image becomes a basic map lookup with interpolation. Distortion models are continuous functions that rarely line up with discrete pixel coordinates. This is represented with the following formula:

$$I_{dest}(x, y) = I_{dist}(M_x(x, y), M_y(x, y)) \tag{4.7}$$

, where I_{dest} is the corrected image, I_{dist} is the distorted image, and (x, y) is the image coordinate to solve for. This is not the only way to account for distortion, but it is the method implemented in the OpenCV library used in this project [42].

4.2.1.2 Distance Filtering

Points that are too close or too far from the camera are filtered out of the image. The Kinect v2 is generally very accurate, but precision error increases quadratically with distance from the camera. Therefore, all points with z values greater than Z_{max} cannot be considered reliable for robust scan matching purposes. Inversely, some points are incorrectly labeled as being too close to the Kinect due to noisy edges and reflective surfaces. Therefore, a distance filter that eliminates points within Z_{min} meters of the camera, and points greater than Z_{max} meters from the camera is implemented. A depth frame before and after depth filtering is depicted in Figure 4.3.

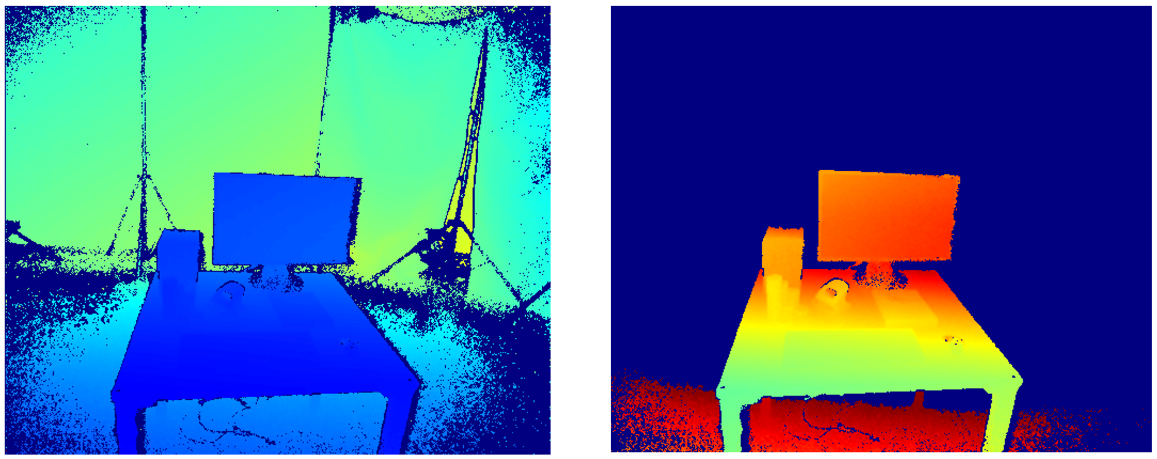


Figure 4.3 Depth Filtered Image

4.2.1.3 Denoising

A basic denoising step using morphological operations was used to smooth out rough, patchy segments of the depth frame. This is common at edges of objects, reflective surfaces, or small complex geometric shapes. First an erosion operation was performed to eliminate these small patches of data, and then a dilation fills in the edges of solid objects. The stages of these denoising steps can be seen in Figure 4.4.

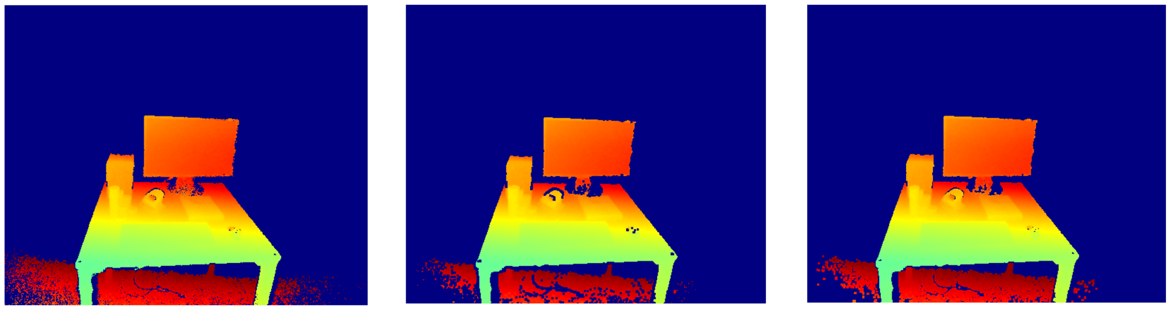


Figure 4.4 Morphological Denoising Operations. The depth filtered image is first eroded and then dilated with a 3x3 rectangular structuring element.

4.2.2 Construct a Point Cloud

Once an image is preprocessed, the 2D depth information needs to be transformed to a 3D representation for use in transform recovery. This is done in a multi-step process that is described below:

1. Subsample the depth image;
2. Project points from screen space to world space;
3. Center all points around the center of mass;
4. Transform the point cloud to the camera's estimated position.

First, random subsampling is applied to the raw Kinect reading. Random subsampling is a method for reducing the computational overhead of ICP presented in [23]. It was shown that cutting down the number of points processed can increase runtime of the Nearest Neighbor calculation by orders of magnitude while keeping alignment accuracy at nearly perfect levels.

Once a point has been selected for inclusion in the cloud, it is projected from screen space to physical space. Given the known Kinect depth camera intrinsics matrix F, with linear focal lengths f_x and f_y, principal point offsets c_x and c_y, and axis skew s, the local (x, y, z) coordinates for a point is calculated using the following formula [13]:

$$F = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$x = \left((x_d - c_x) * \frac{depth(x_d, y_d)}{f_x} \right), \tag{4.8}$$

$$y = \left((y_d - c_y) * \frac{depth(x_d, y_d)}{f_y} \right), \tag{4.9}$$

$$z = depth(x_d, y_d). \tag{4.10}$$

As the Kinect returns depth values in increments of the smallest level of precision (0.2 mm), all z values must be multiplied by 5000 to convert the distance units to meters.

Once all points have been added to the point cloud, each is offset to a center. This center is computed by taking the average x, y, and z, of all points. Centering point clouds is important for calculating relative rotations and transformations.

$$Center_x = \left(\frac{\sum_{i=1}^{n} p_i x}{n} \right), \tag{4.11}$$

$$Center_y = \left(\frac{\sum_{i=1}^{n} p_i y}{n} \right), \tag{4.12}$$

$$Center_z = \left(\frac{\sum_{i=1}^{n} p_i z}{n} \right), \tag{4.13}$$

$$\forall p \in Scan, \quad p_{centered} = p - Center. \tag{4.14}$$

Once the cloud has been fully constructed, it must be transformed from a local coordinate space to the known global coordinate space. The point cloud is rotated by R, the current estimated rotation of the SLAM agent, and then translated by T, the current estimated position. The scan points are aligned into a matrix L, and given a camera rotation R and position p, are transformed into $L_{transformed}$ given the following formula:

$$L = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ & \dots & \\ x_n & y_n & z_n \end{bmatrix}$$

$$L_{rotated} = RL, \tag{4.15}$$

$$L_{transformed} = L_{rotated} + p. \tag{4.16}$$

4.2.3 Retrieve Transform

As described above, the Iterative Closet Point algorithm is used to recover a 6DoF transformation between two 3D shapes. It was selected due to its high accuracy, robust convergence of similar point clouds, and the parallelization potential of the Nearest Neighbor search algorithm. An example of convergence using ICP is displayed in Figure 4.5.

First, a brute force point-to-point Nearest Neighbor search is implemented to find point associations. This all-to-all comparison guarantees that the true nearest neighbor of each point is found, but ensures a worst-case runtime of $O(MN)$, where M is the number of points in the constructed map, and N is the number of points in the newly scanned point cloud.

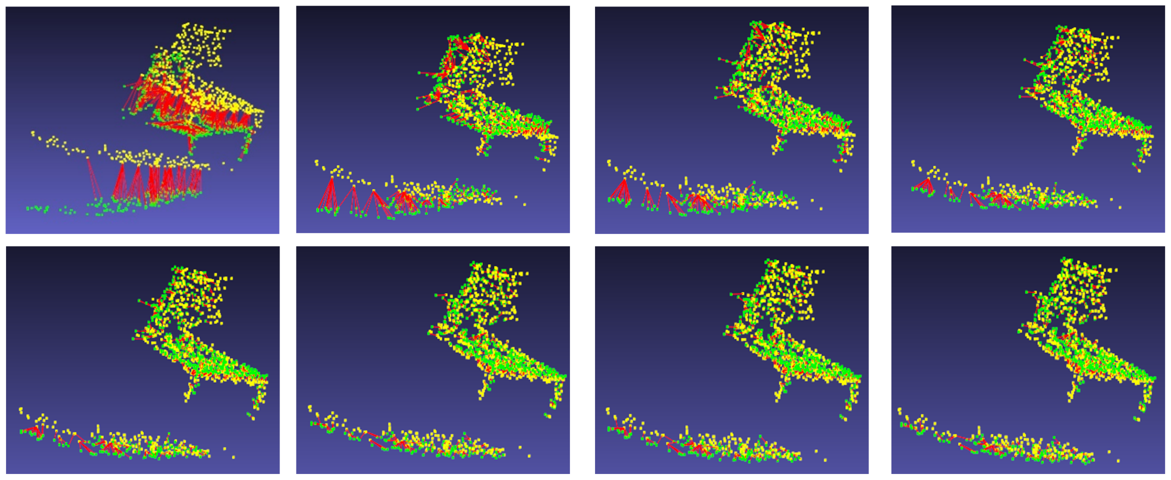


Figure 4.5 Convergence of Point Clouds using ICP. Associations between nearest neighbors are drawn in red.

A modified Euclidean distance formula is presented:

$$d' = (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2. \tag{4.17}$$

This formula does not use the square root function to return the true distance. This is done intentionally to reduce the computational overhead, as a consistent relative cost metric still returns the correct nearest point. If the correct nearest neighbor is returned, the true distance can be calculated at post processing.

Singular value decomposition, or SVD, is the matrix decomposition A into the product of three matrices $A = UDV^T$. U and V have orthonormal columns, and D is diagonal with positive real entries [34].

SVD takes a rectangular matrix of measurement data, referred to here as the $n * p$ matrix A, where each row represents a measurement, and each column represents an experimental condition. The SVD theorem states:

$$A_{n*p} = U_{n*n} S_{n*p} V^T_{p*p}, \quad where \tag{4.18}$$

$$U^T U = I_{n*n},$$

$$V^T V = I_{p*p}.$$

For the purposes of this project, SVD is used to determine the relative rotation between two associated point sets to minimize the distance between them. To format the information in a way that SVD can solve this problem, a covariance matrix must first be constructed. Given a set of 3D point associations E, in which all points p in the latest scan are associated with the nearest point q in the map, matrices M and N can be constructed, and are used to yield the covariance matrix C using the following formula:

$$(p, q) \in E$$

$$M = \begin{bmatrix} q_{1x} & q_{2x} & & q_{n-1x} & q_{nx} \\ q_{1y} & q_{2y} & \dots & q_{n-1y} & q_{ny} \\ q_{1z} & q_{2z} & & q_{n-1z} & q_{nz} \end{bmatrix} \quad N = \begin{bmatrix} p_{1x} & p_{1y} & p_{1z} \\ p_{2x} & p_{2y} & p_{2z} \\ & \dots & \\ p_{n-1x} & p_{n-1y} & p_{n-1z} \\ p_{nx} & p_{ny} & p_{nz} \end{bmatrix}$$

$$C_{3*3} = M_{n*3} N_{3*n}. \tag{4.19}$$

Now that the covariance matrix point alignments have been created, the relative rotation R between the data point cloud and map point cloud can be recovered by applying SVD and using the following formula:

$$C_{3*3} = U_{3*3} S_{3*3} V_{3*3}^T, \tag{4.20}$$

$$R = \left(V^T\right)^T * (U)^T . \tag{4.21}$$

R is given in terms of a rotation that aligns the map points with scan. To apply the rotation to the scan set, the inverse matrix must be calculated and applied to the correctly centered matrix containing the entire scan point cloud. A matrix of points centered around the point cloud origin, L, is constructed and rotated as follows:

$$L = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ & \dots & \\ x_n & y_n & z_n \end{bmatrix}$$

,

$$L_{rotated} = R^{-1} L. \tag{4.22}$$

Now that the rotation has been applied to the scan data, a translation is applied. This is calculated using the distance offset of associated points. It is not applied using the centers of the map and scan point clouds, as non-associated points contain noise and have no bearing on the correct rotation. Given the same set of associations E, in which all points p in the latest scan are associated with the nearest point q in the map:

$$\text{Offset}_x = \left(\frac{\sum_{i=1}^n p_{ix} - q_{ix}}{n}\right), \tag{4.23}$$

$$\text{Offset}_y = \left(\frac{\sum_{i=1}^n p_{iy} - q_{iy}}{n}\right), \tag{4.24}$$

$$\text{Offset}_z = \left(\frac{\sum_{i=1}^n p_{iz} - q_{iz}}{n}\right), \tag{4.25}$$

$$\forall p \in Scan, \quad p_{translated} = p + \text{Offset}. \tag{4.26}$$

The MSE, or Mean Squared Error, of the aligned points is then computed. If this error falls below the predefined threshold e, ICP considers the point clouds to have converged, and subsequently the relative transformation between the two to be correct. If the computed MSE is still higher than the minimum error e, and ICP has not iterated to the maximum number of times specified, Nearest Neighbor alignment and transform recovery are repeated.

Selecting proper values for the minimum acceptable error and maximum number of iterations is a delicate problem. Allowing ICP to iterate too much slows down runtime to a crawl, but iterating too little yields poor transform accuracy. MSE values were recorded to demonstrate the convergence of two scans in a synthetic test bench over the course of 30 ICP iterations in Figure 4.6. Two consecutive Kinect scans were translated by 0.2m and rotated by 3 degrees in random directions to illustrate a dramatic ICP scenario. Convergence is rapid in the first 4 iterations, and becomes sluggish quickly after.

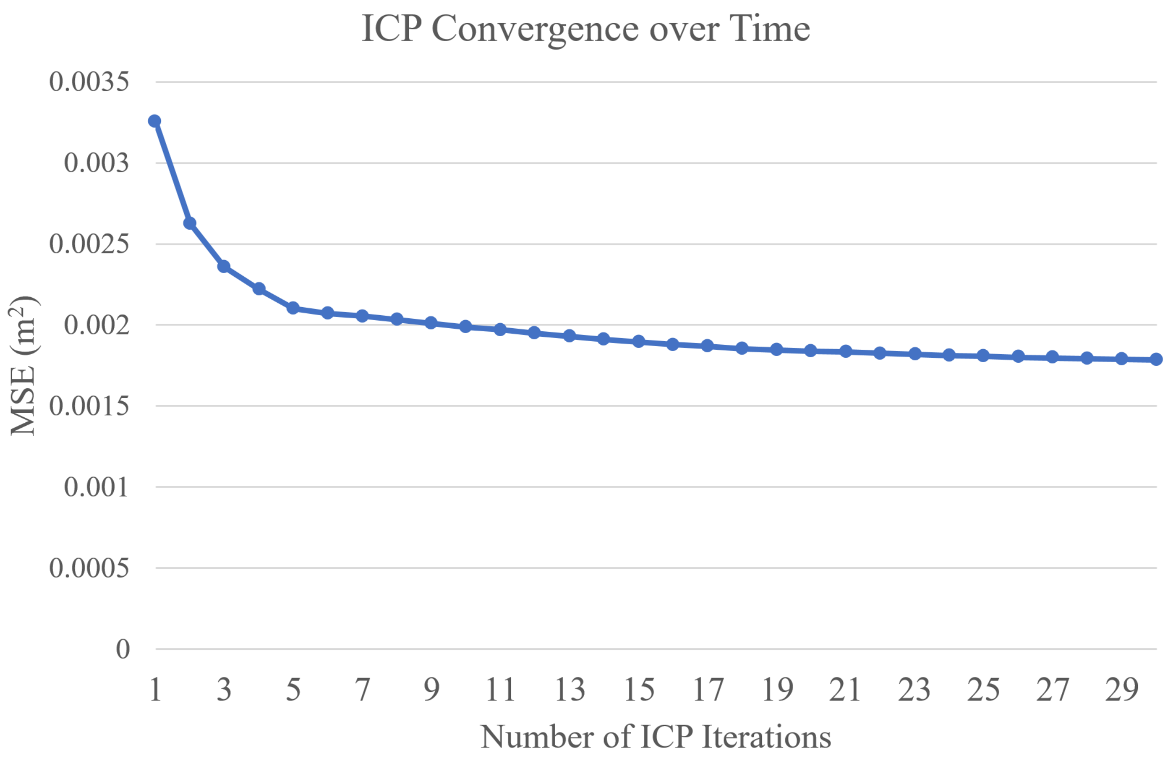


Figure 4.6 ICP MSE over the number of iterations.

Selecting appropriate values for maximum iterations i and maximum error e is a balancing act between acceptable accuracy and speed. An evaluation of runtime vs average alignment MSE was performed over the Desk, Electrical Cabinet, and Race Car datasets, and can be seen in Figure 4.7. A lower MSE indicates that ICP returned a more accurate transformation.

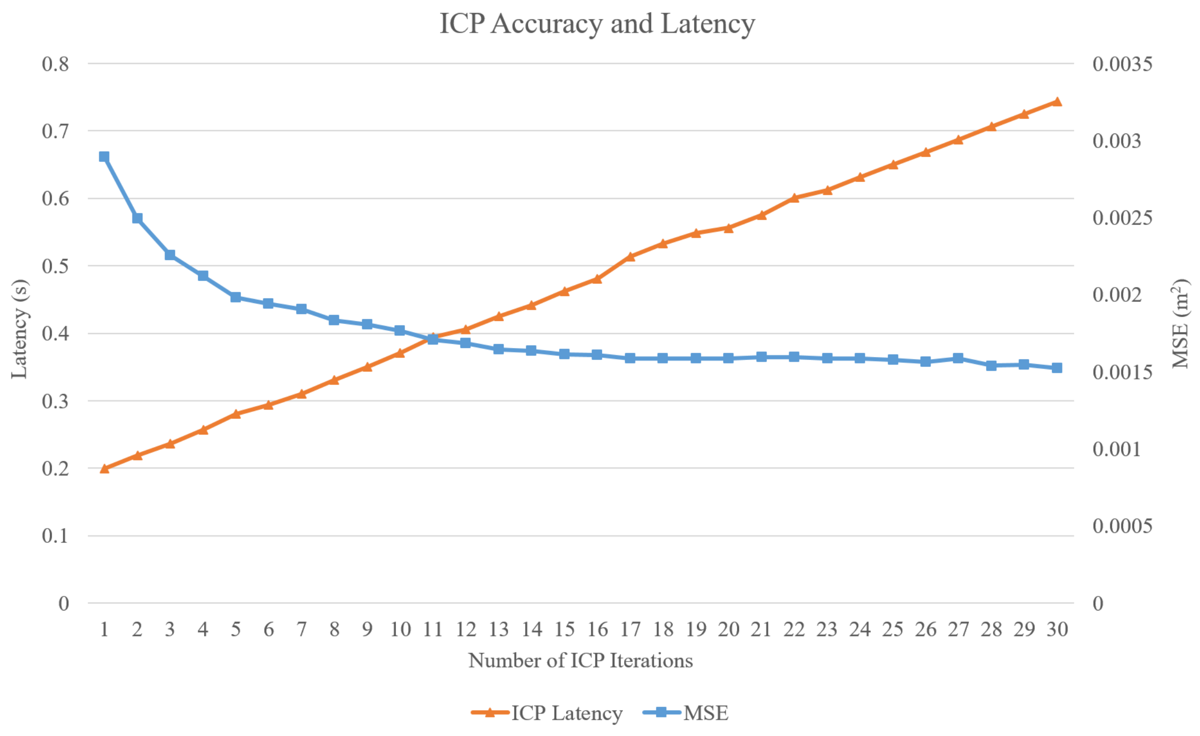


Figure 4.7 MSE and Latency by number of ICP Iterations.

4.2.4 Update Map

Once ICP is confident about the relative transformation between the latest scan and the global map, new scan points are checked for eligibility to be added to the global map. A visualization of map building in progress can be seen in Figure 4.8. For a point to be added to the map, it must meet the following criteria, given minimum point distance m, and threshold certainty c:

1. Point must at a distance be greater than m from its nearest neighbor;
2. Point must be in an unoccupied voxel grid cell;
3. The voxel cell must have a certainty greater than c that it is filled.

The map created by ICP-SLAM consists of a 3D array of grid cells, commonly referred to as voxels. Each voxel represents an even subdivision of the overall map volume. The specifications put forth for this algorithm define a 10x10x10 meter^3 physical space be mapped. The memory

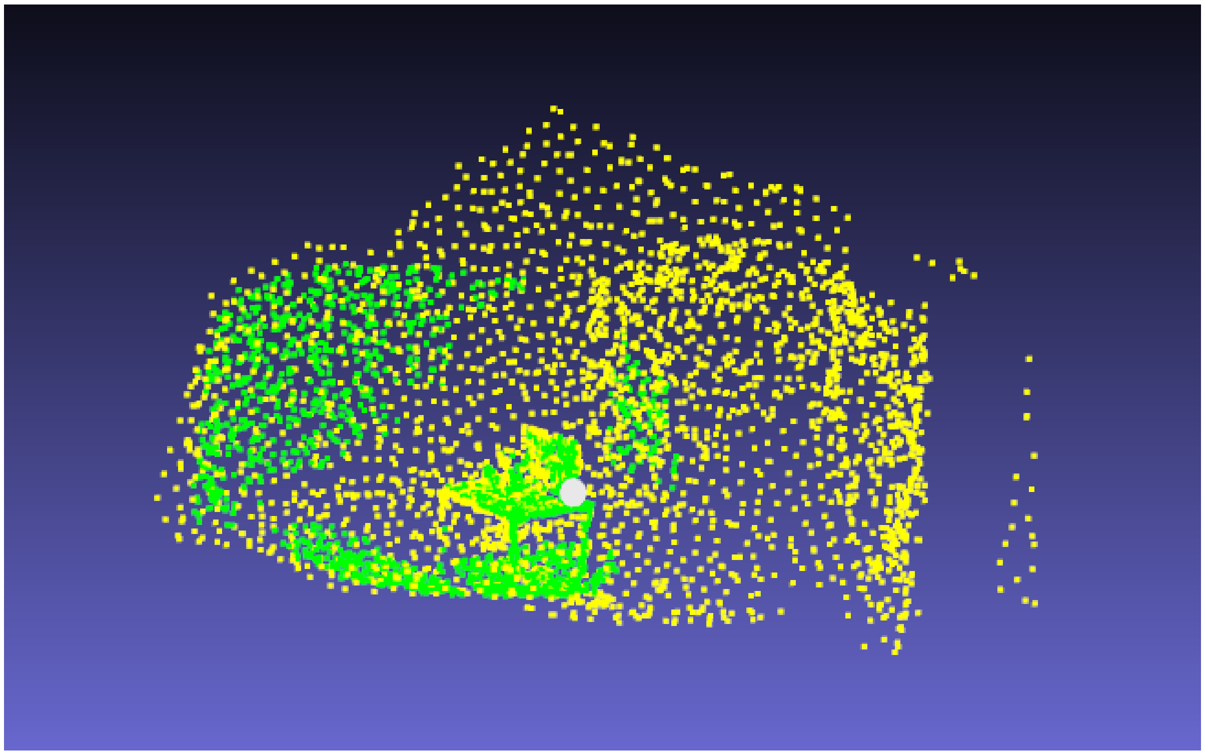


Figure 4.8 A desk scene mapping in progress. The point cloud from the latest scan is rendered in green, and the constructed map is rendered in yellow. The light grey cone shows the estimated pose of the camera.

constraints allow for a maximum map size of 300 cells to a side, allowing for a resolution of 3.3x3.3x3.3 cm^3.

Before any points are added, each new scan point is converted from true 3D space to voxel map space. The following formula is used to retrieve the respective voxel cell given a 3D point:

$$c = \frac{\text{Physical Height}}{\text{Voxel Map Height}}, \tag{4.27}$$

$$X_{voxel} = \left\lfloor \frac{X_{physical}}{c} \right\rfloor, \tag{4.28}$$

$$Y_{voxel} = \left\lfloor \frac{Y_{physical}}{c} \right\rfloor, \tag{4.29}$$

$$Z_{voxel} = \left\lfloor \frac{Z_{physical}}{c} \right\rfloor. \tag{4.30}$$

Whenever a point is detected inside a voxel, the certainty of that voxel being filled space increases. Every update step, the map updates certainty by increasing all observed solid voxels

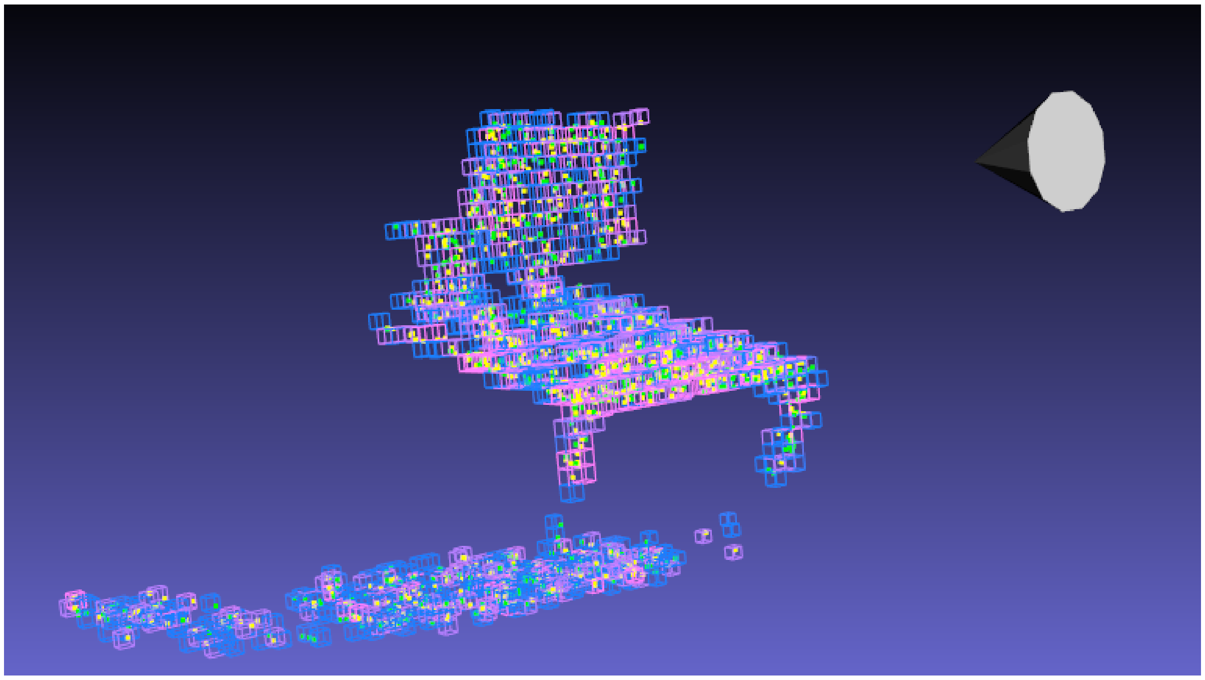


Figure 4.9 Example voxel mapping of a desk scene.

by the metric delta confidence. Once the certainty value of a cell rises above a defined threshold, the voxel is deemed to be solid and the scanned point is added to the map point cloud. A rendering of the voxel mapping of a desk point cloud can be seen in Figure 4.9.

CHAPTER 5. EVALUATION OF ICP-SLAM

The Comprehensive RGB-D Benchmark for SLAM, or CoRBS [46], is a modern RGB-D dataset that includes time stamped depth and color images combined with highly accurate ground truth position and rotation captured at 120 Hz from a 3D motion capture system. Complete scene geometries are also available via high resolution 3D scanners, claiming sub-millimeter precision. There are twenty sequences of images, capturing four different scenes with a Kinect v2. Ground truth coordinates at each frame are provided quick evaluation.

5.1 Tracking Accuracy

To fully gauge the success of a SLAM algorithm, one must both test the quality of localization and the final map produced. As this algorithm creates a sparse map of 3D voxels, the precise nature of the map's accuracy is difficult to compare to conventional 3D models. However, the tracking accuracy is characterized by two key metrics: Absolute Translational Error (ATE) and Relative Pose Error (RPE.) Absolute trajectory error is computed by taking the Root Mean Squared Error (RMSE) of the estimated position against the ground truth position at the same time step. ATE can be computed with the following formula:

$$ATE = \sqrt{\sum distance(Position_{est}, Position_{gt})^2}. \tag{5.1}$$

ATE is a good metric for evaluating long term tracking accuracy, but it doesn't give much information about granular frame to frame error. RPE computes the Root Mean Squared Error of both translation and rotation between frames, calculated as follows:

$$RPE_{translation} = \sqrt{\textstyle\sum_{i=1}^{n}\left(distance(Pos_{est}^{t-1}, Pos_{est}^{t}) - distance(Pos_{gt}^{t-1}, Pos_{gt}^{t})\right)^2}, \quad (5.2)$$

$$RPE_{rotation} = \sqrt{\textstyle\sum_{i=1}^{n}\left(Angle(Rot_{est}^{t-1}, Rot_{est}^{t}) - Angle(Rot_{gt}^{t-1}, Rot_{gt}^{t})\right)^2}. \quad (5.3)$$

Table 5.1 ATE by Method

Method	D1	D2	E4	H1
KinFu [43]	0.023	0.081	0.57	0.102
Bylow [32]	0.021	0.042	0.035	0.061
Canelhas [32]	0.014	-	0.033	0.230
SDF-TAR [40]	0.015	0.021	0.030	0.091
ICP-SLAM	0.086	0.1292	0.1202	0.2415

Table 5.2 RPE by Method

Method	D1		D2		E4		H1	
	tr. [m]	rot. [°]	tr. [m]	rot. [°]	tr. [m]	rot. [°]	tr. [m]	rot. [°]
Canelhas [32]	0.003	0.472	0.007	0.759	0.019	1.080	0.050	2.085
SDF-TAR [40]	0.003	0.442	0.006	0.768	0.009	0.993	0.020	0.844
ICP-SLAM	0.006	0.792	0.013	1.263	0.032	1.684	0.115	2.336

In 2016, [40] collected important information about the relative accuracy of related SLAM systems. ATE and RPE are evaluated using this run accuracy data, and were of great utility for comparing the relative accuracy of this algorithm. PCL's KinFu (an open source implementation of Kinect Fusion) [43], and ROS implementations of point-to-implicit methods [32] were both tested. Results for the relative performance of ICP-SLAM with top of the line SLAM implementations are shown in Tables 5.1 and 5.2. Four runs from the CoRBS dataset, two of the Desk Scene (D1 and D2), one of the Electrical Cabinet scene (E4), and one from the Human Model Scene (H1) were used for direct comparison. The data from ICP-SLAM was only collected while tracking was stable, as pose information after a combination of failures in both localization and mapping, becomes irrelevant. This is discussed further in section 5.2.

It quickly becomes apparent that tracking in ICP-SLAM is the least accurate by all metrics. Unfortunately, the localization method used in ICP-SLAM is not as accurate nor as reliable as the evaluated alternatives. This is likely due to several factors. First, the other methods use dense Signed Distance Fields (SDF) instead of sparse 3D points for their method of surface registration. SDFs represent 3D voxels with the distance from the nearest surface, instead of the probabilistic filled/unfilled model used in this thesis. This allows them to utilize surface normals and interpolation to smooth sensor noise and increase the robustness of tracking.

5.2 Stability

The largest obstacle to ICP-SLAM's success is here referred to as a death spiral. This is the combined failure of both tracking and mapping, and results in a map which cannot be correctly localized, leading to an increasingly incorrectly constructed map. The accumulation of noise and RPE leads to this problem, as is shown in Figure 5.1. ICP-SLAM will often overcorrect for noise present, adding erroneous points to the map and therefore making future tracking more difficult.

An attempt has been made to quantify how and when death spiral occurs in ICP-SLAM. CoRBS provides the average rotational and translational motion for each run. The length of time that a single run tracks successfully has been monitored and characterized as the time that the ATE of the run is less than 0.3 meters. An example run, H1, which tracks successfully for 476 frames, or 15.86 seconds in physical time, is shown below with the ATE over time. At frame 476, tracking officially fails. Some interesting behavior in Figure 5.2 shows that, in this example, the Y position becomes increasingly unstable. This zig zagging overcorrection likely led to the tracking failure.

After analyzing the runs, a severe rise in ATE always signals the beginning of death spiral behavior. Once ATE rises above 0.3m, tracking is exceedingly likely to fail. An exploration of the correlation between the stability of tracking and the camera motion is shown in Figures 5.3 and 5.4. The number of frames tracked before the ATE rose above 0.3 m is measured against the average translation and average rotation of all recorded runs.

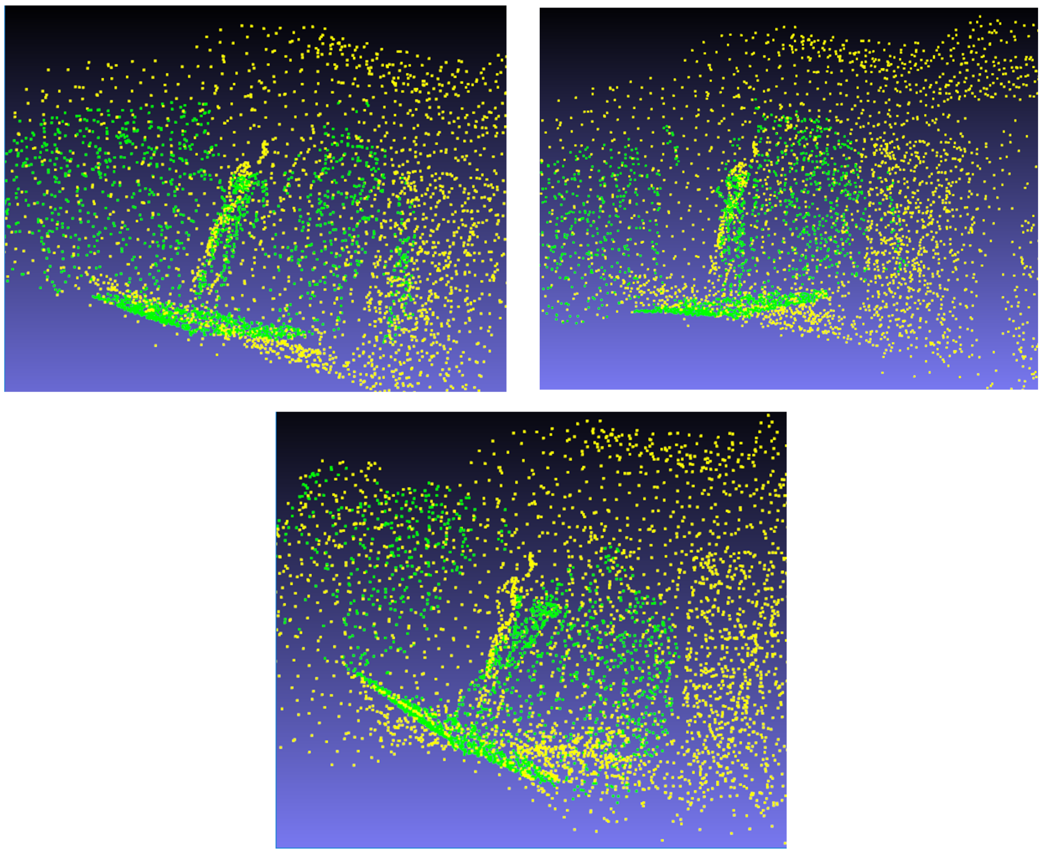


Figure 5.1 Illustration of how accumulated tracking error causes overcompensation in localization.

When the motion of the Kinect depth sensor is lower than 0.2 m/s, tracking is relatively stable and can last for over 30 seconds in a run before a death spiral occurs. A stronger correlation between translation and tracking stability is apparent, although there were three outliers that tracked for more frames than expected given the trend. This is due to a spike in translational movement around the time tracking failed.

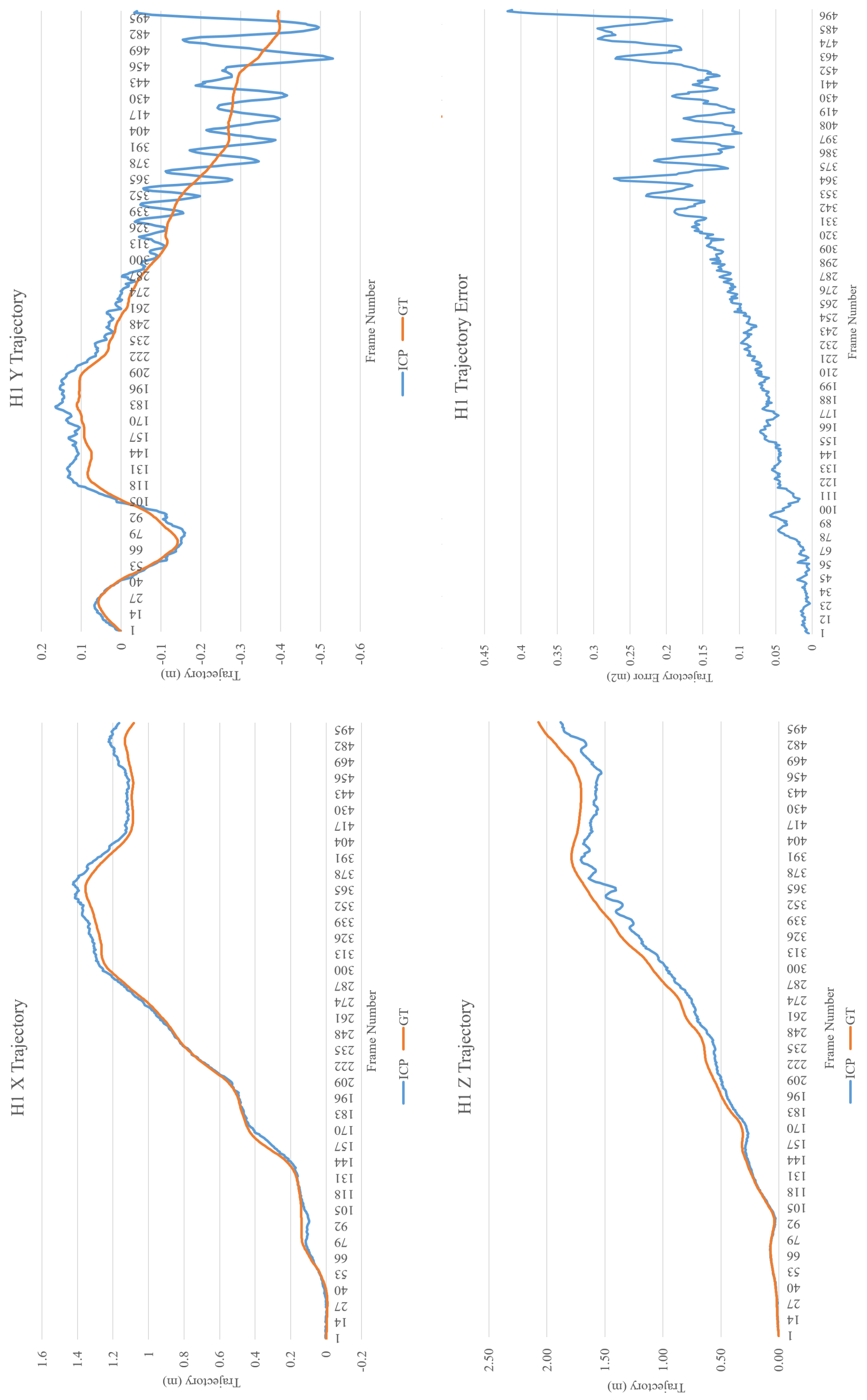


Figure 5.2 H1 Tracking accuracy over time.

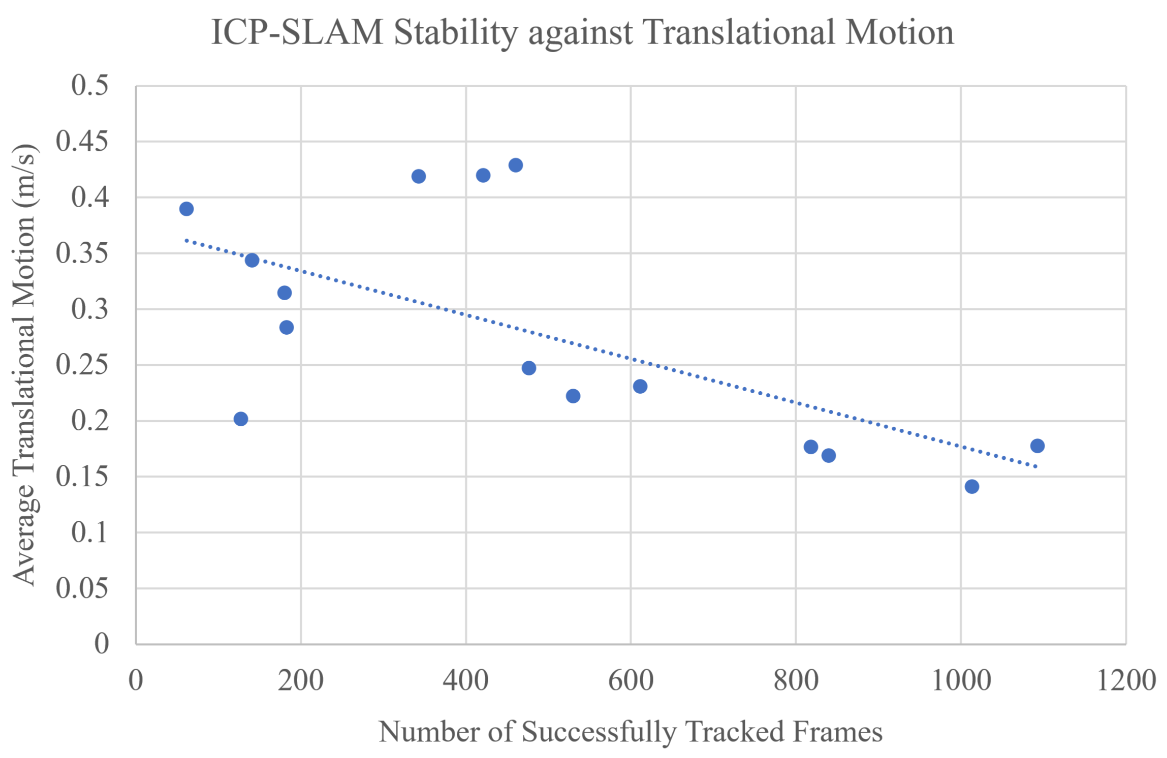


Figure 5.3 Number of Successfully Tracked Frames against Average Translational Motion, evaluated over all available CoRBS data sets.

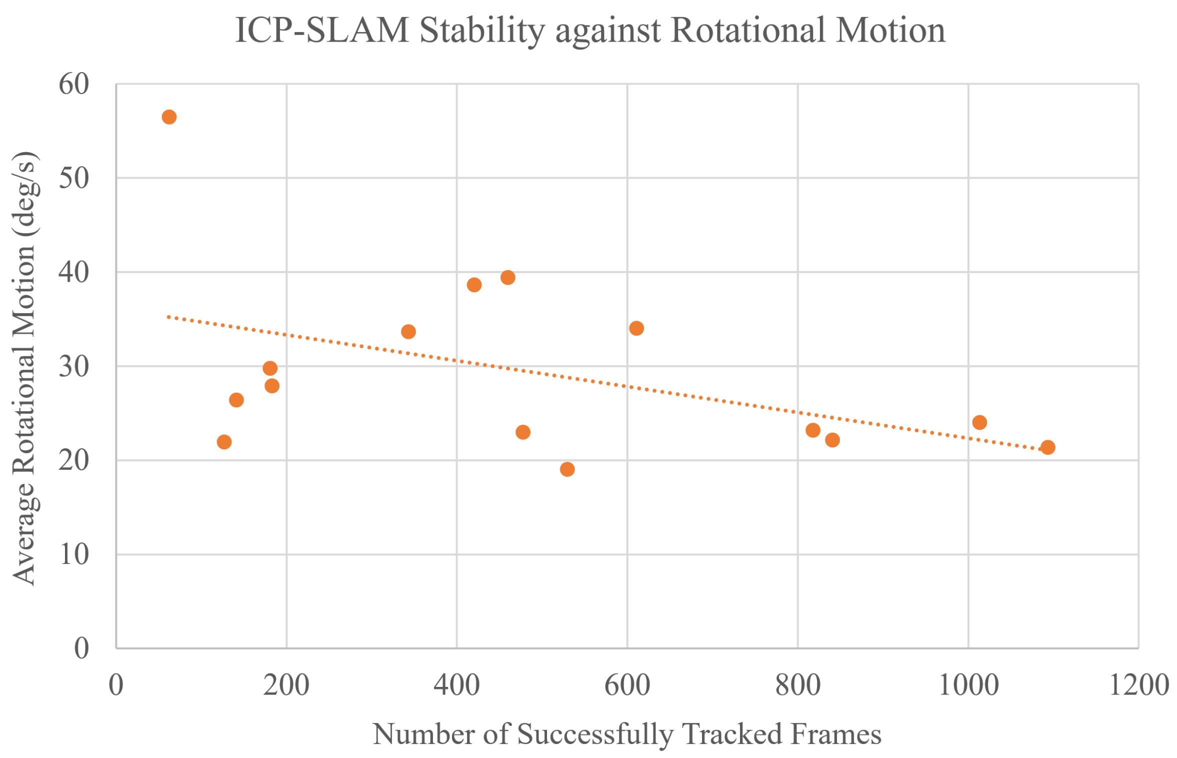


Figure 5.4 Number of Successfully Tracked Frames against Average Rotational Motion, evaluated over all available CoRBS data sets.

CHAPTER 6. A PROPOSED FPGA ACCELERATOR FOR ICP-SLAM

This section proposes two hardware architectures for acceleration of the ICP-SLAM algorithm on an embedded SoC. The tools and methods of the design process are discussed, and an analysis of different design choices is presented. Projected system performance is modeled and evaluated.

6.1 The Design Process

There is a standard process to follow when approaching hardware acceleration using reprogrammable logic. It is necessary to understand the algorithm being accelerated. This algorithm should be profiled using a software implementation to know the candidate functions for hardware acceleration. Once this is known, architectural decisions can be made to best proceed towards any defined performance goals. These include deciding which portions of the algorithm should be implemented in software or hardware; identifying how hardware accelerator resources should be distributed; deciding which resources are best suited for implementing a certain function; and selecting the target platform. Once the architecture has been designed, it is implemented using one of several hardware design tools available to the FPGA designer. The software component must be either adapted from an existing implementation or rewritten to run on the target platform. Finally, the software and hardware subsystems must be integrated together and the output validated.

The platform that is best suited the profile for a modern embedded SoC is the Avnet ZedBoard, a development board for the Zynq-7020 SoC [41]. The Zynq architecture is shown in in Figure 6.1. The chip contains a dual-core ARM Cortex A-9 Application Processor Unit (APU), a motherboard, communication channels with programmable logic, and other peripherals such

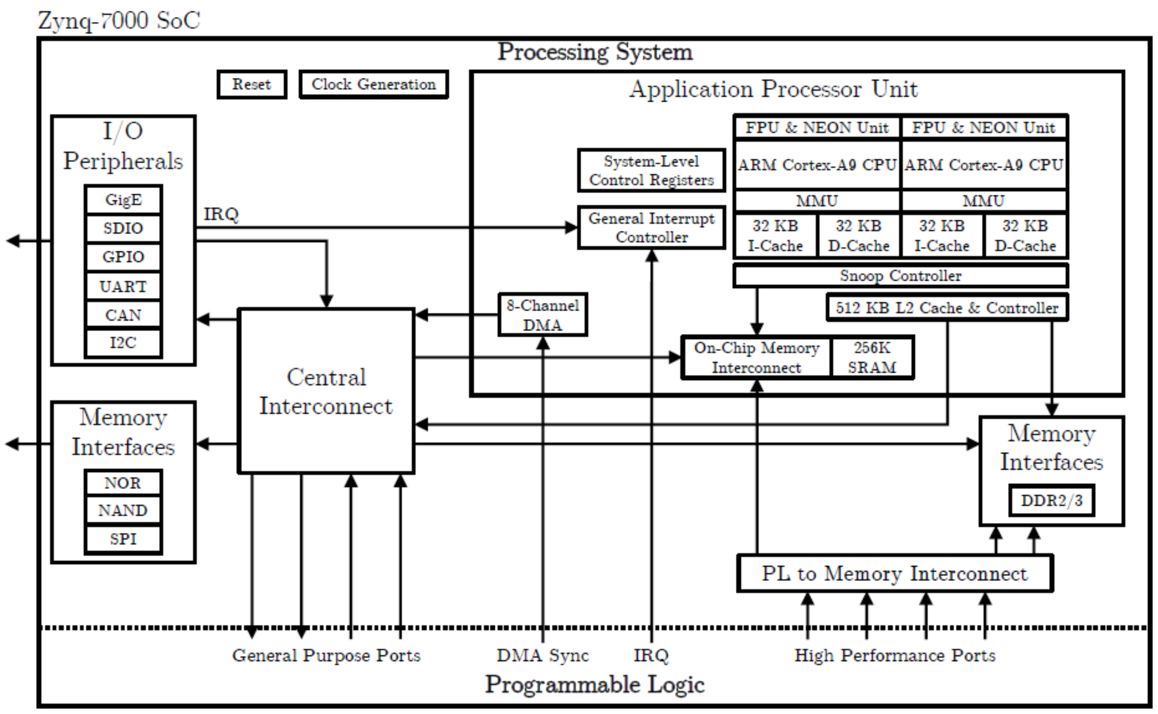


Figure 6.1 A High Level Diagram of the Zynq-7000 Architecture.

as Ethernet and GPIO pins for use with PMOD peripherals. The ARM APU runs at a frequency of 666 MHz. The ZedBoard has 512 MB of DDR3 memory running at 533 MHz, and uses an SD card bay for storage.

This application requires file storage and retrieval, as well as support for OpenCV. Therefore, a variant of the Arch Linux Distribution for ARM was selected to provide an embedded OS with a small footprint and necessary toolchain support. Using CMake, G++, and Arch, the OpenCV application can be natively compiled on the ZedBoard.

6.1.1 Profiling

To determine the performance bottlenecks in ICP-SLAM, the software-only C++ implementation was profiled using the std::chrono timing library. It was initially compiled and evaluated on a desktop computer running Windows 10 with an Intel i7-4770K CPU and 16 GB of RAM. After verification of the algorithm's accuracy, it was built and profiled on the Zedboard ARM APU. These profiling results can be seen in Table 6.1.

Table 6.1 Runtime Distribution by Function of ICP-SLAM

	ARM		X86 Desktop	
Function	Time (s)	Percent	Time (s)	Percent
Nearest Neighbor	34.72189	98.62%	0.49077	80.09%
Preprocess Image	0.192367	0.55%	0.111729	18.23%
Generate Point Cloud	0.211799	0.60%	0.006452	1.05%
Construct Covariance Matrix	0.026415	0.08%	0.000586	0.10%
SVD	0.005226	0.01%	0.0015	0.24%
Apply Transform	0.044032	0.13%	0.001291	0.21%
Compute MSE	0.004303	0.01%	0.000363	0.06%
Update Map	0.000388	0.00%	0.000142	0.02%

Table 6.1 shows that the Nearest Neighbor search takes up the majority of the runtime, both in desktop and ZedBoard implementations. For ARM, 98.62% of the runtime is spent on Nearest Neighbor search, whereas in the x86 Desktop implementation, it takes up 80.08%. The next most intensive function, image preprocessing, took up 18.23% of the x86 runtime. Interestingly, it only took up 0.55% of the Zedboard runtime, as Nearest Neighbor's runtime dominated all other functions. The breakdown of runtime is shown in Figure 6.2. Time spent loading images and visualizing the output was not included in these profiling results

6.1.2 Performance Requirements

The original performance target was 30 frames per second, which is how fast the Kinect sensor returns new depth information. It was determined that this target is not feasible given the timing constraints. Nearest Neighbor takes up the largest portion of runtime, but even if it completed instantaneously, 30 frames per second would not be possible. The rest of the algorithm takes about 0.5 seconds to complete on the ARM APU. Given this information, the target frame rate for this application on the embedded SoC was established at 2 frames per second, as this is the ideal frame rate when the runtime of Nearest Neighbor is minimized, but the remainder of the software implementation is unchanged. Thus, the target Nearest Neighbor search latency for the hardware accelerator was set to 10 ms.

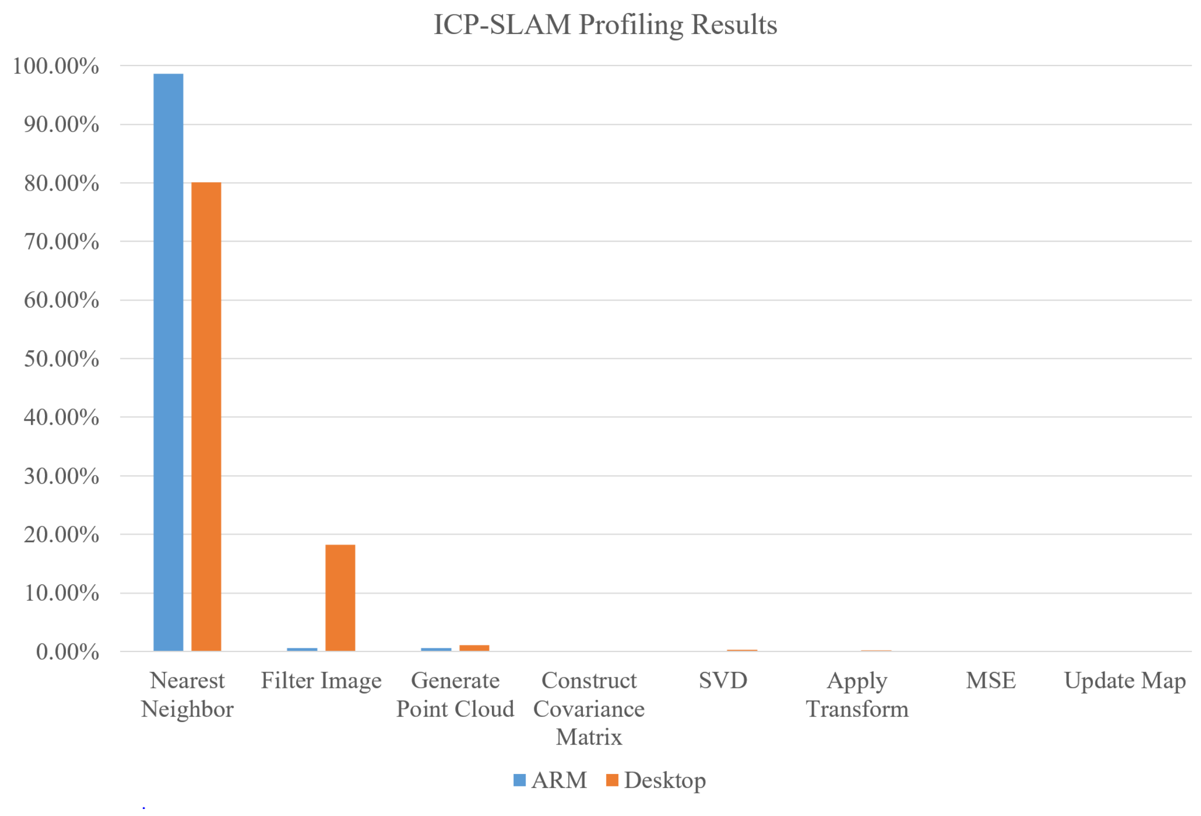


Figure 6.2 Runtime Distribution of ICP-SLAM by Function.

6.1.3 High Level Synthesis

High-level synthesis, or HLS, is a form of automated design that translates an algorithmic specification of a behavior into a functionally equivalent digital hardware implementation [44]. The code is analyzed, constrained per the targeted architecture, and timed to create an RTL implementation in a hardware description language. This HDL implementation is then synthesized to the gate level using logic synthesis tools. HLS allows for FPGA designers to quickly iterate over design parameters without worrying about the specifics of RTL design. Vivado HLS was used to synthesize the proposed architectures in this thesis using C. Preprocessor directives were used to calibrate the utilization of FPGA resources. User-created test benches allowed for quick behavioral verification of the module, and automated instrumentation tests validated the output from simulation. High-level synthesis was used to design and evaluate all hardware architectures presented in this thesis.

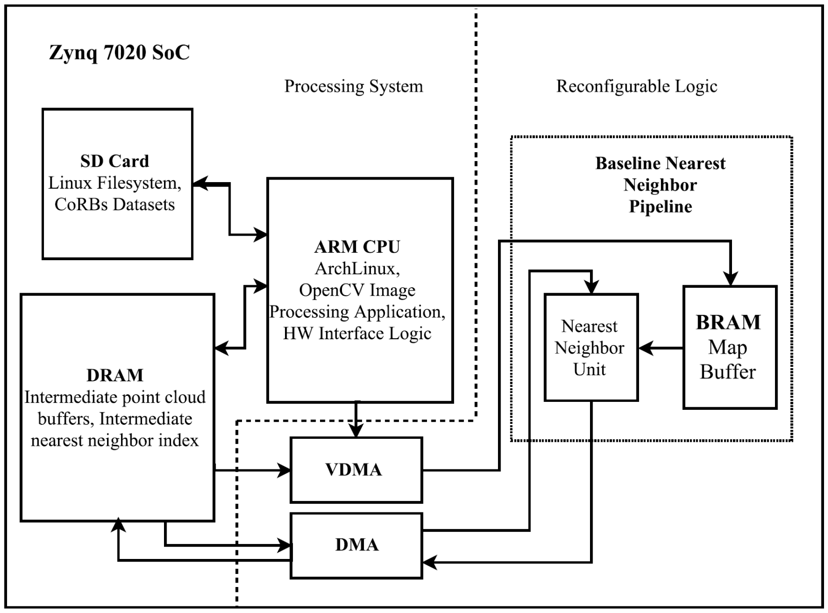


Figure 6.3 Baseline Hardware System Architecture.

6.2 Baseline Architecture

In this section, a Baseline Nearest Neighbor accelerator architecture is presented and analyzed. The system level diagram of the baseline architecture is presented in Figure 6.3. The SD Card stores the Linux filesystem as well as the CoRBs dataset used for evaluation. The ARM APU reads in files from the dataset, performs preprocessing and generates a point cloud as described in Section 4.2.2. A BRAM buffer in the reconfigurable logic is used to store the map cloud, and is populated via VDMA. This buffer is used to reduce communication latency between the Baseline Nearest Neighbor accelerator and the APU. The Baseline Nearest Neighbor Unit is synthesized on the reconfigurable logic and reads in points from the map buffer via an HLS FIFO construct.

6.2.1 Dataflow

The system level data flow between the CPU and the Baseline Nearest Neighbor Architecture can be seen in Figure 6.4. In the initialization step, the map point cloud is seeded with data from the first depth frame. This data is streamed to the map BRAM buffer using VDMA. Image Preprocessing and Point Cloud construction are then performed on scan data in software.

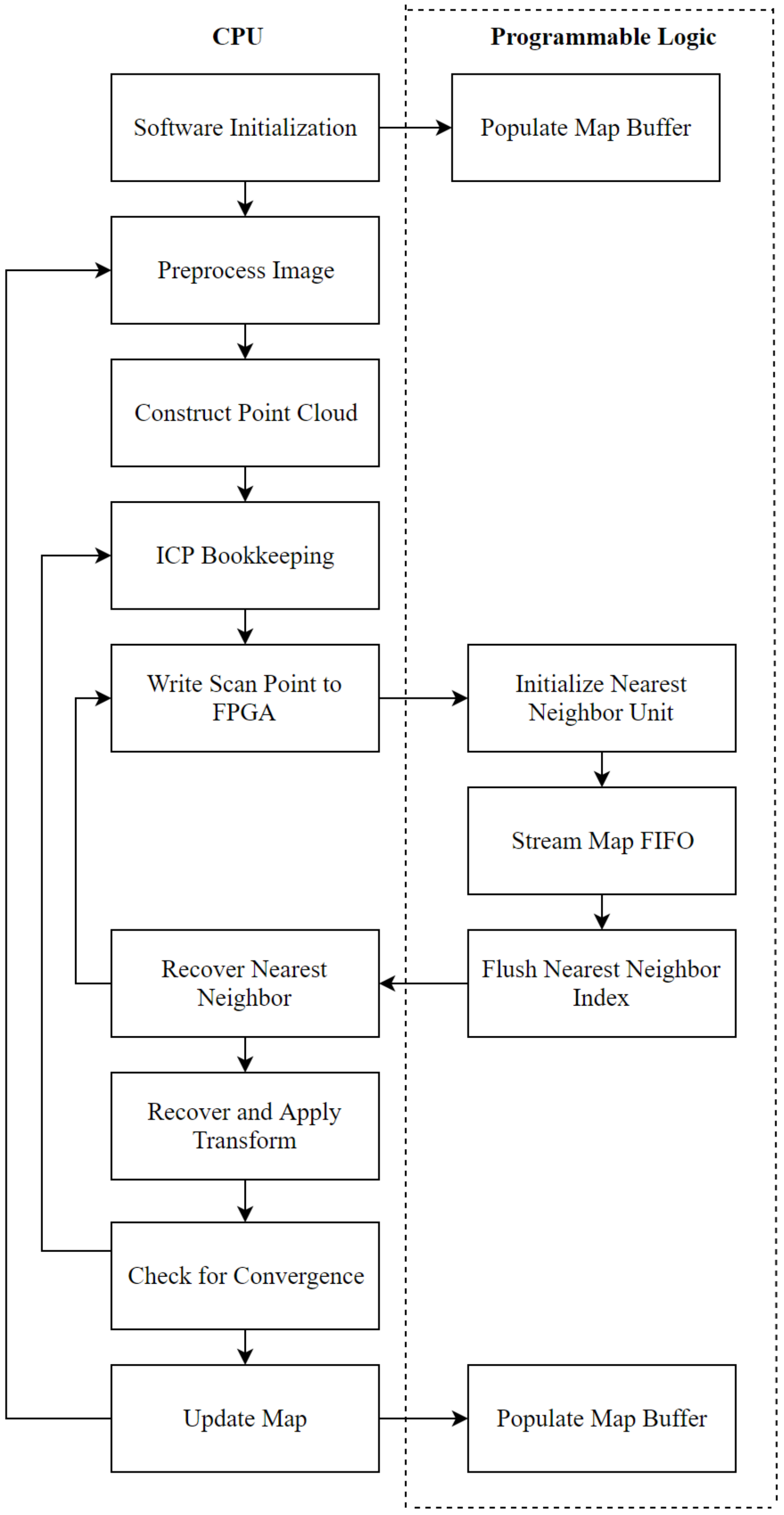
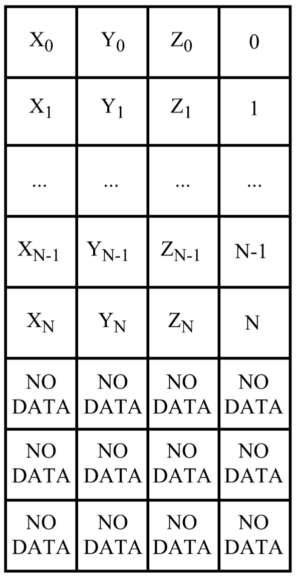


Figure 6.4 Data Flow for the Baseline Architecture.

ICP bookkeeping for preserving the cumulative transformation matrix and number of iterations is initialized. A single scan point is then written to an address in DRAM, sent to the Nearest Neighbor architecture via DMA, and initialization of the baseline architecture occurs. Once the initialization completes, the nearest neighbor search begins and all map points are streamed through the unit. The nearest neighbor index is placed into a predetermined DRAM address and the CPU stores the value for transform recovery. This process is repeated for all points in the scan cloud. Once all nearest neighbors have been recovered, the relative transformation is computed and applied to the scan cloud as described in Section 4.2.3. This process repeats until the maximum number of iterations occurs or the point clouds have converged. The map is then updated as described in section 4.2.4. All new map point indices and coordinates are streamed to the BRAM map buffer. The entire process is then repeated with the next depth image.



X_0	Y_0	Z_0	0
X_1	Y_1	Z_1	1
...	...	...	...
X_{N-1}	Y_{N-1}	Z_{N-1}	N-1
X_N	Y_N	Z_N	N
NO DATA	NO DATA	NO DATA	NO DATA
NO DATA	NO DATA	NO DATA	NO DATA
NO DATA	NO DATA	NO DATA	NO DATA

Figure 6.5 Map BRAM Buffer.

6.2.2 Map Buffer

The map buffer is a 10,000 point-long buffer for storing map points instantiated with BRAM resources. Map points are appended to the buffer in the initialization and map update steps. Each element in the map buffer stores three 24-bit fixed point coordinate values and an index. All points in the map buffer have the index field initialized to NO DATA. This constant is used by the Nearest Neighbor unit to determine when all comparisons have been made. Whenever a new point is added to the map, the values of the corresponding buffer index are updated. The buffer storing a map of size N is depicted in Figure 6.5.

6.2.3 Nearest Neighbor Unit

The architecture of the Nearest Neighbor Unit is shown in Figure 6.6. When a new scan point is ready to be processed, initialization of the Nearest Neighbor Unit begins. The X, Y, and Z values of the point are stored in the Scan Point registers, and the Min Distance and Min Index fields are initialized to predefined constants MAX DISTANCE and NO ASSOCIATON. Once these values have been initialized, the Nearest Neighbor unit enables the Map FIFO to begin streaming map points through the distance comparator as depicted in Figure 6.6. The unit compares each index to the NO DATA constant as the Nearest Neighbor search is executed. When NO DATA is seen in the map index field, the nearest neighbor index is flushed back to DRAM via DMA. This system computes one point-to-point distance in a single clock cycle via the pipelining directive in Vivado HLS. The Map point cloud is subsequently streamed through the nearest neighbor unit. This unit compares the distance between a single scan point and each map point in a single cycle, and updates the index of the closest map point. This architecture computes the nearest neighbor of a single point in $N + D$ cycles, where N is the number of map points, and D is the startup delay and communication overhead.

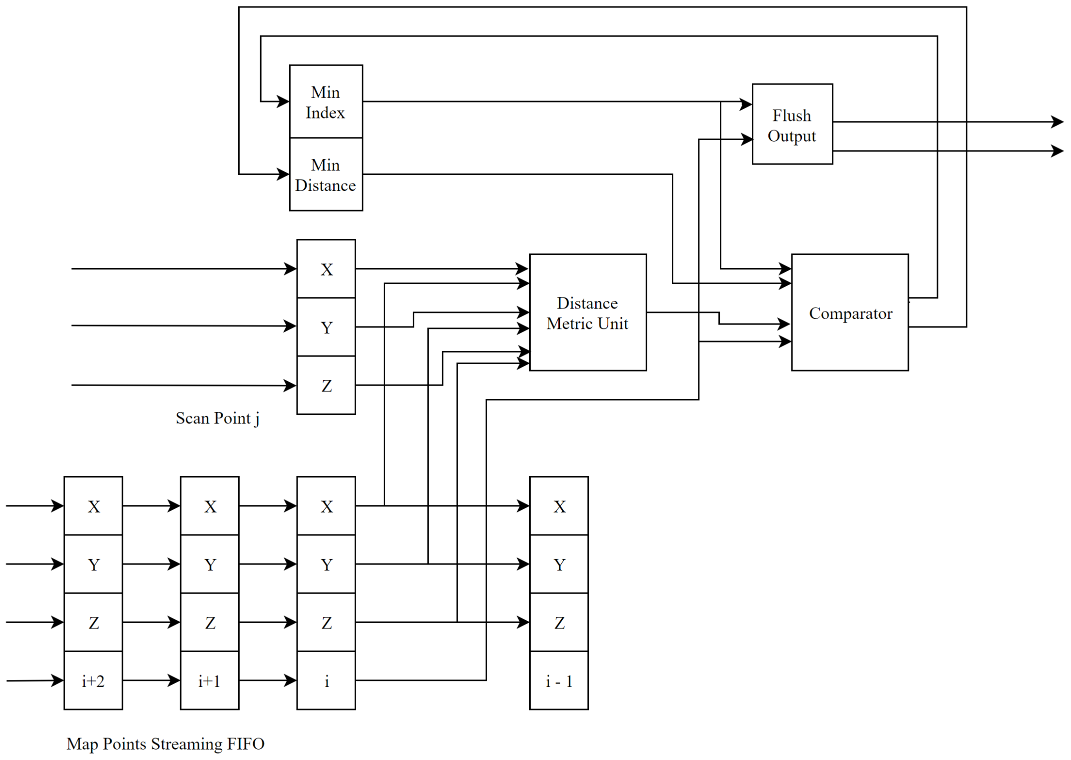


Figure 6.6 Nearest Neighbor Unit Architecture.

6.3 Accuracy and Performance

When designing the baseline architecture, there were several options to choose from with respect to the type and width of the data used for distance equations. Vivado HLS supports synthesizing solutions that use floating point, integer, and fixed point precision computations. An analysis of accuracy and resource utilization was done using an instrumented C test bench simulation. Three fixed point precision formats, 16-bit, 24-bit, and 32-bit, were evaluated against a baseline floating point nearest neighbor unit. The test bench simulated a single association computation using 3000 scan points and 10000 map points, all randomly generated, and was evaluated over the percentage of correctly labeled associations (Pass Percent) and average error of incorrect associations (Average Error.) These sizes are based off of the maximum observed scan and map sizes in ICP-SLAM, given the 1/80 sub-sampling factor used in many test cases. The results of this evaluation can be seen in Table 6.2.

Table 6.2 Nearest Neighbor Accuracy by Bit Widths.

Signed 16-Bit Fixed Point	Pass Percent	Average Error
5.11	88.57%	0.04389
6.10	94.00%	0.02166
7.9	96.60%	0.11177
8.8	16.63%	0.23438
Signed 24-Bit Fixed Point	**Pass Percent**	**Average Error**
10.14	93.73%	0.00731
9.15	95.67%	0.00681
8.16	98.70%	0.00511
7.17	97.23%	0.01040
6.18	97.23%	0.01040
5.19	88.57%	0.04389
Signed 32-Bit Fixed Point	**Pass Percent**	**Average Error**
10.22	99.15%	0.00015
9.23	98.84%	0.00038
8.24	98.70%	0.00511
7.25	97.23%	0.01040
6.26	97.23%	0.01040
5.27	88.57%	0.04388

Surprisingly, increasing the number of fractional bits in the fixed-point format did not always yield an increase in pass percentage. When the number of integer bits falls below 8, the multiplication in the distance calculation overflows for large distances, and subsequently causes large alignment errors. Inversely, when the number of fractional bits falls below 16, a lack of precision causes association discrepancies. This is most apparent in the 16-bit 8.8 format, when 8 bit fractional components proved to be too imprecise for most point pairs to associate correctly.

The total synthesized resource utilization and estimated clock rate varied by data width. Resource usage, estimated performance, and the best accuracy values for a given fixed precision width have been accumulated and summarized in Table 6.3. Throughput was calculated using the estimated scan size, latency, and clock period. Time is a measure of the number of milliseconds that a single associations computation will take to complete under ideal circumstances, at maximum allowable clock rate. ICP Time is the estimated completion time for the Nearest Neighbors step of ICP, factoring in the number of ICP iterations, the time to compute a single scan associations, and the communication delay. These values are calculated using the following formulas:

$$\text{Throughput} = \frac{ClockFrequency}{Latency * ScanSize} = \frac{1}{Latency * ScanSize * ClockPeriod}, \tag{6.1}$$

$$Time = \frac{ScanSize}{Throughput}, \tag{6.2}$$

$$Time_{ICP} = NumICPIterations * (Time + CommunicationDelay). \tag{6.3}$$

After analyzing the resource utilization data, the 24-bit 8.16 fixed point type was chosen as the target format for the Baseline Architecture. It is accurate enough while projected to have the highest throughput of all analyzed bit widths.

Although these performance estimates give a decent idea of the projected throughput, they don't model the delay in communication between the ARM processor and the programmable logic. Streaming each point from the FPGA back to DRAM via DMA takes a small communication overhead. Assuming that the DRAM is running at 533 MHz (as stated in the ZedBoard specification document [41]), and the DMA is running at the same clock speed assumed for the

Table 6.3 Performance vs Resource Utilization for the Baseline Architecture.

Name	16 bit	24 bit	32 bit	Floating Point
Resources				
DSP	2.73%	5.45%	10.91%	8.64%
FF	0.38%	0.66%	0.85%	1.95%
LUT	0.70%	1.26%	1.56%	6.82%
BRAM	14.29%	14.29%	14.29%	14.29%
Performance				
Latency (cycles)	10010	10014	10057	10057
Clock Period (ns)	10.77	8.28	9.53	9.53
Throughput (points/s)	3009	4002	3480	3480
Time (ms)	323.42	248.75	287.53	287.53
Best Time$_{ICP}$ (ms)	5174.77	3979.96	4600.47	4600.47
Best Accuracy	96.60%	98.70%	99.15%	100.00%

Nearest Neighbor unit, total system latency can be modeled using the following equation:

$$Latency = ICPIterations * (T_{associations} + (T_{clock} * Delay_{DMA} + T_{DRAM}) * NumberTransfers, \tag{6.4}$$

$$ICPIterations = 16, \quad T_{associations} = 248.75ms, \quad T_{DRAM} = \frac{1}{f_{DRAM}}, \tag{6.5}$$

$$NumberTransfers = ScanSize * 2 = 6000. \tag{6.6}$$

Figure 6.7 shows the projected latency of the baseline 8.16 fixed-point Nearest Neighbor architecture. As you can see, even in ideal communication circumstances the total latency of performing all 16 nearest neighbor associations takes 3.98 seconds. This results in a total ICP-SLAM latency of 4.46 seconds per frame, or 0.224 frames per second. This is a speedup of 7.89x over the ARM software implementation with an accuracy of 98.7%, but is still not nearly fast enough to meet the 2 FPS performance goal set earlier.

6.4 Parallel Associations Unit

Although the Baseline Architecture has shown that large speedups can be achieved with pipelined distance computations, less than 10% of the programmable logic resources were utilized. There are enough available FPGA resources to compute multiple nearest neighbors from a single stream of map points. In this section, a parallelized architecture to maximize FPGA

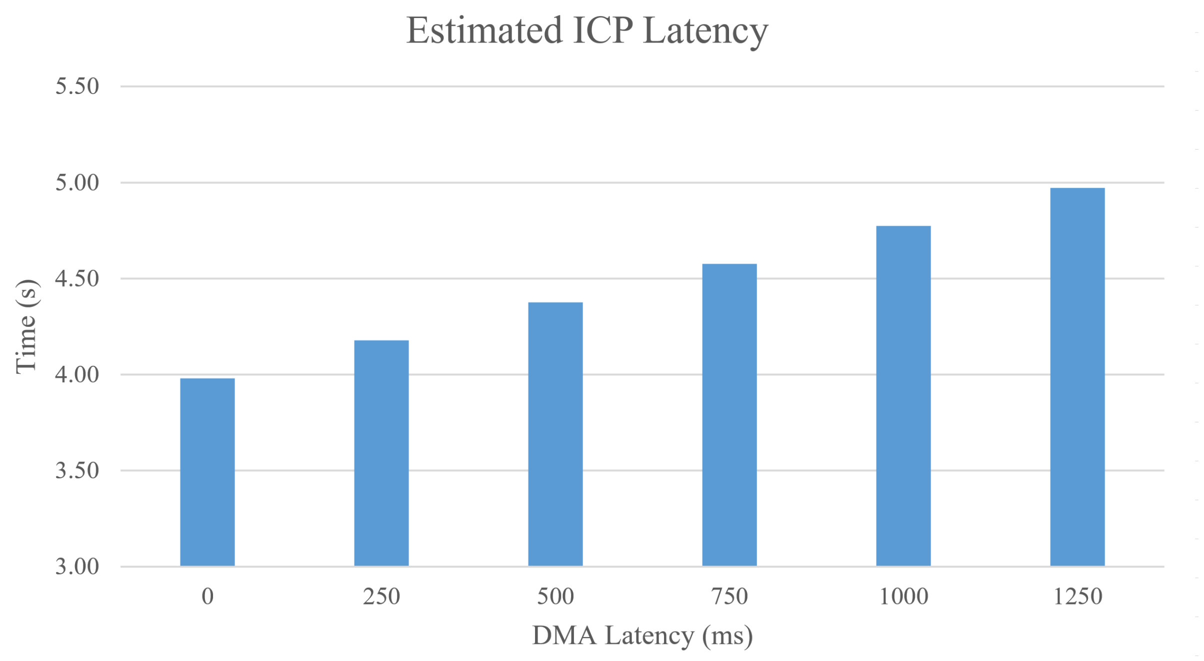


Figure 6.7 Estimated ICP Latency of the Baseline Architecture.

throughput is presented and analyzed with respect to the target performance of 2 FPS for the embedded ICP-SLAM system, referred to as the Parallel Associations Architecture.

The software-hardware architecture for the Parallel Associations Architecture is almost identical to the baseline architecture presented earlier. The one significant difference is that the computation of nearest neighbors in how performed blocks, instead of one at a time. The system data flow for the Parallel Associations Architecture is displayed in Figure 6.10. An ARM APU running Arch Linux retrieves the same datasets from the SD card, and OpenCV performs preprocessing on the image. The VDMA streams blocks of scan points to and from the re-programmable logic. Points are processed in a pipelined streaming architecture, which attempts to mitigate several of the disadvantages present in DRAM-based communication. With limited read and write ports, DRAM access tends to be slow. Computing multiple nearest neighbors in parallel reduces the number of times that data is streamed back and forth from the programmable logic.

This high throughput architecture uses a systolic array of independent Nearest Neighbor processing elements. This systolic array forms the associations pipeline, which is presented

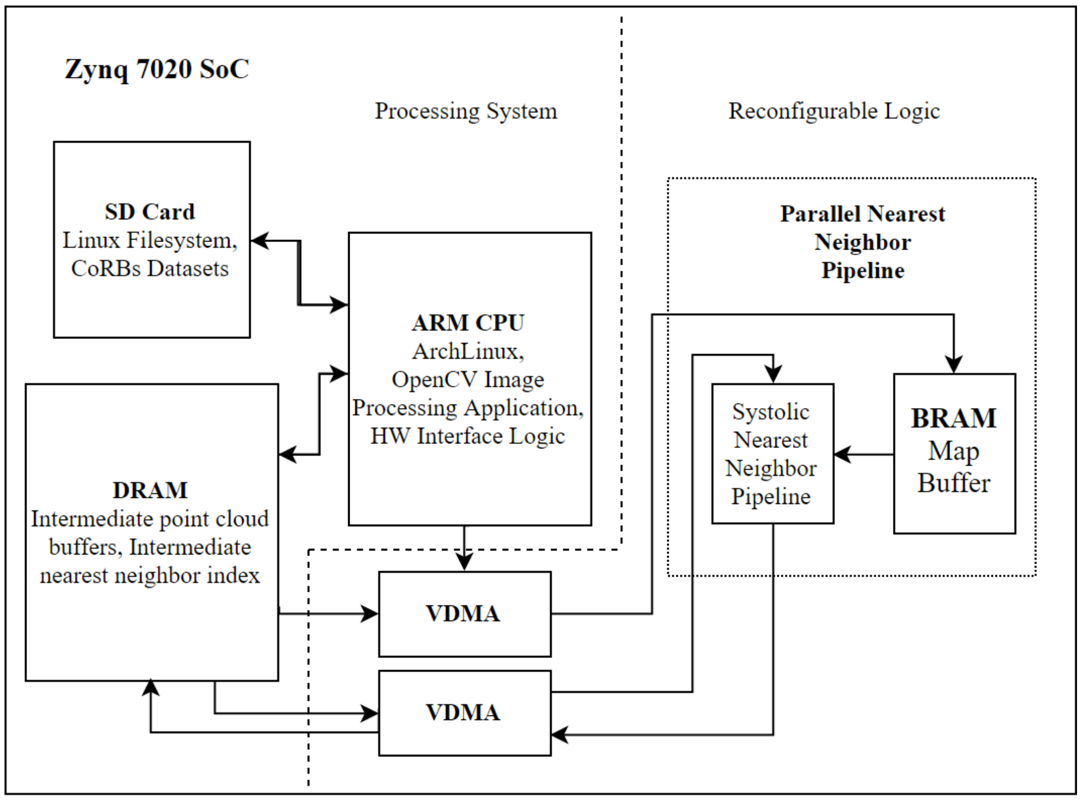


Figure 6.8 High Level Diagram of the Parallel Architecture.

in Figure 6.9. This systolic array pipeline is made up of many Nearest Neighbor Processing Elements (PE.) The individual Nearest Neighbor Processing PE has many similarities to the baseline Nearest Neighbor Unit. It contains a distance metric unit, Nearest Neighbor Index and Minimum Distance storage registers, and a mechanism to flush the output to an output FIFO. The biggest difference is the presence of a PE control unit, which interprets command signals sent by the Pipeline Control FSM. This control unit enables writing for the Minimum Distance, Nearest Neighbor Index, Scan Point registers, and Map Point registers depending on the state that the pipeline is presently in. Every clock cycle, the values of the j-th pipeline register in the Nearest Neighbor unit are written to the next PE, i.e. $j + 1$. This allows for single cycle calculation of distance values among all processing elements as long as the pipeline has valid data. The architecture of the processing element is shown Figure 6.12.

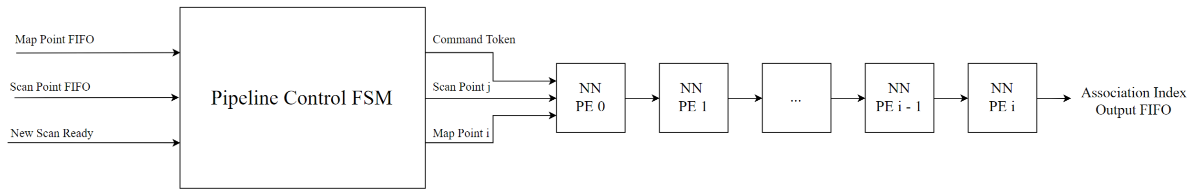


Figure 6.9 Systolic Nearest Neighbor Architecture.

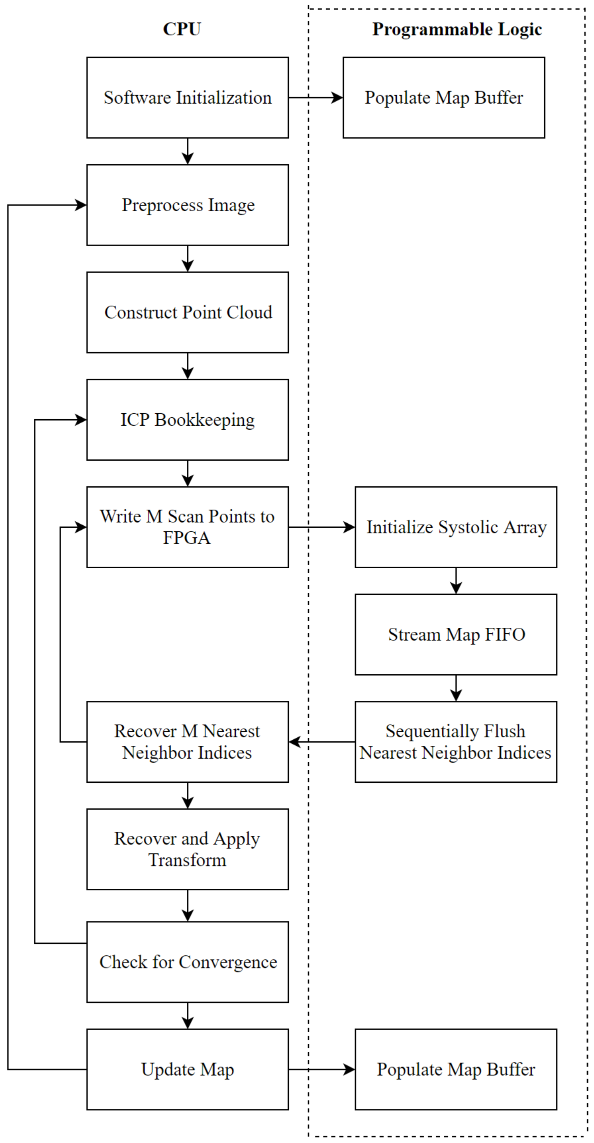


Figure 6.10 Parallel Data Flow Diagram.

The associations pipeline is controlled at the head via a proposed Pipeline Control finite state machine. This module manages the three stages of pipeline operation: loading a new batch of scan points into each nearest neighbor processing element, streaming map points through the pipeline, and flushing the nearest neighbor index of each PE to VDMA.

When implementing the Parallel Associations Architecture, limited ZedBoard system resources led to several interesting design choices. The 8.16 fixed point precision format was chosen for the x, y, and z data type as previous testing had shown that it hits an optimal combination acceptable accuracy of 98.7%, combined with fewer Digital Signal Processor resources used per multiplication operation. Each

Nearest Neighbor PE uses a fixed number of DSP48E instances from the programmable logic depending on the synthesis directives. Each element performs 3 fixed point multiplications, 3 fixed point subtractions, and 3 fixed point additions. For more details, refer to the Vivado HLS code used for implementing a single fixed point distance calculation that is shown in Figure 6.4.

```
fixed_t fixed_distance(point3f_t a, point3f_t b) {
#pragma HLS ALLOCATION instances=mul limit=N operation
        fixed_t deltaX;
        fixed_t deltaY;
        fixed_t deltaZ;

        fixed_t mDeltaX;
        fixed_t mDeltaY;
        fixed_t mDeltaZ;

        // Initialize to 0 for correct hls behavior
        fixed_t sum = 0;

        deltaX = a.x - b.x;
        deltaY = a.y - b.y;
        deltaZ = a.z - b.z;

        mDeltaX = deltaX * deltaX;
        mDeltaY = deltaY * deltaY;
        mDeltaZ = deltaZ * deltaZ;

        sum += mDeltaX + mDeltaY + mDeltaZ;

        return sum;
}
```

Figure 6.11 Vivado HLS Code to return the fixed modified Euclidean distance between two points.

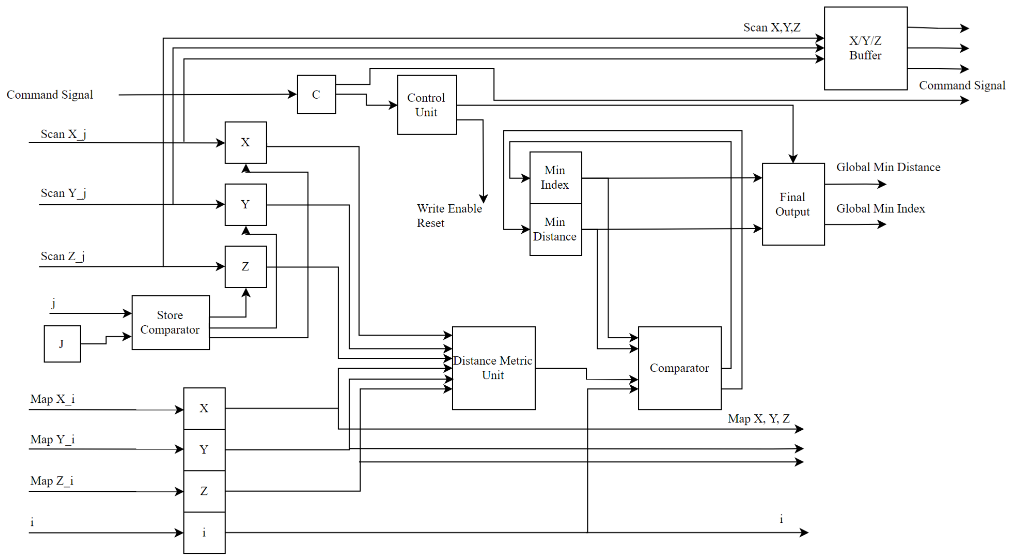


Figure 6.12 Architecture for the Parallel Nearest Neighbor Processing Element.

In this code, a preprocessor directive for multiplier resource allocation is specified. This resource directive limits the number of multiplier functional units assigned to the fixed_distance module. If this line is omitted, then the Vivado HLS synthesizes a module using 3 multipliers, prioritizing maximum pipeline throughput. Each functional multiplier uses a single DSP48E slice on chip. Although this minimizes pipeline latency when unconstrained by DSP resources, each Nearest Neighbor PE would use 6 DSP48E instances for a single distance calculation. Limiting multipliers is shown to increase total associations throughput, as more processing elements can be synthesized on chip. This number was tested using no limit, 2 multiplier limit, and 1 multiplier limit resource directives and the total FPGA resource usage is shown in Table 6.4.

Although a single parallel associations computation has the lowest latency with 3 multipliers used per PE, the total throughput is the lowest of the three due to smaller number of concurrent PE's. Inversely, reusing a single multiplier increased the pipeline initiation interval from 1 to 3, but still tripled accelerator throughput due to the larger number of concurrent PE's. Through experimentation, I have determined that the Vivado HLS synthesizes fixed point distance calculations using an equal number of multiplication and addition-subtraction functional units. This means that for every additional multiplier, latency decreases by N, where

Table 6.4 Nearest Neighbor: Resource Utilization by Multiplier Count.

	3-Mul	2-Mul	1-Mul
Number PEs	36	55	110
Chip Utilization			
DSP	98.18%	100.00%	100.00%
FF	10.07%	19.18%	42.76%
LUT	23.00%	39.67%	79.37%
BRAM	14.29%	14.29%	14.29%
Estimated Performance			
Latency	10081	20119	30229
Clock Period	8.28	8.28	8.28
Throughput	5175.47	6052.954	16114.25
Time per Computation (ms)	0.193219	0.165209	0.062057
Time per ICP (ms)	3.091507	2.643337	0.99291

N is the size of the map, and the number of DSP48E resources used increases by $2M$, where M is the number of concurrent Nearest Neighbor PE's. Therefore, the highest theoretical output came from the implementation that used the most concurrent PE's, each of which required the fewest possible DSP slices.

The same modeling of communication latency must be evaluated on this architecture as well. For this delay estimation, a higher level constant time delay for sending all scan data is modeled given the number of unknowns involved with sending larger blocks of scan data over a VDMA communication channel. In Figure 6.13, a timing estimation based on different startup cycle delays of VDMA is presented. The formula to calculate the estimated latency is as follows:

$$Latency = \left(\frac{1}{Throughput} + CommunicationDelay\right) * NumberICPIterations \tag{6.7}$$

As communication delay increases, the pipeline becomes increasingly bogged down waiting on new information from VDMA, as shown in Table 6.5 and Figure 6.13. Even with a 500 microsecond delay added to every ICP iteration, the parallel associations architecture has a projected latency of 8.99 ms, which is under the target latency of 10 ms. This leads to a total ICP-SLAM latency of 0.489 seconds per frame, or 2.044 frames per second. Furthermore, this translates to a 17.22x speedup over the ARM software implementation, and a 1.95x times

Table 6.5 Parallel Architecture - Estimated ICP Latency by Communication Delay.

Communication Delay (ms)	Estimated ICP Latency (ms)
0	0.99
0.1	2.59
0.2	4.19
0.3	5.79
0.4	7.39
0.5	8.99

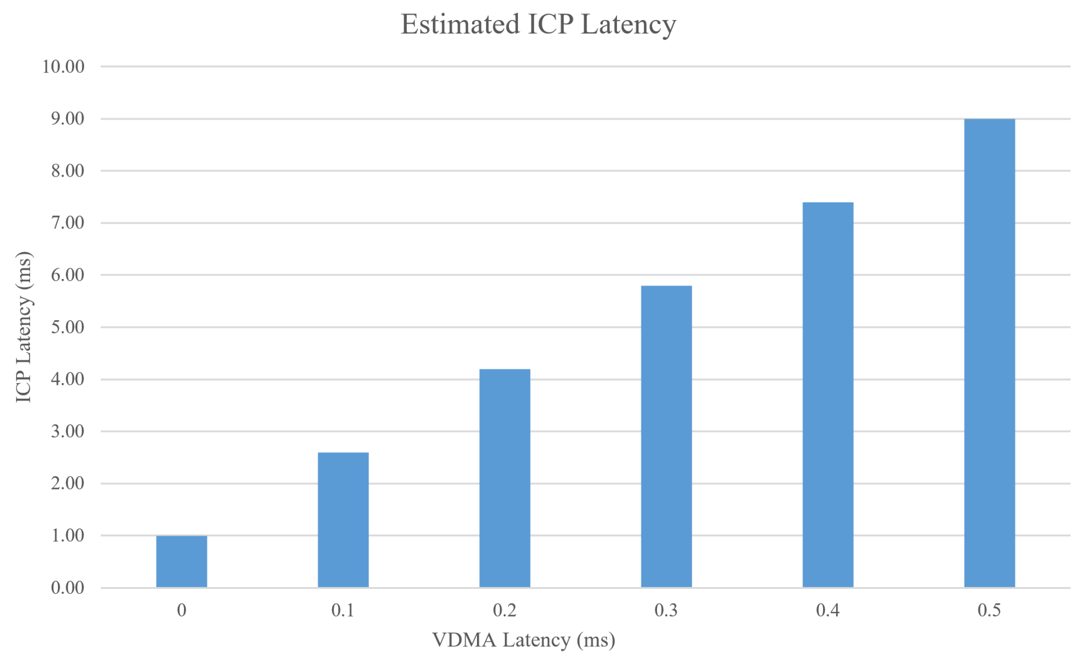


Figure 6.13 Estimated Parallel Architecture ICP Latency by Communication Latency.

speedup over the best case baseline architecture model.

6.5 Fixed-Point Evaluation

A comparison of floating point and fixed-point software implementations was done to further explore projected system behavior. Although an attempt to quantify the performance and accuracy of the Nearest Neighbor accelerator was done using synthetic test benches, evaluation of the full ICP-SLAM system output using fixed-point was done to understand projected system behavior with an integrated hardware accelerator. Fixed-point is expected to run faster on the ARM APU due to the elimination of floating point multiplications, and may yield runtime improvements on the x86 Desktop platform as well. Floating point data was converted 32-bit 10.22 signed fixed-point representation and profiled using the same steps detailed in Section 6.1.1. The results are displayed in Table 6.6. The floating point conversion step represents the time spent converting fixed-point values back to floating point for processing using SVD.

Table 6.6 Runtime Distribution by Function of Fixed-Point ICP-SLAM

	ARM		X86 Desktop	
Function	Time (s)	Percent	Time (s)	Percent
Nearest Neighbor	6.159809	91.48%	0.47122	77.34%
Preprocess Image	0.193128	2.87%	0.120502	19.78%
Generate Point Cloud	0.209230	3.11%	0.006344	1.04%
Construct Covariance Matrix	0.028392	0.42%	0.000584	0.10%
Convert To Float	0.088594	1.32%	0.007329	1.20%
SVD	0.005200	0.08%	0.001491	0.24%
Apply Transform	0.043921	0.65%	0.001293	0.21%
Compute MSE	0.004508	0.07%	0.000382	0.06%
Update Map	0.000371	0.01%	0.000144	0.02%

There were significant speedups in the Nearest Neighbor step for the ARM profiling, whereas the optimized floating point architecture of the x86 desktop saw minimal gains. The nearest neighbor step of the fixed-point ARM implementation boasted a 5.64x speedup over the same floating point function. This is due to the conversion of floating point multiplications and comparisons in the distance calculation to integer multiplication and bit shift operations, instructions shown to be much quicker in ARM.

This brings context to the projected baseline and parallel architecture speedups. Compared to the fixed-point ARM implementation, the baseline architecture achieved ICP-SLAM speedups of only 1.49x, instead of 7.89x when compared to the floating point software. The parallel architecture achieves a total projected system speedup of 13.74x, instead of 17.22x when compared to floating point.

To evaluate the accuracy of the 10.22 fixed-point implementation, the estimated pose was compared against the same CoRBS datasets and ground truth trajectories [46]. Negligible reductions in accuracy were observed as shown in Table 6.7 and Table 6.8.

The similarity of results can be attributed to two primary reasons. First, 10.22 fixed-point was shown to associate fewer than 1% points incorrectly with respect to floating point, as displayed in Table 6.3. With over 99% of associations being labeled correctly, nearly identical rotations and translations should be returned. Secondly, the granularity of a single fractional component of 10.22, 2^{-22}, is far below the 3.33 cm^3 voxel granularity of the ICP-SLAM map as discussed in Section 4.2.4.

Table 6.7 ATE by Method

Method	D1	D2	E4	H1
KinFu [43]	0.023	0.081	0.57	0.102
Bylow [32]	0.021	0.042	0.035	0.061
Canelhas [32]	0.014	-	0.033	0.230
SDF-TAR [40]	0.015	0.021	0.030	0.091
ICP-SLAM	0.086	0.1292	0.1202	0.2415
Fixed ICP-SLAM	0.087	0.1293	0.1202	0.2416

Table 6.8 RPE by Method

Method	D1		D2		E4		H1	
	tr. [m]	rot. [°]	tr. [m]	rot. [°]	tr. [m]	rot. [°]	tr. [m]	rot. [°]
Canelhas [32]	0.003	0.472	0.007	0.759	0.019	1.080	0.050	2.085
SDF-TAR [40]	0.003	0.442	0.006	0.768	0.009	0.993	0.020	0.844
ICP-SLAM	0.006	0.792	0.013	1.263	0.032	1.684	0.115	2.336
Fixed ICP-SLAM	0.006	0.796	0.013	1.270	0.035	1.688	0.117	2.336

CHAPTER 7. CONCLUSION AND FUTURE WORK

Computer vision based SLAM is a computationally intensive task rarely attempted on mobile SoC platforms. In this thesis, two hybrid CPU-FPGA architectures were proposed that aim to accelerate a depth frame based SLAM system, providing a hardware coprocessor to quickly compute the Nearest Neighbors Search portion of the Iterative Closest Point algorithm. A basic SLAM algorithm was designed and evaluated on a Desktop PC using the industry standard Kinect v2 CoRBS dataset. The ZedBoard, a Xilinx SoC, was used as an evaluation platform to profile the performance of a mobile device. Using the CPU only implementation of this SLAM algorithm yields a rough run time of 35 seconds per depth frame. Profiling the code determined that 98% of the algorithm runtime is spent in the Nearest Neighbor Search step.

Vivado HLS, a high-level synthesis tool from Xilinx, was used to design, synthesize, and simulate the accuracy of various proposed hardware acceleration solutions. A baseline architecture was predicted to yield speedups of up to 7.89x over the CPU implementation. Another parallel architecture was explored aiming to maximize system throughput. This improved architecture used 110 Nearest Neighbor Processing Elements in a systolic array, yielding a maximum theoretical speedup of 17.22x over the ARM software implementation.

Much work must be done to both increase the SLAM algorithm's accuracy as well as increase the total embedded system throughput. While 2 fps was a realistic goal for this project, 30 fps is the desired image processing framerate. More of the SLAM pipeline must be implemented in hardware, notably the entire ICP algorithm including rotation and SVD, as well as image preprocessing and point cloud generation. If the system can be tuned to reduce the resources allocated the Nearest Neighbor pipeline in exchange for placing more SLAM functionality in programmable logic, performance could be increased massively.

Purely depth based ICP-SLAM has been demonstrated to work on limited examples with low rotational and translational motion, but using additional sensors commonly present on embedded platforms such as an IMU or wheel encoder for robots could yield improved tracking accuracy. Fusing sensor input into a probabilistic model using an algorithm such as an Extended Kalman Filter could aid localization in frames when quick motions degrade the quality of tracking.

Pure point-to-point distance metric has been demonstrated to work for ICP convergence [3], but the point to plane error metric would likely yield improved results for the type of indoor environments seen in the CoRBS dataset. Point-to-plane assumes that points with similar normals to a plane either in front or behind are likely to be a part of that plane, and tends to perform well given many flat surfaces in an environment. Point-to-plane is used in many advanced SLAM systems such as the aforementioned Kinect Fusion system [28], and seems to be a promising metric that takes advantage of dense depth information provided by the Kinect v2.

Representing the map as a voxel grid is a simplistic way of mapping, and holds no helpful information to improve the alignment as localization can't be done accurately using voxels of 3.3 cm to a side. Using Signed Distance Fields (SDF) for map representation is a modern SLAM practice, also used in Kinect Fusion, that is useful for rendering, tracking, and aligning depth information in a consistent global model.

Using ICP alone yields decent results for small rotational and translational distance, but fails given a quick enough motion. Techniques have been proposed to mitigate ICP's short-comings using combined visual feature matching. A high-level approximation of rigid motion approximation combined with ICP for pose refinement has yielded impressive results with far more resilience to noise [38]. Implementing a hybrid feature-ICP tracking solution appears as a viable next step for improving tracking accuracy.

BIBLIOGRAPHY

[1] Sunil Arya and David M. Mount. Approximate range searching. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 172–181. ACM, 1995.

[2] Michael S. Belshaw and Michael A. Greenspan. A high speed iterative closest point tracker on an FPGA platform. In *Proceedings of the Computer Vision Workshop, IEEE 12th International Conference on Computer Vision (ICCV)*, pages 1449–1456. IEEE, 2009.

[3] Paul J. Besl and Neil D. McKay. Method for registration of 3-D shapes. In Paul S. Schenker, editor, *Sensor Fusion IV: Control Paradigms and Data Structures*, pages 586–606. International Society for Optics and Photonics, 1992.

[4] Gérard Blais and Martin D. Levine. Registering multiview range data to create 3D computer objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):820–824, 1995.

[5] Nicolas Burrus. Kinect for windows sensor components and specifications. https://msdn.microsoft.com/en-us/library/jj131033.aspx, 2014. Accessed: 2017-06-03.

[6] David M. Cole and Paul M. Newman. Using laser range data for 3D SLAM in outdoor environments. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation, (ICRA) 2006.*, pages 1556–1563. IEEE, 2006.

[7] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach.* Morgan Kaufmann Publishers, Burlington, MA, 1999.

[8] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6), 2007.

[9] Albert Diosi and Lindsay Kleeman. Laser scan matching in polar coordinates with application to SLAM. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS) 2005.*, pages 3317–3322. IEEE, 2005.

[10] Nikolas Engelhard, Felix Endres, Jürgen Hess, Jürgen Sturm, and Wolfram Burgard. Real-time 3D visual SLAM with a hand-held RGB-D camera. In *Proceedings of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, volume 180, pages 1–15, 2011.

[11] Allan L. Fisher. Systolic algorithms for running order statistics in signal and image processing. In *VLSI Systems and Computations*, pages 265–272. Springer, 1981.

[12] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A novel SIMD architecture for the cell heterogeneous chip-multiprocessor. In *Hot Chips XVII Symposium (HCS)*, pages 1–31. IEEE, 2005.

[13] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision.* Cambridge university press, Cambridge, UK, 2003.

[14] Georg Klein and David Murray. Parallel tracking and mapping for small AR workspaces. In *Proceedings of the 6th IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 225–234. IEEE, 2007.

[15] Peter M. Kogge. *The architecture of pipelined computers.* CRC Press, New York, USA, 1981.

[16] V.K. Prasanna Kumar and Y-C. Tsai. On synthesizing optimal family of linear systolic arrays for matrix multiplication. *IEEE Transactions on Computers*, 40(6):770–774, 1991.

[17] Sun Yuan Kung. VLSI array processors. In *Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy*, volume 1, page 685. Prentice Hall, 1988.

[18] Christian Langis, Michael Greenspan, and Guy Godin. The parallel iterative closest point algorithm. In *Proceedings of the Third International Conference on 3-D Digital Imaging and Modeling*, pages 195–202. IEEE, 2001.

[19] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55. ACM, 2001.

[20] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 579–588. ACM, 2006.

[21] Charles E. Leiserson. Systolic priority queues. Technical report, DTIC Document, 1979.

[22] Tomasz Malisiewicz. The future of real-time SLAM and deep learning vs SLAM. http://www.computervisionblog.com/2016/01/why-slam-matters-future-of-real-time.html. Accessed: 2017-06-18.

[23] Takeshi Masuda, Katsuhiko Sakaue, and Naokazu Yokoya. Registration and integration of multiple range images for 3-D model construction. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume 1, pages 879–883. IEEE, 1996.

[24] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[25] Christopher Mei, Gabe Sibley, Mark Cummins, Paul M. Newman, and Ian D. Reid. A constant-time efficient stereo slam system. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 1–11, 2009.

[26] Maxime Meilland, Andrew Ian Comport, and Patrick Rives. Dense visual mapping of large scale environments for real-time localisation. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4242–4248. IEEE, 2011.

[27] Hans P. Moravec. Sensor fusion in certainty grids for mobile robots. *AI magazine*, 9(2):61, 1988.

[28] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *10th IEEE international symposium on Mixed and augmented reality (ISMAR)*, pages 127–136. IEEE, 2011.

[29] Viet Nguyen, Ahad Harati, and Roland Siegwart. A lightweight slam algorithm using orthogonal planes for indoor mobile robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 658–663. IEEE, 2007.

[30] Janosch Nikolic, Joern Rehder, Michael Burri, Pascal Gohl, Stefan Leutenegger, Paul T. Furgale, and Roland Siegwart. A synchronized visual-inertial sensor system with fpga pre-processing for accurate real-time SLAM. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 431–437. IEEE, 2014.

[31] Taihú Pire, Thomas Fischer, Javier Civera, Pablo De Cristóforis, and Julio Jacobo Berlles. Stereo parallel tracking and mapping for robot localization. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1373–1378. IEEE, 2015.

[32] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[33] Jorge Rodriguez-Araujo, Juan J. Rodriguez-Andina, Jose Farina, and Mo-Yuen Chow. Field-programmable system-on-chip for localization of UGVs in an indoor ispace. *IEEE Transactions on Industrial Informatics*, 10(2):1033–1043, 2014.

[34] Dan Roggow. Real-time ellipse detection on an embedded reconfigurable system-on-chip. Master's thesis, Department of Electrical and Computer Engineering, Iowa State University, 2017.

[35] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM, 1995.

[36] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the ICP algorithm. In *Proceedings of the Third International Conference on 3-D Digital Imaging and Modeling*, pages 145–152. IEEE, 2001.

[37] Will J. Schroeder, Bill Lorensen, and Ken Martin. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.

[38] Gregory C. Sharp, Sang W. Lee, and David K. Wehe. ICP registration using invariant features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(1):90–102, 2002.

[39] David A. Simon. *Fast and accurate shape-based registration*. PhD thesis, Carnegie Mellon University, 1996.

[40] Miroslava Slavcheva and Slobodan Ilic. SDF-TAR: Parallel tracking and refinement in RGB-D data using volumetric registration. In *Proceedings of the British Machine Vision Conference*. British Machine Vision Association, 2016.

[41] Avnet Team. Zedboard. http://zedboard.org/, 2017. Accessed: 2017-06-09.

[42] OpenCV Developers Team. Opencv. http://opencv.org, 2017. Accessed: 2017-06-09.

[43] PCL Developers Team. Point cloud library. http://www.pointclouds.org/, 2017. Accessed: 2017-05-02.

[44] Xilinx Team. Vivado high-level synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2017. Accessed: 2017-06-09.

[45] Masahiro Tomono. Robust 3d slam with a stereo camera based on an edge-point icp algorithm. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, pages 4306–4311. IEEE, 2009.

[46] Oliver Wasenmüller, Marcel Meyer, and Didier Stricker. CoRBS: Comprehensive RGB-D benchmark for SLAM using kinect v2. In *Proceedings of IEEE Winter Conference on Applications of Computer Vision (WACV), pages=1–7, year=2016, organization=IEEE.*