# Hardware Accelerator of Bundle Adjustment Algorithm

Yichen Wang
yi0847wa-s@student.lu.se
Yuzhe Zhang
yuzhe.zhang.2623@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu, Lucas Ferreira, Ilayda Yaman

Examiner: Erik Larsson

October 30, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

# Abstract

With the popularization and development of CV technology, the SLAM algorithm is widely used in scenarios such as self-driving cars and autonomous navigation robots. As a key step in the SLAM system, the BA algorithm is responsible for optimizing camera parameters and 3D point coordinates. BA obtains more accurate estimates by shrinking the re-projection error. So as to support the SLAM system in building a more accurate 3D model of the surrounding environment and a more reliable trajectory of the moving camera. However, due to the high computational complexity of the BA algorithm, its computational efficiency becomes a bottleneck limiting the real-time performance of SLAM. In order to improve the performance of the BA algorithm in practical applications, the goal of our thesis work is to build and implement an efficient hardware accelerator for BA.

The main tasks are as follows:

- Theoretical Understanding: To fully understand the theory and ideas of the BA algorithm, do a comprehensive review of the related literature. For hardware implementation, this understanding provides a strong basis.
- High-Level Architecture: Create a high-level architecture with an emphasis on the CC and JU components. This work offers a well-organized framework for the BA algorithm and points out the parts that affect performance.
- Hardware Implementation: Create specialized hardware accelerators by translating the high-level architecture into particular hardware designs. The hardware accelerator needs to effectively process the JU and CC components of the BA algorithm. We aim to focus on the performance and accuracy aspects of the BA accelerator.

# Popular Science Summary

Are you interested in technologies such as robotics and self-driving cars? An important technology in these fields is called SLAM, which enables robots to perceive and navigate. In SLAM, there is a key step called the BA algorithm, which can improve the positioning and map construction accuracy of robots in unknown environments. Let's take a look at this fantastic algorithm together.

SLAM is a technology that enables a robot to simultaneously build a map of the surrounding environment and determine their own position in an unknown environment. Imagine that when a robot moves in an unfamiliar area, it needs to constantly perceive the surrounding environment through its "eyes," like us humans, and let its "brain" remember the map information around it and infer its location. This ensures autonomous real-time navigation when it is difficult to obtain GPS signals or there is no network. SLAM is a key technology that helps robots accomplish this task.

The BA algorithm is an important step in SLAM. Its goal is to improve the accuracy of the robot's "eyes" in the SLAM process by optimizing the robot's "brain"'s estimation of its own position and surrounding map information. Simply put, it finds the best model by constantly adjusting camera parameters and the 3D point positions, making positioning and mapping more accurate. It corrects the camera distortion and finds the best camera pose. At the same time, by optimizing the positions of 3D points, the accuracy of the map can be further improved. In this way, a robot or self-driving car will be more reliable at navigating and perceiving its surrounding environment.

BA's algorithm structure has many matrix operations. This requires more computing power to support SLAM implemented in software to reduce time consumption. The hardware accelerator we designed is specifically used to accelerate the most time-consuming process of the BA algorithm. This helps to improve the real-time performance and practicality of the entire SLAM system.

SLAM and BA algorithms in CV technology give robots the ability to perceive and navigate. The development of these technologies will bring better performance and reliability to applications in self-driving cars, robotics, and other fields, allowing the "eyes" of robots to have a more accurate field of vision!

# Table of Contents

**1 Introduction** **1**
1.1 Background . . . 1
1.2 Related work . . . 1
1.3 Goals and Challenges . . . 2

**2 Methodology** **3**
2.1 System Analysis and High-Level Architecture . . . 4
2.2 Arithmetic Circuit Design and Optimization . . . 6
2.3 System Integration and Evaluation . . . 6

**3 Theory** **9**
3.1 Fundamentals of Bundle Adjustments . . . 9
3.2 Nonlinear optimization . . . 11
3.3 Implementation of LM Algorithm . . . 12
3.4 Software implementation of BA using MATLAB . . . 16

**4 Hardware Implementation and Modification** **21**
4.1 Hardware Implementation . . . 21
4.2 Hardware Modification . . . 25

**5 Performance and Correctness Verification** **29**
5.1 Computational Correctness Verification . . . 30
5.2 Performance Comparison . . . 31
5.3 Estimated Energy Saving Comparison . . . 33

**6 Conclusion and Future Works** **35**
6.1 Conclusion . . . 35
6.2 Future Works . . . 35

**References** **39**

# List of Figures

2.1 "Top-down" Methodology Chart. 3
2.2 Calculations Involved in JU and CC. 4
2.3 Initial Hardware Structure. 5

3.1 BA Image Model 9
3.2 BA Graph Model 10
3.3 LM Flowchart 18
3.4 Frames taken by camera at different positions 18
3.5 Zoom in plot of bundle adjustment 19
3.6 RMS variation in each iteration 19
3.7 RMS error at different fractional word lengths of our fixed-point number. 20

4.1 Data Format and Actual Value of "Q4.16". 21
4.2 Structure Diagram of Carry Look-ahead Adder. 22
4.3 8-bits "Shift-Sub" Divider Algorithm Flow. 23
4.4 Accuracy Comparison of Different Newton Iterations. 25
4.5 ASMD of our controller. 26
4.6 Modified Hardware Structure. 26

5.1 Integrated System Block Diagram. 29
5.2 Correctness Verification Flowchart. 30
5.3 RMS Comparison for 10 Iterations. 31
5.4 Hardware Speedup Compare to Software (SBA). 32
5.5 Hardware Speedup Compare to Software (Ceres). 32

# List of Tables

3.1 LM algorithm data range . . . . . . . . . . . . . . . . . . . . . . . . 20

4.1 Timing and Resource Utilization Comparison . . . . . . . . . . . . . 22
4.2 Performance and Area of 20-bits "Shift-Sub" . . . . . . . . . . . . . 24
4.3 Performance and Area of 20-bits "Newton" . . . . . . . . . . . . . . 24
4.4 Performance and Area Comparison after Modification . . . . . . . . 27

5.1 Performance and Area after System Integration . . . . . . . . . . . . 30
5.2 Speedup of Hardware Compared to Software for 10 Iterations. . . . . 31
5.3 Estimated Energy Saving Comparison . . . . . . . . . . . . . . . . . 33

# Chapter 1 Introduction

In recent years, the development of autonomous driving and other related fields has led to an increased application of SLAM in embedded real-time systems. SLAM is an algorithm that enables a robot to determine its position and orientation within an unknown environment by repeatedly observing map features, such as wall corners and pillars, during movement. This process allows for simultaneous positioning and map construction, which is essential for effective navigation and operation in various settings.

## 1.1 Background

The BA method plays a crucial role in refining the SLAM process. This optimization technique refines estimated camera poses and 3D point positions by minimizing the error between observed and predicted image points. In the context of SLAM, BA contributes to improving the accuracy and robustness of the reconstructed map and estimated trajectory, resulting in more precise and reliable navigation for autonomous systems. However, using traditional BA software for real-time processing will take longer to compute, making it difficult to meet the performance and accuracy requirements of real-time SLAM in various scenarios. Therefore, implementing an efficient BA algorithm that can run in real time becomes the key to solving the practical application of SLAM.

BA is a computationally intensive algorithm for optimal estimation of map data, and high-latency calculations limit the application scenarios of BA. Recent research focuses on how to achieve efficient computing based on software and often focuses on optimizing computational efficiency and reducing processing times. Some of the methods, including parallelization, sparse matrices, and enhanced robustness, can inspire the design of hardware accelerators. Compared to instruction-based software, implementing the optimization of algorithmic operators directly into hardware offers significant performance advantages.

## 1.2 Related work

For the acceleration of the BA algorithm, researchers have tried many methods, including software optimization and hardware acceleration. In this section, we review these related works.

Triggs et al. introduced the application of BA in CV in 1999 [1], including applications in 3D structure restoration, visual SLAM, and other fields. The book summarizes the traditional BA algorithm and the newly proposed algorithm and makes a comprehensive introduction and evaluation of BA, which provides an important reference for follow-up BA research.

In 2021, Liu et al. [2] proposed a BA accelerator based on an FPGA hardware accelerator that achieves parallel acceleration of BA by utilizing the 3D point distribution in the input image. Experiments show that the accelerator can achieve efficient BA calculations and significantly reduce running time and energy consumption.

Manolis et al. introduced the SBA software package in 2009 [3], which provides a general sparse matrix solution for BA and supports a variety of constraints and optimization algorithms. This software package is widely used in various aspects of the field of CV, providing researchers with a convenient and effective BA tool.

In 2014, Zach et al. [4] improved the problems of deviation and outliers in the traditional BA algorithm and proposed a new BA algorithm based on robust estimation. Experiments show that the algorithm can better deal with outliers in the input data and improve the robustness and accuracy of BA.

Cao et al. proposed an incremental SfM method based on the parallel BA algorithm in 2020 [5], which achieves efficient 3D reconstruction and pose estimation by processing input data in parallel and optimizing camera pose and 3D points. Experimental results show that the algorithm has higher efficiency and precision than the traditional BA algorithm.

## 1.3 Goals and Challenges

It can be seen from the previous work that the implementation of the BA algorithm is often concentrated on the software level, and a lot of research has been done on robustness, sparse matrices, and parallelism. But CPU-based software lacks the high-efficiency advantages of hardware such as FPGA in computing. "pi-BA," [2] which is similar to our work, focuses on how to reduce repeated calculations by parallel computing on an FPGA. However, it ignores the simplification of the differential operations involved in the Jacobian matrix update of the BA algorithm, which can reduce hardware complexity and improve efficiency. This thesis is mainly aimed at designing a hardware accelerator for the JU and the CC based on the simplification of the differential operation.

# Chapter 2
# Methodology

Methodology refers to a set of systematic and organized plans and guidelines for researching, discussing, or analyzing problems, which play a vital role in in-depth understanding and effective problem-solving. The application of methodology can ensure the research process's logical coherence, precision, and robustness. A good methodology design can effectively provide a solid foundation for the next hardware implementation work.

In this project, we adopted a "top-down" strategy to plan the relevant work steps, as shown in Fig. 2.1. We divide the main hardware implementation tasks into three phases: high-level algorithm analysis, arithmetic circuit design, and system integration and evaluation.

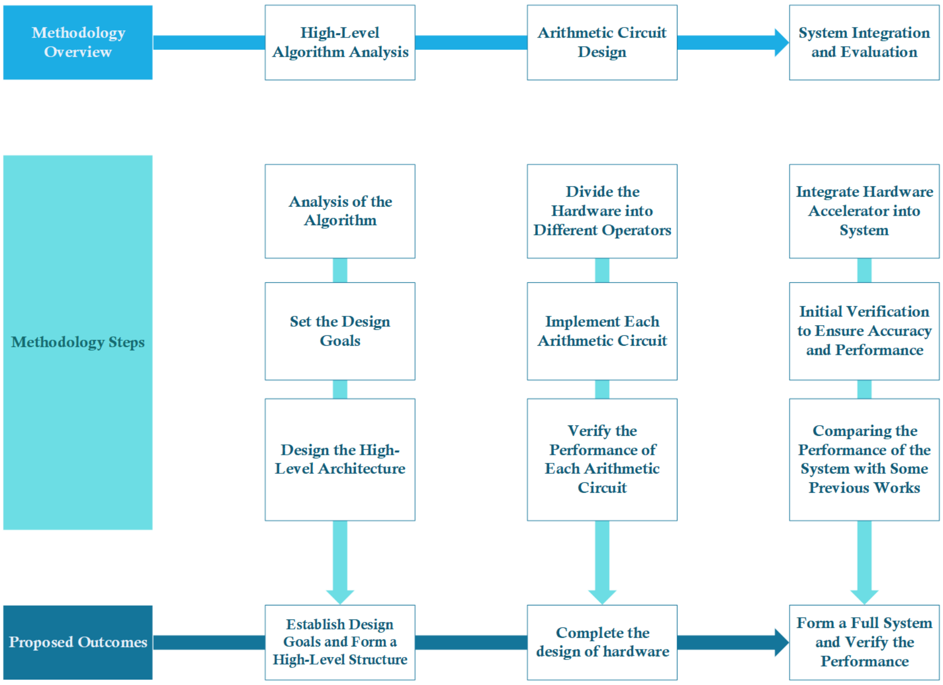

Methodology Overview
High-Level Algorithm Analysis
Arithmetic Circuit Design
System Integration and Evaluation
Methodology Steps
Analysis of the Algorithm
Set the Design Goals
Design the High-Level Architecture
Divide the Hardware into Different Operators
Implement Each Arithmetic Circuit
Verify the Performance of Each Arithmetic Circuit
Integrate Hardware Accelerator into System
Initial Verification to Ensure Accuracy and Performance
Comparing the Performance of the System with Some Previous Works
Proposed Outcomes
Establish Design Goals and Form a High-Level Structure
Complete the design of hardware
Form a Full System and Verify the Performance

**Figure 2.1:** "Top-down" Methodology Chart.

## 2.1 System Analysis and High-Level Architecture

### 2.1.1 Algorithm Analysis

As shown in Fig. 2.2, JU and CC are key steps in the BA algorithm, and their time-consuming nature has a significant impact on the performance of the entire system. This part usually occupies more than 60 percent of the entire BA algorithm’s running time [2]. Since this step is a differential calculation of the projection function for the three-dimensional points, the rotation matrix, and the translation vector. It involves a large number of matrix multiplication and accumulation operations, so it is very important to understand its computational complexity and parallelism characteristics.

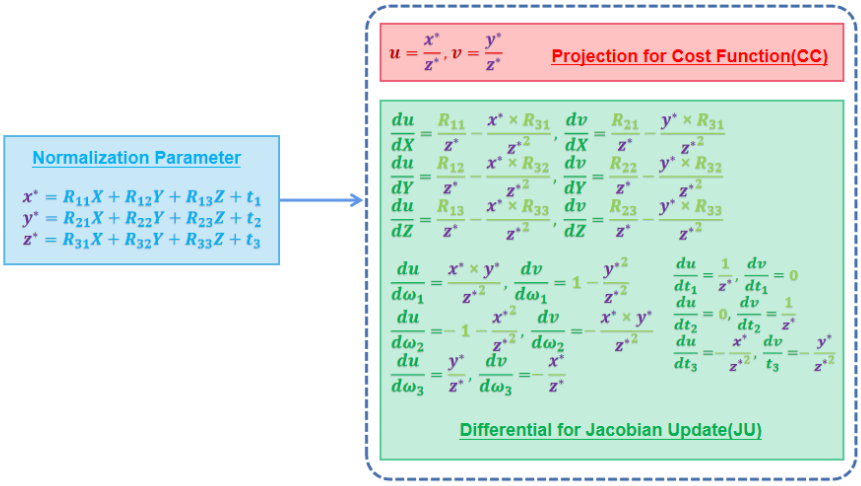

Normalization Parameter
$x^* = R_{11}X + R_{12}Y + R_{13}Z + t_1$
$y^* = R_{21}X + R_{22}Y + R_{23}Z + t_2$
$z^* = R_{31}X + R_{32}Y + R_{33}Z + t_3$
$u = \frac{x^*}{z^*}, v = \frac{y^*}{z^*}$
Projection for Cost Function(CC)
$\frac{du}{dX} = \frac{R_{11}}{z^*} - \frac{x^* \times R_{31}}{z^{*2}}, \frac{dv}{dX} = \frac{R_{21}}{z^*} - \frac{y^* \times R_{31}}{z^{*2}}$
$\frac{du}{dY} = \frac{R_{12}}{z^*} - \frac{x^* \times R_{32}}{z^{*2}}, \frac{dv}{dY} = \frac{R_{22}}{z^*} - \frac{y^* \times R_{32}}{z^{*2}}$
$\frac{du}{dZ} = \frac{R_{13}}{z^*} - \frac{x^* \times R_{33}}{z^{*2}}, \frac{dv}{dZ} = \frac{R_{23}}{z^*} - \frac{y^* \times R_{33}}{z^{*2}}$
$\frac{du}{d\omega_1} = \frac{x^* \times y^*}{z^{*2}}, \frac{dv}{d\omega_1} = 1 - \frac{y^{*2}}{z^{*2}}$
$\frac{du}{d\omega_2} = - 1 - \frac{x^{*2}}{z^{*2}}, \frac{dv}{d\omega_2} = - \frac{x^* \times y^*}{z^{*2}}$
$\frac{du}{d\omega_3} = \frac{y^*}{z^*}, \frac{dv}{d\omega_3} = - \frac{x^*}{z^*}$
$\frac{du}{dt_1} = \frac{1}{z^*}, \frac{dv}{dt_1} = 0$
$\frac{du}{dt_2} = 0, \frac{dv}{dt_2} = \frac{1}{z^*}$
$\frac{du}{dt_3} = - \frac{x^*}{z^{*2}}, \frac{dv}{t_3} = - \frac{y^*}{z^{*2}}$
Differential for Jacobian Update(JU)

**Figure 2.2:** Calculations Involved in JU and CC.

First, we studied the algorithm flow of the JU in detail, analyzed its mathematical operations, such as multiplication and addition, division to find the reciprocal, etc., and simplified its computational complexity. We explored the parallelism of the algorithm, in which "x*, y*, and z*" can perform computing tasks simultaneously at the first stage, providing normalization parameters for 18 differential formulas of each feature that can be processed in parallel in subsequent stages. At the same time, we separately extract the "1/z*" that exists in the differential operation and place it in a division module to complete the calculation, and then deliver it to subsequent processing to simplify the overall hardware complexity. These provide the possibility for us to design parallel hardware accelerators to the greatest extent and simplify the calculation process.

### 2.1.2 Requirements and Goals of Design

We further clarified the design needs and objectives of the hardware accelerator after defining the computational properties of the algorithm.

Firstly, performance is our primary design objective. We expect that the developed hardware accelerator will allow the JU and CC to be computed at a significantly faster rate while also increasing the overall performance of the BA algorithm while consuming less power. The hardware accelerator's main frequency will initially be set to 100MHz in order to accomplish this purpose. Our hardware accelerator is effective enough at this frequency level to speed up JU and CC without significantly increasing power usage.

In addition, we also need to carefully consider the hardware area. We expect that the hardware accelerator will be compact and easy to integrate into an embedded system. Our design has to utilize internal FPGA resources such as logic units (LUTs), flip-flops (FFs), and multipliers (DSPs) as minimally as possible because it will be implemented on the FPGA. The specific resource constraints will depend on the FPGA type.

### 2.1.3 High-Level Architecture Design Strategies

After clarifying the design requirements and goals, we began to design the high-level structure of the hardware accelerator. The initial, unmodified hardware structure is shown in Fig. 2.3.

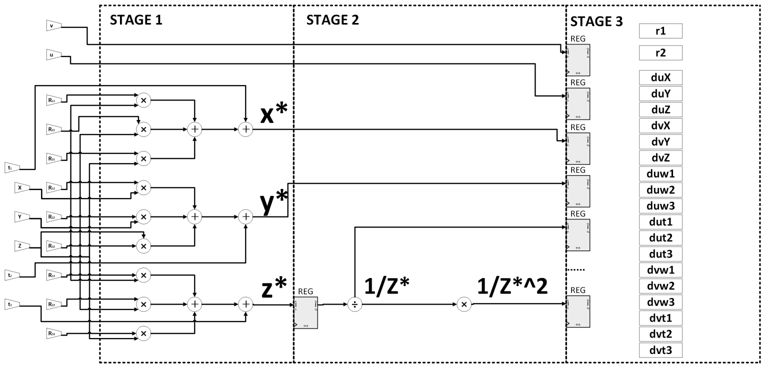


**Figure 2.3:** Initial Hardware Structure.

First, we determined the initial design of the hardware accelerator. In this design, we should make use of the parallelism of the algorithm as much as possible and distribute the computing tasks that can be executed in parallel to different hardware units. Second, we need to design control logic for the hardware accelerator to ensure that it behaves as expected. When designing the control logic, we should minimize the complexity of the control logic to reduce the design complexity and possible errors.

In general, our high-level architecture design strategy is to design an initial hardware accelerator with high performance, low power consumption, compact size, and flexibility based on the results of algorithm analysis and design requirements. This design strategy provides a clear direction for our subsequent specific design and modification.

## 2.2 Arithmetic Circuit Design and Optimization

### 2.2.1 Operator Division Strategy

In our hardware accelerator design, we focus on several main arithmetic circuits involved in the algorithm flow based on the JU and CC. These arithmetic circuits include adders, multipliers, and dividers, among others. Each operator corresponds to one or more steps in the algorithm flow, where multiplication and addition need to be designed to be executed in a single cycle as much as possible. This granular strategy not only makes our design clearer and easier to optimize but also helps to improve performance and save resources.

### 2.2.2 Arithmetic Circuit Design and Optimization Strategies

We must properly compare performance and resource utilization for the implementation of each arithmetic circuit. Before we start synthesizing combinatorial logic into hardware circuits of LUTs cascaded with one another in accordance with Boolean expressions within an FPGA, we must first compare the implementation of combinatorial logic (such as "carry look-ahead adder" and "Booth multiplier"). To carry out addition and multiplication, resources inside the FPGA such as "CARRY4" and "DSPs" are often used to implement such operators.

Regarding the divider, it is a key component necessary to ensure "1/z*" is involved in all calculations in the differential operation. Implementing an effective divider, however, requires more thought than multiplication and addition do because of the challenge of implementing the division operation in hardware. Considerations like operational accuracy, speed, and resource usage must be made when implementing a divider on hardware. We suggest the "Shift-Sub" and "Newton-Raphson method" as two divider implementations to achieve this. They are all suitable for embedded systems due to their low complexity and compact area, but some operations may result in accuracy losses. In order to select the implementation with the best performance and efficiency, we will compare the running speed and hardware resource usage of the two implementations.

## 2.3 System Integration and Evaluation

### 2.3.1 System Integration

We will integrate these operators into a full hardware accelerator after the arithmetic circuit and each module have been built and verified. In order for the system to work more effectively as expected during this process, we first defined the ASMD of the controller. At the same time, we made some performance and area optimizations to create a more efficient and compact hardware accelerator. We will design a module with ROM inside to input data in parallel after the controller and hardware accelerator are ready. After that, carry out corresponding tasks, such as system simulation.

### 2.3.2 Performance Evaluation and Accuracy Comparison

In this step, we will conduct a performance test of the calculation accuracy of the hardware accelerator and compare the speedup ratio of the software runtime. At the same time, we can also estimate the energy consumption through the running time of the hardware accelerator and the estimated power consumption to compare the advantages of hardware accelerators in terms of energy efficiency with the software implementation of JU and CC.

# Chapter 3
# Theory

This chapter aims to introduce the fundamental principles and relevant theories of BA while highlighting the significance of JU and CC within the algorithm. The objective is to provide a comprehensive understanding of BA and emphasize the crucial roles played by JU and CC.

## 3.1 Fundamentals of Bundle Adjustments

### 3.1.1 Introduction of Bundle adjustment

A beam of light originates from a point in three-dimensional space and refers to a projection of light onto an image plane. BA is the process of reconstructing the geometric structure of a three-dimensional scene using image data observed from multiple angles, with the goal of optimizing the estimation model. At its core, BA involves an optimization model that seeks to minimize reprojection errors as much as possible [6].

Reprojection errors are the differences between the projections of real 3D points on the image and the re-projections based on our calculations. Reprojection involves two stages of projection. The first projection refers to projecting 3D points onto the image using the camera during photography. We utilize these images to extract feature points. By employing these feature points, we can estimate the position of the 3D points through our calculations. Finally, we use the calculated coordinates of the 3D points and the camera matrix to perform a second projection, known as reprojection [7].

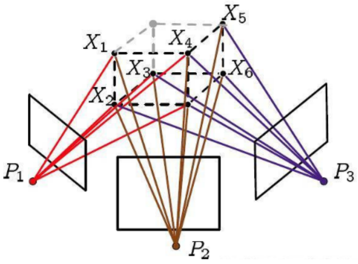


**Figure 3.1:** BA Image Model

Fig. 3.1 [8] illustrates the concept of BA. In the figure, $P_1$, $P_2$, and $P_3$ represent the poses of three cameras. $X_1$, $X_2$, $X_3$, $X_4$, $X_5$, and $X_6$ denote six 3D points that are visible and captured in the images taken by the cameras at poses $P_1$, $P_2$, and $P_3$. Minimizing the sum of these differences becomes necessary to obtain the optimal camera pose parameters and the coordinates of the 3D points.

### 3.1.2 Mathematical model of Bundle Adjustment

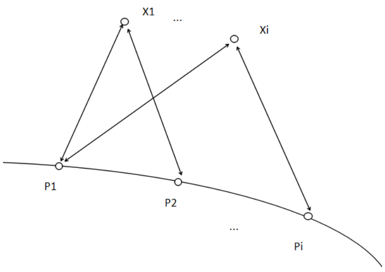


**Figure 3.2:** BA Graph Model

BA is a graph optimization model [9], [10] which is shown in Fig. 3.2. The nodes of the graph model are composed of camera $P_j$ and 3D points $X_i$. If the camera $P_j$ can see the point $X_j$, 2 nodes $X_j$ and $P_j$ can be connected. That means we can write an observation function. We predict the position of the camera based on the observation equation [11].

To get reprojection errors, we use the following observation function as follows Eq. 3.1 [12][13].

$$\begin{bmatrix}\mathbf{U}\\\mathbf{V}\\1\end{bmatrix} = \mathbf{KTP} = \mathbf{K} \times [\mathbf{R}|\mathbf{t}] \times \begin{bmatrix}\mathbf{X}\\\mathbf{Y}\\\mathbf{Z}\\1\end{bmatrix} \tag{3.1}$$

$[\mathbf{UV}]$ is the projection position in the image of 3D points under a camera model, which is shown in the image. $[\mathbf{XYZ}]$ is the 3D Points position matrix in the real world. $\mathbf{T}$ is a 3 × 4 transition matrix containing a 3 × 3 rotation matrix $\mathbf{R}$ and a 3 × 1 translation matrix. $\mathbf{K}$ is the camera parameter model, which includes distortion and focal length information. It is the camera’s internal parameter and is fixed for a camera.

For the projection of any 3D point Xi in the image, BA needs to minimize the sum of errors between the camera shooting point and the reprojected point. Assuming a is the position of point $X_j$ captured by the camera at $P_j$ in the image. b is the reprojection calculated by Eq. 3.2. We can get a reprojection error by Eq. 3.2 [7].

$$Err(i,j) = \mathbf{a}(i,j) - \mathbf{b}(i,j) \tag{3.2}$$

The purpose of BA is to minimize the sum of reprojection errors. Using the least squares method, we obtain Eq. 3.3 [4][7].

$$\min_{Ri,ti,Xj} \sum_{i,j} (\sigma(i,j) \left\| Err(i,j) \right\|)^2 \tag{3.3}$$

In Eq. 3.3 if $X_i$ is shown in $P_j$ image, $\sigma$ equals 1, otherwise, it equals 0.

## 3.2 Nonlinear optimization

### 3.2.1 Solving Nonlinear Equations

To solve the least squares problem Eq. 3.3, we use iterative methods. Finding the minimum value of a function can be transformed into finding a derivative that equals 0. In this situation, first, give an initial value, and then add a small increment to the independent variable each time to calculate the minimum value. After multiple iterations, if the increment is small enough, the minimum value is found and the linear equation is solved.

One convenient method for solving function increments is to perform Taylor expansion on the function, which is shown in Eq. 3.4 [14].

$$\left\| f(x + \triangle x) \right\|^2 = f(x)^2 + \mathbf{J}(\mathbf{x}) \triangle x + \frac{1}{2} \triangle x \mathbf{H} \triangle x \tag{3.4}$$

J is the Jacobian matrix of the function, which is the first derivative of the function. H is the Hessian matrix, the second derivative of the function. Through Eq. 3.4 , the solution of X can be obtained as Eq. 3.5 . This method is known as the gradient descent method.

$$\triangle x = -\mathbf{J^T}(\mathbf{x}) \tag{3.5}$$

Using negative values of gradients for descent is the simplest method Eq. 3.5, but convergence is slow. Therefore, the Newton method was proposed to solve this problem. When we consider the second derivative to solve x, we get Eq. 3.6 [7].

$$\mathbf{H} \triangle x = -\mathbf{J^T}(\mathbf{x}) \tag{3.6}$$

This method is called Newton's method. However, the calculation cost of the Hessian matrix is high. To solve this problem, the Newton-Gaussian method was proposed. Gaussian Newton method uses $\mathbf{J} \times \mathbf{J^T}$ transposition instead of the Hessian matrix. However, there is no guarantee that every iteration error will decrease. The LM algorithm is generally used to solve $\triangle$ x in slam, it can solve the above problems very well.

### 3.2.2 Levenberg Marquardt (LM) algorithm

The LM algorithm is an improvement of the above algorithm. LM is a trust domain-based optimization algorithm that limits the range of $\triangle$ x per iteration. Its calculation method is Eq. 3.7 [15].

$$(\mathbf{J^T J} + \mu \mathbf{I}) \triangle \mathbf{x} = -\mathbf{J^T}(\mathbf{x}) \tag{3.7}$$

Compared with the Gaussian Newton method and the Steepest Descent method, the LM algorithm combines the advantages of both. When $\mu$ is relatively large, Eq. 3.7 approaches the steepest descent algorithm. When $\mu$ is small, Eq. 3.7 approaches the Gaussian Newton method.

The LM algorithm ensures that every iteration is descending and can converge quickly. Besides, by solving the H-matrix with rank dissatisfaction or non-positive definite, the LM algorithm is widely used in SLAM [16].

## 3.3 Implementation of LM Algorithm

To perform nonlinear optimization using Eq. 3.7, it is necessary to first calculate the Jacobian matrix "J". The Jacobian matrix consists of first-order partial derivatives, which reflect the optimal linear approximation of the equation. For a sampling point, the first-order partial derivative mentioned in Eq. 3.8 can form a matrix with 2 rows and 9 columns Eq. 3.8 [17].

$$\begin{bmatrix} \frac{\mathbf{dU}}{\mathbf{dX}} & \frac{\mathbf{dU}}{\mathbf{dY}} & \frac{\mathbf{dU}}{\mathbf{dZ}} & \frac{\mathbf{dU}}{\mathbf{d}\omega\mathbf{1}} & \frac{\mathbf{dU}}{\mathbf{d}\omega\mathbf{2}} & \frac{\mathbf{dU}}{\mathbf{d}\omega\mathbf{3}} & \frac{\mathbf{dU}}{\mathbf{dt1}} & \frac{\mathbf{dU}}{\mathbf{dt2}} & \frac{\mathbf{dU}}{\mathbf{dt3}} \\ \\ \frac{\mathbf{dV}}{\mathbf{dX}} & \frac{\mathbf{dV}}{\mathbf{dY}} & \frac{\mathbf{dV}}{\mathbf{dZ}} & \frac{\mathbf{dV}}{\mathbf{d}\omega\mathbf{1}} & \frac{\mathbf{dV}}{\mathbf{d}\omega\mathbf{2}} & \frac{\mathbf{dV}}{\mathbf{d}\omega\mathbf{3}} & \frac{\mathbf{dV}}{\mathbf{dt1}} & \frac{\mathbf{dV}}{\mathbf{dt2}} & \frac{\mathbf{dV}}{\mathbf{dt3}} \end{bmatrix} \tag{3.8}$$

Eq. 3.8 is the Jacobian martix mentioned in Eq. 3.7. The $[\mathbf{U}, \mathbf{V}]$ is the projection position in the image of 3D points under a camera model Eq. 3.1. They represent the horizontal and vertical distances in the frame, respectively. 9 parameters, including rotation, translation, and 3D points in space, determine the position of feature points in the image which is $[\mathbf{U}, \mathbf{V}]$. For a characteristic point, we need to calculate the derivatives of $\mathbf{U}$ and $\mathbf{V}$ respectively. So we need 18 partial derivatives to solve, shown in Eq.3.8. $[\mathbf{X}, \mathbf{Y}, \mathbf{Z}]$ are 3D point position which is derived from Eq. 3.1. $\omega$ are the 3 Lie algebras of the rotation matrix, and t are the 3 parameters of the translation matrix. We will discuss this problem in detail in section 3.3.2.

### 3.3.1 Partial derivative of 3D point

Expanding Eq. 3.1, we obtain Eq. 3.9.

$$\begin{bmatrix} \mathbf{U} \\ \mathbf{V} \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{R_{11}} & \mathbf{R_{12}} & \mathbf{R_{13}} \\ \mathbf{R_{21}} & \mathbf{R_{22}} & \mathbf{R_{23}} \\ \mathbf{R_{31}} & \mathbf{R_{32}} & \mathbf{R33} \end{bmatrix} \times \begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \\ \mathbf{1} \end{bmatrix} + \begin{bmatrix} \mathbf{t1} \\ \mathbf{t2} \\ \mathbf{t3} \end{bmatrix} \tag{3.9}$$

The above formula can also be written as Eq. 3.10. As a constant, K does not affect the result of the derivative and there is no need to calculate in the Jacobian matrix.

$$\begin{cases} \mathbf{X}^* = \mathbf{R_{11}X} + \mathbf{R_{12}Y} + \mathbf{R_{13}Z} + \mathbf{t1} \\ \mathbf{Y}^* = \mathbf{R_{21}X} + \mathbf{R_{22}Y} + \mathbf{R_{23}Z} + \mathbf{t1} \\ \mathbf{Z}^* = \mathbf{R_{31}X} + \mathbf{R_{32}Y} + \mathbf{R_{33}Z} + \mathbf{t3} \end{cases} \tag{3.10}$$

By normalizing Eq. 3.10, we can obtain Eq. 3.11 [18] as

$$\begin{cases}\mathbf{U} = \frac{\mathbf{X}^*}{\mathbf{Z}^*}\\ \mathbf{V} = \frac{\mathbf{Y}^*}{\mathbf{Z}^*}\end{cases} \tag{3.11}$$

Combining Eq. 3.10 and Eq. 3.11, using the Chain Derivation Rule we get Eq. 3.12 [18] .

$$\begin{cases}\frac{du}{dX} = \frac{R_{11}}{z*} - \frac{R_{31} \times X^*}{Z*^2}\\ \frac{du}{dY} = \frac{R_{12}}{Z^*} - \frac{R_{32} \times X^*}{Z^{*2}}\\ \frac{du}{dZ} = \frac{R_{13}}{Z^*} - \frac{R_{33} \times X^*}{Z^{*2}}\\ \frac{dv}{dX} = \frac{R_{21}}{Z^*} - \frac{R_{31} \times Y^*}{Z^{*2}}\\ \frac{dv}{dY} = \frac{R_{22}}{Z^*} - \frac{R_{32} \times Y^*}{Z^{*2}}\\ \frac{dv}{dZ} = \frac{R_{23}}{Z^*} - \frac{R_{33} \times Y^*}{Z^{*2}}\end{cases} \tag{3.12}$$

Eq. 3.12 obtains the partial derivative of 3D points in space relative to the projected coordinate in BA.

### 3.3.2 The partial derivative of the rotation matrix and translation matrix

In order to describe the camera position, it is necessary to know the camera's rotation and translation parameters to construct motion equations. However, the derivation of rotation parameters is relatively not as simple due to the nonlinearity of the rotation matrix. Therefore, firstly, we need to know how to describe rotation. For a vector in a given coordinate system, we can define the base vector of this coordinate system as [$e_1$,$e_2$,$e_3$], they are orthogonal to each other, and then we can get any vector in this coordinate system like Eq. 3.13 [19][7].

$$a = [\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}] \times \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\\ \mathbf{z}\end{bmatrix} = \mathbf{e_1} \times \mathbf{x} + \mathbf{e_2} \times \mathbf{y} + \mathbf{e_3} \times \mathbf{z} \tag{3.13}$$

For the same point in space, vectors in two different coordinate systems can be written as Eq. 3.14.

$$a = [\mathbf{e_1 1}, \mathbf{e_1 2}, \mathbf{e_1 3}] \times \begin{bmatrix}\mathbf{x_1}\\ \mathbf{y_1}\\ \mathbf{z_1}\end{bmatrix} = [\mathbf{e_2 1}, \mathbf{e_2 2}, \mathbf{e_2 3}] \times \begin{bmatrix}\mathbf{x_2}\\ \mathbf{y_2}\\ \mathbf{z_2}\end{bmatrix} \tag{3.14}$$

Multiplying both sides of the equation by $\begin{bmatrix}\mathbf{e_{11}^T}\\ \mathbf{e_{12}^T}\\ \mathbf{e_{13}^T}\end{bmatrix}$ to the left,we get Eq. 3.15.

$$\begin{bmatrix}\mathbf{x_1}\\ \mathbf{y_1}\\ \mathbf{z_1}\end{bmatrix} = \begin{bmatrix}\mathbf{e_{11}^T e_{21}} & \mathbf{e_{11}^T e_{22}} & \mathbf{e_{11}^T e_{23}}\\ \mathbf{e_{12}^T e_{21}} & \mathbf{e_{12}^T e_{22}} & \mathbf{e_{12}^T e_{23}}\\ \mathbf{e_{13}^T e_{21}} & \mathbf{e_{13}^T e_{22}} & \mathbf{e_{13}^T e_{23}}\end{bmatrix} \times \begin{bmatrix}\mathbf{x_2}\\ \mathbf{y_2}\\ \mathbf{z_2}\end{bmatrix} \tag{3.15}$$

The 3 × 3 matrix in the middle is the rotation matrix $\mathbf{R}$ in Eq. 3.1. The rotation matrix is an orthogonal matrix whose determinant is 1. At the same time, the orthogonal matrix with determinant 1 is also a rotation matrix. The set of rotation matrices can be defined as an orthogonal group. We use a rotation matrix to describe camera rotation in SLAM.

Lie group is defined as a group with continuous properties. The group of rotation matrix is a Lie group. The rotation matrix is not closed to addition, and the increment of the derivative of the rotation matrix cannot be directly added to the old matrix to update the matrix, so a way to calculate the partial differential of rotation is needed. Therefore, Lie algebra was proposed to solve this problem.

Lie algebra is the tangent space at the identity element of the Lie group, which describes the Local property of the Lie group. Each Lie group has a corresponding Lie algebra. For rotation matrix, any rotation matrix $\mathbf{R}$ have this characteristic $\mathbf{R} \times \mathbf{R^T} = \mathbf{I}$. Assuming that the rotation matrix describes the continuous motion in space, we can get Eq. 3.16 [7]

$$\mathbf{R(t)R(t)^T} = \mathbf{I} \tag{3.16}$$

Taking the derivative of time at both sides of the equation Eq. 3.17 [7].

$$\dot{\mathbf{R(t)}}\mathbf{R(t)^T} + \mathbf{R(t)}\dot{\mathbf{R(t)}}\mathbf{^T} = 0 \tag{3.17}$$

Multiply $\mathbf{R}$ to the right of the Eq. 3.18

$$\dot{\mathbf{R(t)}}\mathbf{R(t)^T} = -(\dot{\mathbf{R(t)}}\mathbf{R(t)^T})^\mathbf{T} \tag{3.18}$$

$\dot{\mathbf{R(t)}}\mathbf{R(t)^T}$ is the Lie algebra of the rotation matrix group and can be written as $\phi(t)$. It is easy to see from Eq. 3.18 that $\phi(t)$ is an anti-symmetric matrix because according to the properties of the anti-symmetric matrix, $\phi$ can be written as Eq. 3.19 [20]

$$\phi = \begin{bmatrix} \mathbf{0} & -\omega 3 & \omega 2 \\ \omega 3 & \mathbf{0} & -\omega 1 \\ -\omega 2 & \omega 1 & \mathbf{0} \end{bmatrix} \tag{3.19}$$

Multiply both sides of the equation to the right by a $\mathbf{R(t)}$ Eq. 3.20.

$$\dot{\mathbf{R(t)}} = \phi(\mathbf{t}) \times \mathbf{R(t)^T} \tag{3.20}$$

Eq. 3.20 shows that the derivation of the rotation matrix can be obtained by left multiplying a Lie algebra. Solving the partial differential equation Eq. 3.20. yields Eq. 3.21 [21].

$$\mathbf{R(t)} = \mathbf{exp}(\phi(\mathbf{t})) \tag{3.21}$$

This formula shows that for any rotation matrix $\mathbf{R}$, there is a determined $\phi$ corresponding to it, which is called lie algebra. Lie algebra describes the derivative relationship of Lie groups.

For a rotating matrix group, finding the tangent plane of a point on the group is the process of finding the partial derivative. Assuming the rotation matrix of spatial point p is represented as $\mathbf{Rp}$. To obtain the derivative after rotation, it can be written as Eq. 3.22.

$$\frac{\partial \mathbf{Rp}}{\partial \mathbf{R}} = \frac{\partial \mathbf{exp}(\phi)\mathbf{p}}{\partial \phi} \tag{3.22}$$

Since the rotation matrix group is not close to addition, it is necessary to find the derivative in another way. we use perturbation models to solve this problem. A small perturbation is made to the rotation matrix R, assuming that the perturbed Lie algebra is $\varphi$, and the derivative of $\varphi$ is obtained Eq. 3.23 [7].

$$\begin{aligned}\frac{\partial Rp}{\partial\varphi} &= \lim_{\varphi\to 0}\frac{exp(\varphi)exp(\phi)p-exp(\phi)p}{\varphi}\\ &\approx \lim_{\varphi\to 0}\frac{(1+\varphi)exp(\phi)p-exp(\phi)p}{\varphi}\\ &= \lim_{\varphi\to 0}\frac{\varphi Rp}{\varphi}\\ &= -Rp\end{aligned} \tag{3.23}$$

The transition matrices are also a Lie group, which can be written as Eq. 3.24 .

$$T = \begin{bmatrix}\mathbf{R} & \mathbf{t}\\ \mathbf{0} & \mathbf{1}\end{bmatrix} \tag{3.24}$$

**T** describes the transitions of the camera. The Lie algebra of the transition matrix is Eq. 3.25. $\rho$ is the Lie algebra of a translation matrix.

$$\xi = \begin{bmatrix}\phi & \rho\\ \mathbf{0} & \mathbf{0}\end{bmatrix} \tag{3.25}$$

Similar to Eq. 3.23 , we can obtain the derivative of the transitions matrix Eq. 3.26 [18][7] as

$$\begin{aligned}J1 = \frac{\partial Tp}{\triangle\xi} &= \lim_{\xi\to 0}\frac{exp(\triangle\xi)exp(\xi)p-exp(\xi)p}{\triangle\xi}\\ &\approx \lim_{\xi\to 0}\frac{(1+\triangle\xi)exp(\xi)p-exp(\xi)p}{\triangle\xi}\\ &= \lim_{\xi\to 0}\frac{\triangle\xi exp(\xi)p}{\triangle\xi}\\ &= \lim_{\xi\to 0}\frac{\begin{bmatrix}\phi & \rho\\ 0 & 0\end{bmatrix}\begin{bmatrix}Rt+p\\ 1\end{bmatrix}}{\triangle\xi}\\ &= \begin{bmatrix}I & -(Rp_+t)\\ 0 & 0\end{bmatrix}\\ &= \begin{bmatrix}1 & 0 & 0 & 0 & Z* & -Y*\\ 0 & 1 & 0 & -Z* & 0 & X*\\ 0 & 0 & 1 & Y* & -X* & 0\end{bmatrix}\end{aligned} \tag{3.26}$$

The derivative of Eq. 3.26 for the rotation matrix and translation matrix is Eq. 3.27.

$$J2 = \begin{bmatrix}\frac{1}{Z*} & 0 & -\frac{X*}{Z*^2}\\ 0 & \frac{1}{Z*} & -\frac{Y*}{Z*^2}\end{bmatrix} \tag{3.27}$$

According to the chain derivative rule, the derivative of a feature point can be calculated as Eq. 3.28.

$$J1J2 = \begin{bmatrix}\frac{1}{Z*} & 0 & -\frac{X*}{Z*^2} & -\frac{X*Y*}{Z*^2} & 1+\frac{X*^2}{Z*^2} & -\frac{Y*}{Z*}\\ 0 & \frac{1}{Z*} & -\frac{Y*}{Z*^2} & -(1+\frac{Y*^2}{Z*^2}) & -\frac{X*Y*}{Z*^2} & -\frac{X*}{Z*}\end{bmatrix} \tag{3.28}$$

At this point, we have derived the Jacobian matrix Eq. 3.29 and Eq. 3.11, and the structure is shown in Eq. 3.8.

$$\left\{\begin{array}{c}\frac{du}{dt1}=\frac{1}{Z^*}\\ \frac{du}{dt2}=0\\ \frac{du}{dt3}=\frac{X^*}{Z^{*2}}\\ \frac{dv}{dt1}=0\\ \frac{dv}{dt2}=\frac{1}{Z^*}\\ \frac{dv}{dt3}=-\frac{Y^*}{Z^{*2}}\\ \frac{du}{d\omega1}=-\frac{X^*Y^*}{Z^{*2}}\\ \frac{du}{d\omega2}=1+\frac{X^{*2}}{Z^{*2}}\\ \frac{du}{d\omega3}=-\frac{Y^*}{Z^*}\\ \frac{dv}{d\omega1}=-(1+\frac{Y^{*2}}{Z^{*2}})\\ \frac{dv}{d\omega2}=-\frac{X^*Y^*}{Z^{*2}}\\ \frac{dv}{d\omega3}=-\frac{X^*}{Z^*}\end{array}\right. \tag{3.29}$$

## 3.4 Software implementation of BA using MATLAB

### 3.4.1 Software Implementation of BA

Based on the above theory, we designed MATLAB code to simulate and implement the BA algorithm. Firstly, according to Eq. 3.1, it was clarified that **K**, Transition matrix, and Position matrix were used as inputs to the function. Using two photos taken with the same camera, after re-projection, we obtained 1465 feature points that can be seen in both images Fig. 3.4(b). Therefore, the form of the Jacobian matrix can be shown as Eq. 3.30.

$$\begin{bmatrix} \frac{dUV1_{i1}}{dP_{i1}} & 0 & ... & ... & 0 & \frac{dUV1_{i1}}{dT_{i1}} & 0 \\ 0 & \frac{dUV1_{i2}}{dP_{i2}} & 0 & ... & 0 & \frac{dUV1_{i2}}{dT_{i2}} & 0 \\ 0 & 0 & ... & ... & ... & ... & \\ 0 & 0 & 0 & ... & \frac{dUV1_{in}}{dP_{in}} & \frac{dUV1_{in}}{dT_{in}} & 0 \\ \frac{dUV21_{i1}}{dP1_{i1}} & 0 & ... & ... & 0 & 0 & \frac{dUV21_{i1}}{dT1_{i1}} \\ 0 & \frac{dUV21_{i2}}{dP_{i2}} & 0 & ... & 0 & 0 & \frac{dUV2_{i2}}{dT_{i2}} \\ 0 & 0 & ... & ... & ... & 0 & ... \\ 0 & 0 & 0 & ... & \frac{dUV2_{in}}{dP_{in}} & 0 & \frac{dUV2_{in}}{dT_{in}} \end{bmatrix} \tag{3.30}$$

In Eq. 3.30, UV1 and UV2 are the reprojection coordinates of the two images, respectively. $\frac{dUV}{dP}$ is a 2 × 3 matrix containing the partial derivative of X, Y, and Z to UV (Eq. 3.31).

$$\frac{dUV}{dP} = \begin{bmatrix} \frac{\mathbf{dU}}{\mathbf{dX}} & \frac{\mathbf{dU}}{\mathbf{dY}} & \frac{\mathbf{dU}}{\mathbf{dZ}} \\ \\ \frac{\mathbf{dV}}{\mathbf{dX}} & \frac{\mathbf{dV}}{\mathbf{dY}} & \frac{\mathbf{dV}}{\mathbf{dZ}} \end{bmatrix} \tag{3.31}$$

$\frac{dUV}{dT}$ is a 2 × 6 matrix containing the partial derivative of T to UV (Eq. 3.32). From Eq. 3.30, it can be seen that each $\frac{dUV1}{dP}$ occupies one row and one column, while $\frac{dUV1}{dT}$ only occupies one column and is stacked together.

$$\frac{dUV1}{dT} = \begin{bmatrix} \frac{\mathbf{dU}}{\mathbf{d\omega 1}} & \frac{\mathbf{dU}}{\mathbf{d\omega 2}} & \frac{\mathbf{dU}}{\mathbf{d\omega 3}} & \frac{\mathbf{dU}}{\mathbf{dt1}} & \frac{\mathbf{dU}}{\mathbf{dt2}} & \frac{\mathbf{dU}}{\mathbf{dt3}} \\ \\ \frac{\mathbf{dV}}{\mathbf{d\omega 1}} & \frac{\mathbf{dV}}{\mathbf{d\omega 2}} & \frac{\mathbf{dV}}{\mathbf{d\omega 3}} & \frac{\mathbf{dV}}{\mathbf{dt1}} & \frac{\mathbf{dV}}{\mathbf{dt2}} & \frac{\mathbf{dV}}{\mathbf{dt3}} \end{bmatrix} \tag{3.32}$$

In this article, 1465 feature points were used, resulting in a Jacobian matrix of 1465 × 2 × 2 rows and 1465 × 3+6 × 2 columns.

We can see a large number of zeros in the Jacobian matrix, which means that there will be a lot of meaningless calculations when solving J × J in the subsequent LM algorithm calculations. Therefore, we learned that the idea of the sparse matrix was introduced to solve this problem. When storing, only the rows, columns, and contents of the data are recorded, which greatly accelerates the calculation speed and reduces the computational workload, making SLAM possible in scenarios that require rapid data processing, such as in the field of autonomous driving.

We will divide the BA algorithm into four functions to solve the problem. The first function is used to update the JU. Using the above formula, construct Jacobian Eq. 3.29 and Eq. 3.12 for the calculation of the Jacobian matrix. Then use the sparse matrix to store the matrix. The second function is to use the obtained Jacobian matrix to calculate the △x that needs to be corrected, using Eq. 3.7. The third function is to update the input. After each iteration, update the input **P** matrix and **T** matrix once to obtain better optimization results. It should be noted that for the update of the rotation matrix R, it is necessary to use Eq. 3.21 to add Lie algebra, instead of adding the rotation matrix directly. Finally, it is necessary to calculate the RMS error and determine that the calculation result converges at each iteration based on the results. The complete flowchart is shown in Fig. 3.3.

Fig. 3.4(a) and Fig. 3.4(b) are two frames taken from different angles. Due to the different shooting positions of the camera, rotation and translation of the camera occur.

In Fig. 3.4(a) and Fig. 3.4(b), the SIFT point [22] are feature points that can help confirm the pixel position in the image, drawing with a red circle in the diagram. The initial solution is the initial result of camera reprojection, which has not been optimized by the BA algorithm and is marked with a yellow cross on the image. The blue cross is re-projection optimized by BA, and we can see that it basically overlaps with the red cross in the figure.

When we zoom in on Fig. 3.4(b), we have obtained Fig. 3.5 and we can clearly see that the results after BA are almost identical to the results of the feature points. However, the results without BA have visible errors.

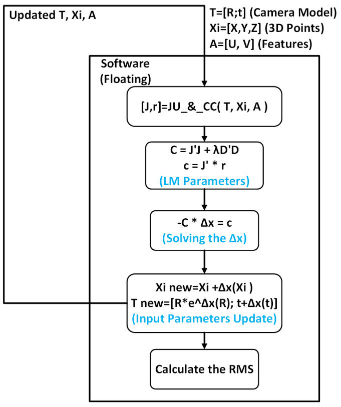
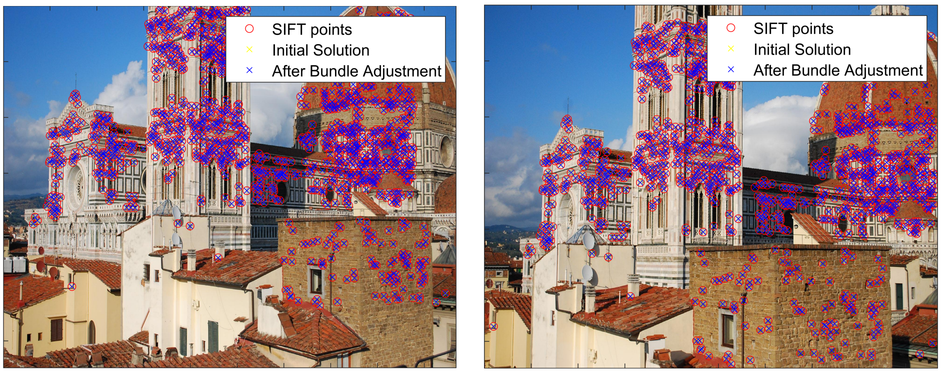


**Figure 3.3:** LM Flowchart



((a)) First Frame

((b)) Second Frame

**Figure 3.4:** Frames taken by camera at different positions

In Fig. 3.6, it can be clearly seen that the RMS error rapidly decreased from 66.75 in the first iteration to 1.47 in the fifth iteration, ultimately reaching 0.4906 in the tenth iteration. Generally speaking, the lower the RMS, the better. However, according to this paper in [3], the results optimized by RMS are generally below 2.

### 3.4.2 Fixed point analysis

When considering using FPGA as hardware to implement BA calculations, fixed-point numbers are used for low-power and high throughput implementations. However, fixed point numbers require a trade-off in data accuracy. Therefore, we designed code to verify the impact of converting data to fixed point numbers on BA results.

There are many intermediate data in the LM algorithm, and for each intermediate data, we perform data distribution analysis. Obtained Table. 3.1. According

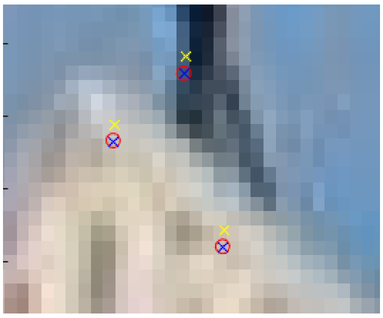


Figure 3.5: Zoom in plot of bundle adjustment

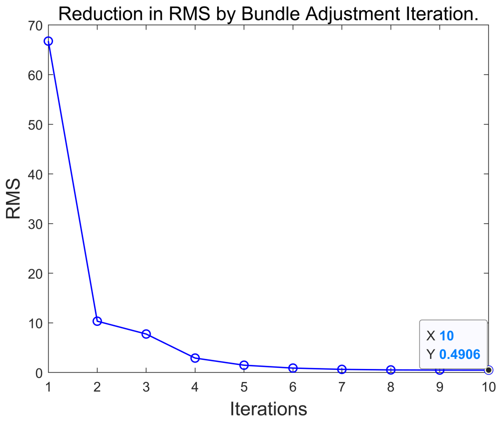


Figure 3.6: RMS variation in each iteration

to [2], the JU time accounts for approximately 60% of the total LM calculation time. Therefore, we have chosen to implement the JU and CC functions as hardware to accelerate computation speed. Using MATLAB to simulate fixed point number results with 1-bit signed numbers, and 3-bit fixed point integers (to prevent overflow).

**Table 3.1:** LM algorithm data range

| **Intermediate Data** | **Maximum Value** | **Minimum Value** |
|---|---|---|
| Jacobian Matrix | 2.3041 | -2.3041 |
| C(Left side of Eq. 3.7) | 4.1236e+03 | -2.3886e+03 |
| c(Right side of Eq. 3.7) | 0.0418 | -0.0796 |
| $\triangle x$ | 2.2857e-05 | -2.57301e-05 |

We simulate the RMS error for different fractional word lengths to achieve the expected result. In Fig. 3.7, after 10 times iterations, it can be seen that at 10-bit fractional word lengths, the RMS error is 28.37. As the bits of fractional word length increase, the RMS increases but decreases at 13 and 17 decimal places, which is caused by quantization errors. The RMS error is less than 2 when the fractional word length is 14, and the best result of 0.5328 is achieved at 16 fractional word length. In conclusion, our designed hardware will use 16 bits fractional, 3 bits integers, and 1 bits signed numbers.

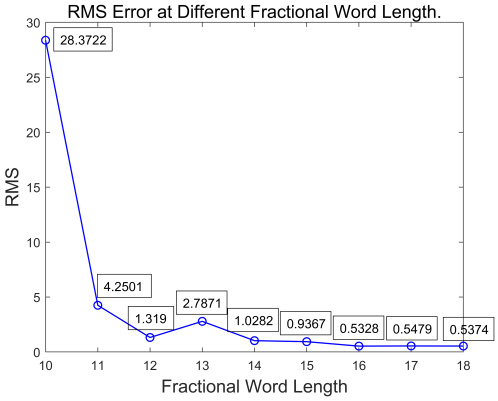


**Figure 3.7:** RMS error at different fractional word lengths of our fixed-point number.

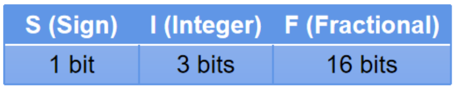
# Chapter 4

# Hardware Implementation and Modification

## 4.1 Hardware Implementation

This chapter will detail how to implement the hardware accelerator based on the "top-down" methodology presented in Chapter 2. The main task of our hardware accelerator is to execute the JU and CC parts to improve the computational efficiency of this operation. Since the initial high-level hardware structure block diagram is shown in Fig. 2.3 in Chapter 2, this chapter will first introduce the implementation and comparison of each arithmetic circuit of the hardware accelerator and then give the ASMD of the control logic. Finally, according to the overall modified circuit structure, the modification method of the hardware accelerator will be discussed in detail. In Chapter 3, we have tested that all data involved in the JU and CC parts are below 3, and the use of Q4.16 fixed-point numbers is sufficient to meet the accuracy requirements of the BA algorithm and avoid overflow during the operation. All data on the hardware is represented by 20-bit fixed-point numbers containing 16 fractional bits (we call such fixed-point numbers "Q4.16" for short), and the specific data format and actual values are shown in Fig. 4.1 and Eq. 4.1.

$$x = (-1)^S (I + \frac{F}{2^{16}}) \tag{4.1}$$



| S (Sign) | I (Integer) | F (Fractional) |
|---|---|---|
| 1 bit | 3 bits | 16 bits |

**Figure 4.1:** Data Format and Actual Value of "Q4.16".

### 4.1.1 Implementation of Arithmetic Circuits

In this section, we will detail the specific implementation of each arithmetic circuit. There are three operations we look at in detail: addition, multiplication, and division.

- Adders: We first considered the implementation of the "Custom Carry-Lookahead Adder", but in the FPGA, the combinational logic is implemented in a way that the LUTs is connected. In this way, compared with ASIC, which uses standard cells to implement logic circuits, FPGA will have a larger area (the LUTs of Xilinx FPGA is concentrated in each Slice logic block, which also includes many FFs, resulting in the additional area after synthesis). The structure of a 4-bit "Custom Carry-Lookahead Adder" and the logic gate diagram of the full adder is shown in Fig. 4.2 [23]. It can be seen that this implementation will significantly increase the critical path and area for addition with a large bit width. From the comprehensive comparison in Table. 4.1, it can be seen that the CARRY4 inside the FPGA is actually more compact and efficient than manually implementing the adder based on the logic gate connections. For this reason, we have adopted the adder design based on the parallel resource of CARRY4 on the FPGA. This design can effectively utilize the existing resources on the FPGA and increase the delay of addition operations.

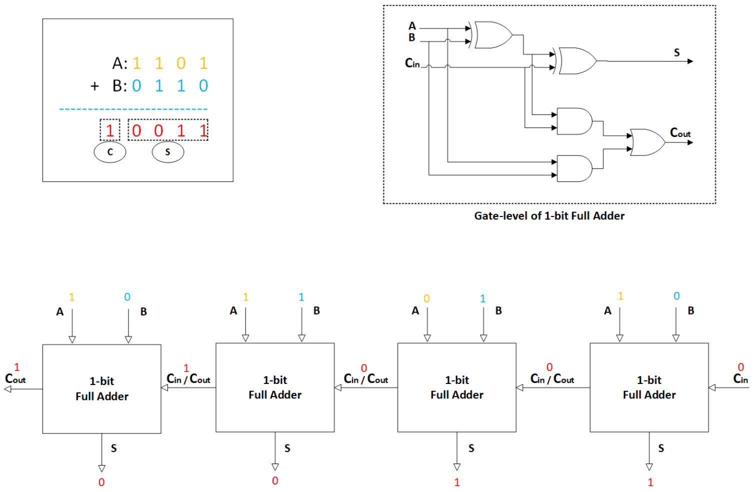


**Figure 4.2:** Structure Diagram of Carry Look-ahead Adder.

| 20-bits Adder Implementations | LUTs | Logic Delay |
|---|---|---|
| Custom Carry-Lookahead Adder | 80 | 6.07 ns |
| CARRY4 on FPGA | 20 | 1.84 ns |

**Table 4.1:** Timing and Resource Utilization Comparison

- Multipliers: we have also considered implementations such as Radix-4 Booth encoding, but through previous experience, we know that such a combinational circuit is not as efficient as the DSP multiplier on the FPGA chip (although the sequential multiplier is more compact, it also requires more clock cycles to complete the calculation, unlike the DSP multiplier that can complete the calculation within one cycle). For this reason, we decided to use the DSP parallel resources on the FPGA to realize the multiplier. This design can make full use of the existing resources on the FPGA to improve the speed of multiplication.

- Dividers: Compared with the adder or multiplier, the structure of the divider is more complex and usually requires multiple clock cycles to complete an operation. For this reason, it is necessary to minimize the use of the divider as much as possible. Fortunately, our algorithm only needs to calculate 1/z* once, and we just need to find an efficient and compact divider implementation.

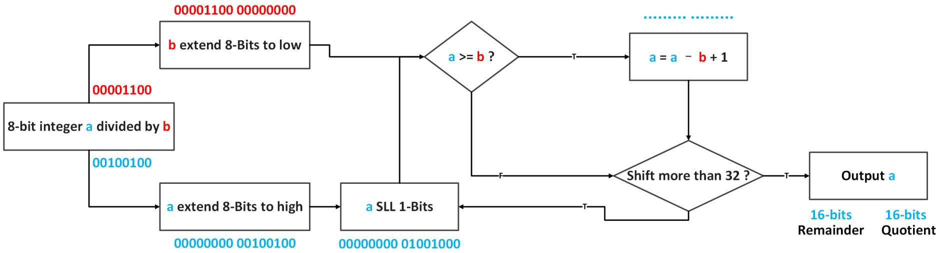


**Figure 4.3:** 8-bits "Shift-Sub" Divider Algorithm Flow.

We first chose the non-restorative divider design based on the "Shift-Sub" method [24]. The algorithm flow of the 8-bit implementation of the divider is shown in Fig. 4.3. The algorithm is based on a continuous heuristic and correction strategy: first, the divisor a and the dividend b are bit-extended, and then the divisor is shifted to the left by one bit at each step, and at the same time, it is judged whether the divisor a is greater than or equal to the dividend b. If the judgment is "true", the difference between a and b needs to be assigned to a; otherwise, continue to logically shift a to the left. The entire division requires 32 logical left shifts to complete. This "Shift-Sub" design has advantages in hardware implementation compactness and power consumption and is suitable for implementation in hardware accelerators, but its operation speed is generally slower than partial recovery or full recovery division algorithms. We use Verilog to describe the logical structure of the 20-bit "Shift-Sub" divider, and through functional verification, we ensure the correctness of the implementation of the divider and integrate it into the early hardware structure shown in Fig. 2.3. Table. 4.2 shows the timing and resource utilization report corresponding to the implementation of the divider and how many clock cycles are required for a division calculation. Although the timing and area of this implementation are in line with our expectations, the operation speed is very slow, and resources such as the DSP multiplier on the FPGA chip are not utilized as much as possible to bring higher efficiency. This makes us try to find new implementations.

Based on the Newton-Raphson method [25] we try to implement a more efficient divider. This is an iterative method for finding numerically approximate solutions to equations. It was first proposed by British scientist Isaac Newton in

| Logic Delay | LUTs | FFs | DSPs | Cycles per Execution |
|---|---|---|---|---|
| 2.888 ns | 207 | 168 | 0 | 86 |

**Table 4.2:** Performance and Area of 20-bits "Shift-Sub"

the 17th century and is widely used in numerical analysis and optimization problems. The basic idea is to approximate the roots of the equation by using the tangents of the equation. During iteration, we start with an initial guess, compute the derivative of the function at that point, and then pass the intersection of the tangent with the x-axis as the new guess. This process iterates until specified convergence criteria are met or a preset number of iterations is reached.

$$f(z) = \frac{1}{z} - x \tag{4.2}$$

$$z(n+1) = z(n) - \frac{f(z(n))}{f'(z(n))} \tag{4.3}$$

$$z(n+1) = z(n)(2 - xz(n)) \tag{4.4}$$

According to 1/z*, we can construct the functional Eq. 4.2 and bring it into the Newton iteration Eq. 4.3, and the simplified formula Eq. 4.4 is our iterative formula. Newton's iteration method is widely used in mathematics and engineering, especially for solving nonlinear equations and optimization problems. It is characterized by fast convergence speed and high accuracy, but there may be cases where iterations diverge or converge to the wrong root. Therefore, I need to perform the Newton method in different iterations on a set of Q4.16 fixed-point z* on MATLAB. Then it was compared with the result of the double floating-point calculation 1/z* to ensure that there was no loss of precision. The comparison results are shown in Fig . 4.4. It can be seen that 8 iterations inside the divider can fully meet the accuracy requirements.

After designing the 20-bit Newton divider through Verilog, we ensured the correctness of the implementation of the divider through functional verification and integrated it into the optimized hardware structure. Table. 4.3 shows the timing and resource utilization report corresponding to the implementation of the divider and how many clock cycles are required for a division calculation. It is obvious that the divider not only operates faster but also utilizes more "DSPs" on-chip resources, which can significantly improve the overall efficiency of the hardware.

| Logic Delay | LUTs | FFs | DSPs | Cycles per Execution |
|---|---|---|---|---|
| 4.077 ns | 207 | 28 | 4 | 19 |

**Table 4.3:** Performance and Area of 20-bits "Newton"

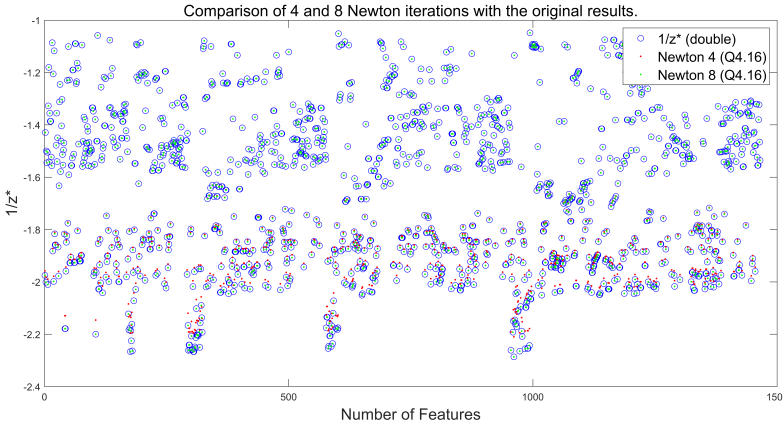


**Figure 4.4:** Accuracy Comparison of Different Newton Iterations.

### 4.1.2 System Integration Preparation

After the arithmetic circuit is implemented, we need to integrate all the modules together to form a complete hardware accelerator. We first designed the interface of each module to ensure the correct connection between them. We have adopted a parametric design method, and the data bit width, address, and number of feature points of each module can be modified arbitrarily, and then these modules are connected through interfaces. This design method not only makes the design and testing of each module easier but also facilitates the subsequent modification and expansion of the hardware accelerator.

We built a controller for the overall hardware according to the ASMD shown in Fig. 4.5 and performed system-level simulations using the Vivado design suite after system integration. The simulation results show that our hardware accelerator can correctly execute the JU and CC parts, and the speed is significantly improved compared with the software implementation. The detailed results comparison report and the whole structure after system integration will be described in Chapter 5.

In general, we have successfully implemented a hardware accelerator that can effectively accelerate the JU and CC parts, and the optimized circuit structure is shown in Fig. 4.6. In the following sections, we will introduce how we optimize the hardware accelerator to improve its performance and on-chip resource utilization.

## 4.2 Hardware Modification

After successfully implementing the hardware accelerator, we further modified it to improve performance and reduce area. This chapter will detail these optimization methods. Table. 4.3 shows the performance and resource utilization report after passing Vivado's "Implementation" according to the original hardware architecture shown in Fig 2.3 (including the "Shift-Sub" divider inside).

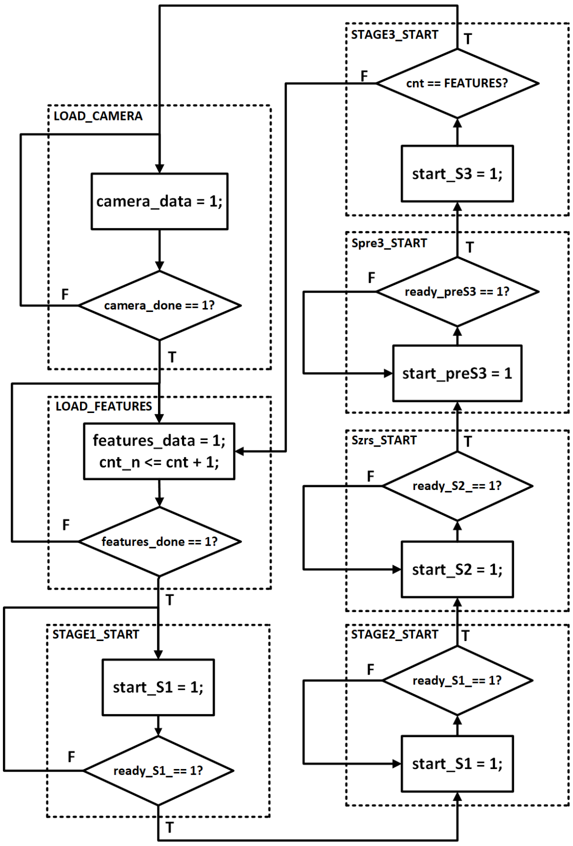


**Figure 4.5:** ASMD of our controller.

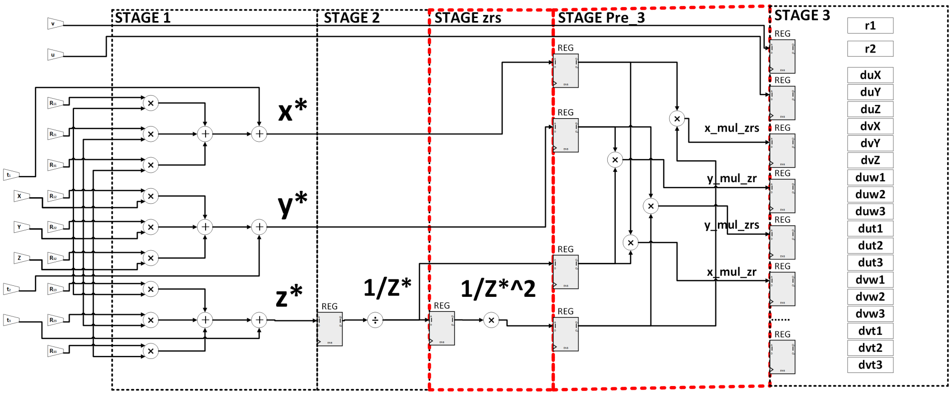


**Figure 4.6:** Modified Hardware Structure.

### 4.2.1 Performance and Area Modification

In hardware design, performance is usually the first consideration. We have used several methods to increase the operating speed of hardware accelerators.

First of all, as mentioned before, the comparison between the "Shift-Sub" divider and the Newton iteration method to realize the divider. We adopted a more efficient Newton divider and separated the calculation of "1/z*" into a new pipeline "Szrs" to shrink the critical path.

Through these methods, we successfully increased the operating speed of the hardware accelerator, allowing it to complete the operation of the JU and CC faster.

As shown in Fig. 2.2, a common formula is involved in calculating the Jacobian differential:

$$\frac{1}{z*} \times x* \tag{4.5}$$

$$\frac{1}{z*^2} \times x* \tag{4.6}$$

$$\frac{1}{z*} \times y* \tag{4.7}$$

$$\frac{1}{z*^2} \times y* \tag{4.8}$$

We list these calculations separately as the new pipeline "preS3", which will greatly reduce the use of DSP multipliers and some LUTs and FFs and will also reduce the critical path to a certain extent.

### 4.2.2 Comparison Before and After Modification

Table. 4.4 shows the comparison of the performance and resource utilization of the new (first row) and old (second row) hardware accelerator implementations.

| Critical Delay | LUTs | FFs | DSPs | Cycles per Feature |
|---|---|---|---|---|
| 7.337 ns | 1706 | 662 | 28 | 32 |
| 10.68 ns | 1644 | 292 | 36 | 96 |

**Table 4.4:** Performance and Area Comparison after Modification

Overall, we successfully improved the performance and area of the hardware accelerator. This enables our hardware accelerators to be more efficient and smaller when performing JU and CC operations.

Chapter 5

# Performance and Correctness Verification

Before the verification work started, we designed a "S1-IR" module and integrated the ROM generated by the "Distributed Memory Generator" inside it. For each iteration, we will manually modify the internal data in the ROM by modifying the COE file. The main function of "S1-IR" is to input the data (such as 3D points and camera models), which is controlled by the controller implemented as shown in Fig. 4.5, to the hardware accelerator implemented as shown in Fig. 4.6. The integrated system block diagram for integrating all modules is shown in Fig. 5.1. At the same time, the resource utilization rate and other reports after the placement and routing work on the FPGA "xc7a200tffg1156-3" through Vivado are shown in Table. 5.1. Compared with the critical path of the hardware accelerator itself shown in Table. 4.4, the critical path brought by the "S1-IR" internal ROM after system integration is longer, so the overall critical path (and area too) of the system has increased.

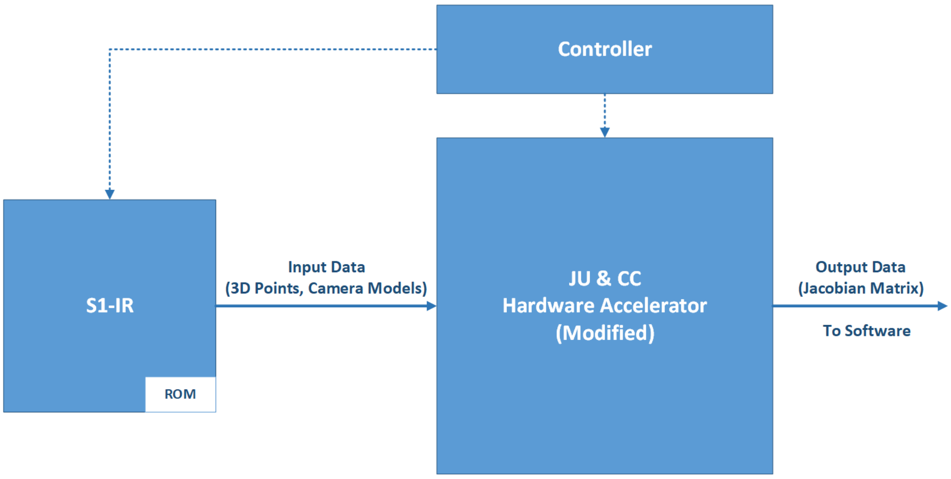


**Figure 5.1:** Integrated System Block Diagram.

| Critical Delay | LUTs | FFs | DSPs | Cycles per Feature |
|---|---|---|---|---|
| 8.114 ns | 6435 | 1023 | 28 | 32 |

**Table 5.1:** Performance and Area after System Integration

## 5.1 Computational Correctness Verification

In order to verify that our accelerator can output data that meets the accuracy requirements, we follow the MATLAB software algorithm simulation process shown in Fig. 3.3.in Chapter 2 and build a simulation process as shown in Fig. 5.2., which allows the hardware accelerator to run in parallel with MATLAB. In this way, we can bring the Jacobian matrix "J" and cost deviation "r" calculated by the hardware accelerator into the software and compare the resulting RMS with the original one. The comparison results are shown in Fig. 5.3. After completing the 10th iteration, the original RMS is 0.4906, while the new RMS after introducing the hardware accelerator is 0.5405. This shows that our hardware accelerator not only meets the calculation accuracy but also has little difference in correctness deviation compared with software.

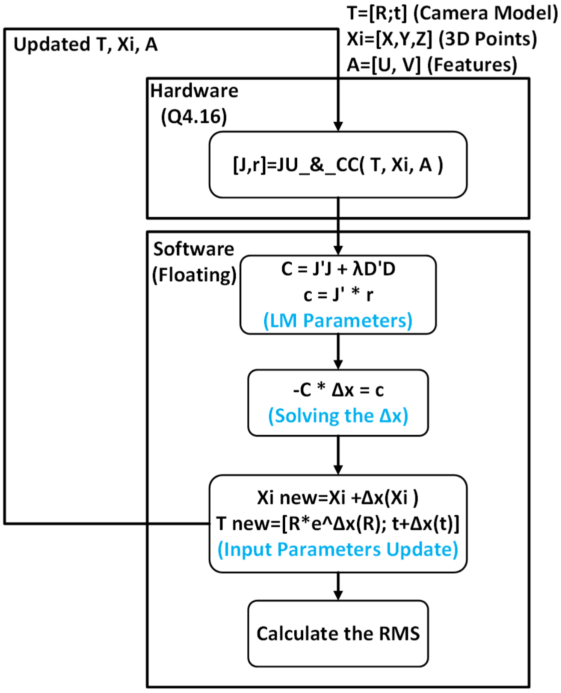


**Figure 5.2:** Correctness Verification Flowchart.

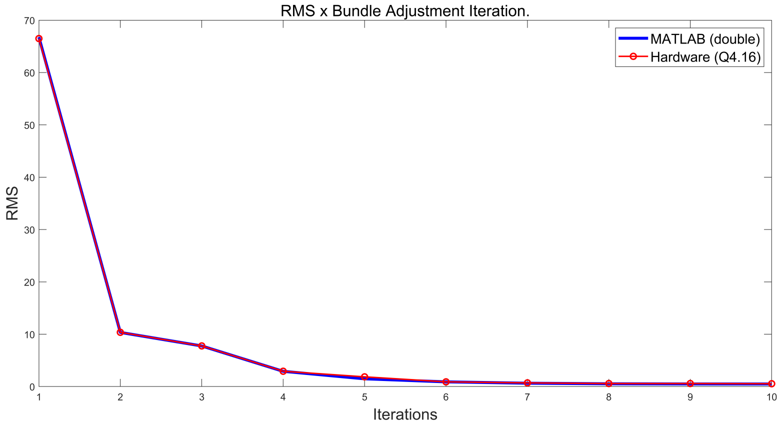


**Figure 5.3:** RMS Comparison for 10 Iterations.

## 5.2 Performance Comparison

The BA algorithm MATLAB simulation code built in the second chapter, we also implemented the same simulation code through Python and measured the running time of the JU and CC parts on the software. At the same time, we also know the simulation time of the whole system from the previous work. In this way, it can be concluded how much the hardware accelerator accelerates the running time of the JU and CC parts compared to the software at shown in the Table. 5.2.:

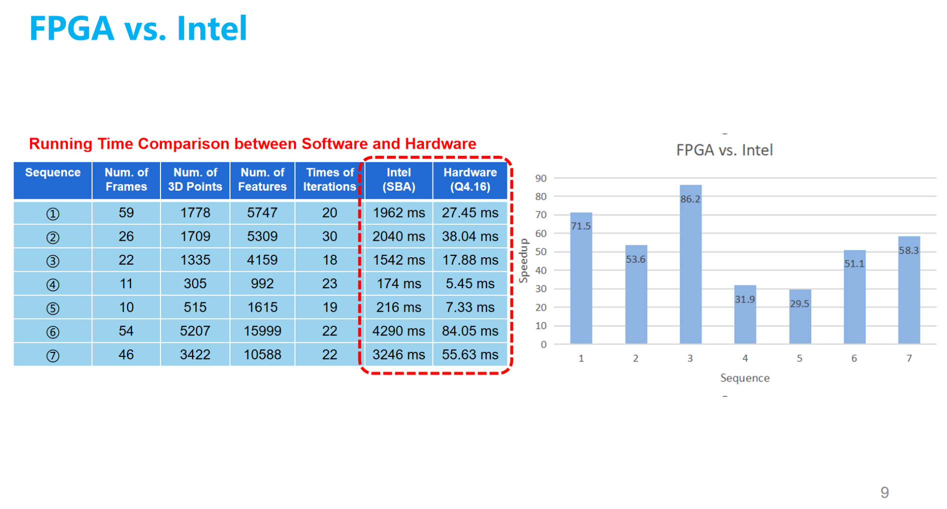
| Python(Numpy) | Hardware(Q4.16) | Speedup |
|---|---|---|
| 328.64 ms | 6.92 ms | 47.5 |

**Table 5.2:** Speedup of Hardware Compared to Software for 10 Iterations.

At the same time, we refer to other work [2] [3] to find out the running time of the JU and CC parts of BA on different data sets. Because our hardware accelerator has the same 32 clock cycles for the execution time of each Feature in each iteration, and the highest system frequency is 134MHz. From this combination Eq. 5.1, we can easily estimate the corresponding running time of the hardware accelerator under each data set.

$$T_{Execution}(Sec.) = (features \times iterations \times cycles)/134000000 \qquad (5.1)$$

The accelerator run time compared to the run time of the JU and CC parts with SBA on the Intel P4 processor is shown in Fig. 5.4. Since small-scale matrices do not consume too much time in software operations, the speedup ratios of sequences 4 and 5 are relatively insignificant. In the case of relatively large input data sets, our hardware accelerator can perform better.



FPGA vs. Intel

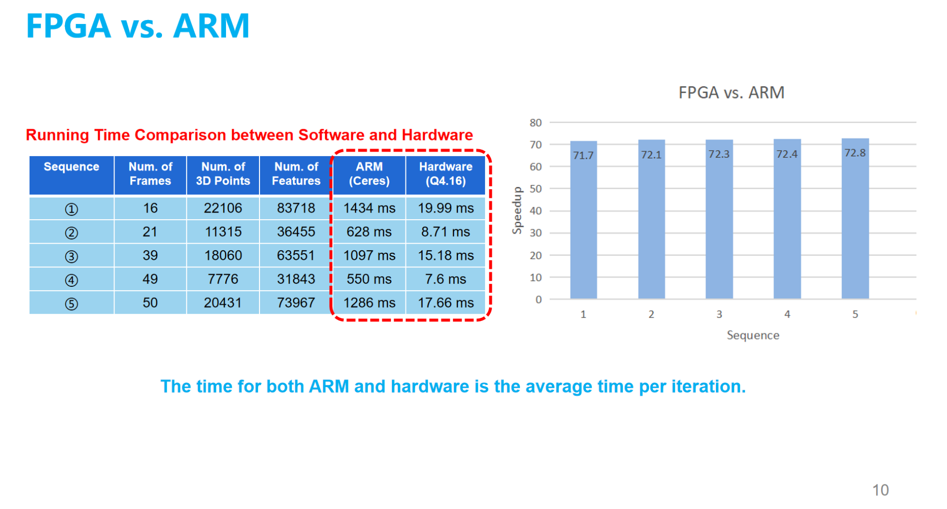
Running Time Comparison between Software and Hardware

| Sequence | Num. of Frames | Num. of 3D Points | Num. of Features | Times of Iterations | Intel (SBA) | Hardware (Q4.16) |
|---|---|---|---|---|---|---|
| ① | 59 | 1778 | 5747 | 20 | 1962 ms | 27.45 ms |
| ② | 26 | 1709 | 5309 | 30 | 2040 ms | 38.04 ms |
| ③ | 22 | 1335 | 4159 | 18 | 1542 ms | 17.88 ms |
| ④ | 11 | 305 | 992 | 23 | 174 ms | 5.45 ms |
| ⑤ | 10 | 515 | 1615 | 19 | 216 ms | 7.33 ms |
| ⑥ | 54 | 5207 | 15999 | 22 | 4290 ms | 84.05 ms |
| ⑦ | 46 | 3422 | 10588 | 22 | 3246 ms | 55.63 ms |

9

**Figure 5.4:** Hardware Speedup Compare to Software (SBA).

Similarly, we also compared the running time of JU and CC with the Ceres library on the popular ARM-embedded processor Cortex-A9, as shown in Fig. 5.5. This group of comparisons illustrates well that our hardware accelerator has great potential to be applied in embedded scenarios.



FPGA vs. ARM

Running Time Comparison between Software and Hardware

| Sequence | Num. of Frames | Num. of 3D Points | Num. of Features | ARM (Ceres) | Hardware (Q4.16) |
|---|---|---|---|---|---|
| ① | 16 | 22106 | 83718 | 1434 ms | 19.99 ms |
| ② | 21 | 11315 | 36455 | 628 ms | 8.71 ms |
| ③ | 39 | 18060 | 63551 | 1097 ms | 15.18 ms |
| ④ | 49 | 7776 | 31843 | 550 ms | 7.6 ms |
| ⑤ | 50 | 20431 | 73967 | 1286 ms | 17.66 ms |

The time for both ARM and hardware is the average time per iteration.

10

**Figure 5.5:** Hardware Speedup Compare to Software (Ceres).

## 5.3 Estimated Energy Saving Comparison

We also estimated the power consumption of our system through Vivado to be 0.221 W. Although there may be some differences from the actual on-chip power consumption, we can still make some energy-saving comparisons. It is known that the Thermal Design Power (TDP) of the ARM Cortex-A9 at 667 MHz main frequency is 1.5 W [2], and the following comparison results can be obtained by combining the running time of hardware and ARM:

| Sequence | ARM(Ceres) | Hardware(Q4.16) | Energy Saved |
|---|---|---|---|
| 1 | 2151 mJ | 4.2 mJ | 99.805% |
| 2 | 942 mJ | 1.8 mJ | 99.809% |
| 3 | 1645.5 mJ | 3.2 mJ | 99.806% |
| 4 | 825 mJ | 1.6 mJ | 99.806% |
| 5 | 1929 mJ | 3.7 mJ | 99.808% |

**Table 5.3:** Estimated Energy Saving Comparison

Chapter 6

# Conclusion and Future Works

## 6.1 Conclusion

In this thesis, we investigate the hardware accelerator for the BA algorithm, which is an essential non-linear optimization part of the SLAM process. Implementing efficient hardware accelerators to improve the computational efficiency of BA algorithms is our key objective. We successfully constructed a hardware accelerator for the JU and CC parts of the BA algorithm by comprehending the theoretical basis, developing a high-level architecture, and translating it into our hardware structure. We conducted the initial verification and further comparison of the hardware accelerator's performance and accuracy after we modified our hardware structure to be more extended.

The outcomes demonstrate a considerable increase in the computational efficiency of the BA algorithm due to the hardware accelerator. Using specialized hardware, we speed up camera parameter and 3D point coordinate optimization processing, leading to more accurate 3D models of the environment and a more reliable motion trace of a moving camera. This development opens up new opportunities for real-world applications where real-time speed is crucial, such as self-driving automobiles and robotic navigation.

## 6.2 Future Works

Although this thesis work has achieved a significant improvement in the computational efficiency of the BA algorithm through hardware accelerating, there is still room for further exploration and improvement. Here are some potential future research directions:

- Hardware Optimizations: Explore new approach for hardware accelerators to improve its efficiency and reduce resource utilization. This may involve research on new architectures, reducing latency, improving memory access patterns, and more. For example, a divider will be realized by LUTs to reduce the overall latency of the hardware accelerator.
- Integrate into the SLAM system: Test the SLAM system integrated with this BA hardware accelerator IP in the real environment and compare its performance with the existing SLAM system such as ORB-SLAM. It can be

seen that our hardware accelerator has a high energy efficiency advantage compared with the popular ARM embedded processor, which implements JU and CC in SLAM through software. We will alter this hardware accelerator's interface with on-chip communication bus protocol (such as AXI4, etc.), then package it as an IP and also open source it to facilitate subsequent research by others.

- Shure Elimination Hardware Accelerator: While further improving the hardware accelerator, consider the design and implementation of the Shure Elimination hardware accelerator. Schur Elimination is a common method for sparse matrix solving and is often used in the BA algorithm. And this part often takes a lot of running time. By developing a dedicated hardware accelerator, the Schur Elimination process can be accelerated, thereby further improving the computational efficiency of the entire BA algorithm. This work could include hardware architecture optimization based on simplified Schur Elimination algorithm, optimization of data storage patterns, and application of parallel computing techniques.

# Acronyms

**ASIC** Application Specific Integrated Circuit. 22

**ASMD** Algorithm State Machine Diagram. 6, 21, 25

**BA** Bundle Adjustment. i, iii, 1, 2, 4, 5, 9, 10, 13, 16–18, 31, 35, 36

**CC** Cost-function Calculation. i, vii, 2, 4–7, 9, 20, 21, 25, 27, 31, 32, 35, 36

**CV** Computer Vision. i, iii, 2

**FPGA** Field Programmable Gate Arrays. 2, 5, 6, 18, 22, 23, 29

**JU** Jacobian Update. i, vii, 2, 4–7, 9, 17, 20, 21, 25, 27, 31, 32, 35, 36

**LM** Levenberg Marquardt. v, 11, 12, 17, 18, 20

**LUTs** Look-up Tables. 6, 22, 35

**RMS** Root Mean Square. 17, 18, 20, 30

**ROM** Read-Only Memory. 6, 29

**SIFT** Scale-Invariant Feature Transform. 17

**SLAM** Simultaneous Localization and Mapping. i, iii, 1, 2, 12, 13, 17, 35, 36

# References

[1] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment — a modern synthesis," in *Vision Algorithms: Theory and Practice* (B. Triggs, A. Zisserman, and R. Szeliski, eds.), (Berlin, Heidelberg), pp. 298–372, Springer Berlin Heidelberg, 2000.

[2] Q. Liu, S. Qin, B. Yu, J. Tang, and S. Liu, "$\pi$-ba: Bundle adjustment hardware accelerator based on distribution of 3d-point observations," *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 1083–1095, 2020.

[3] M. I. A. Lourakis and A. A. Argyros, "Sba: A software package for generic sparse bundle adjustment," *ACM Trans. Math. Softw.*, vol. 36, mar 2009.

[4] C. Zach, "Robust bundle adjustment revisited," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 772–787, Springer International Publishing, 2014.

[5] M. Cao, L. Zheng, W. Jia, and X. Liu, "Fast incremental structure from motion based on parallel bundle adjustment," *Journal of Real-Time Image Processing*, vol. 18, pp. 379–392, 2021.

[6] G. Sibley, "Relative bundle adjustment," *Department of Engineering Science, Oxford University, Tech. Rep*, vol. 2307, no. 09, 2009.

[7] X. Gao, T. Zhang, Y. Liu, and Q. Yan, "14 lectures on visual slam: From theory to practice," 2017.

[8] Cheng Wei, "Bundle adjustment," 2018. `https://scm_mos.gitlab.io/vision/bundle-adjustment/`, Last accessed on 2019-12-25.

[9] D. Sibley, C. Mei, I. Reid, and P. Newman, "Adaptive relative bundle adjustment," in *Robotics: science and systems*, 2009.

[10] F. Endres, J. Hess, J. Sturm, D. Cremers, and W. Burgard, "3-d mapping with an rgb-d camera," *IEEE Transactions on Robotics*, vol. 30, no. 1, pp. 177–187, 2014.

[11] T. Botterill, S. Mills, and R. Green, "Correcting scale drift by object recognition in single-camera slam," *IEEE Transactions On Cybernetics*, vol. 43, no. 6SI, pp. 1767–1780, 2013.

[12] K. Konolige and M. Agrawal, "Frameslam: From bundle adjustment to real-time visual mapping," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1066–1077, 2008.

[13] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-squares fitting of two 3-d point sets," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 5, pp. 698–700, 1987.

[14] P. Agarwal, W. Burgard, and C. Stachniss, "Survey of geodetic mapping methods: Geodetic approaches to mapping and the relationship to graph-based slam," *Robotics Automation Magazine, IEEE*, vol. 21, pp. 63–80, Sept 2014.

[15] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, "Complete solution classification for the perspective-three-point problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, pp. 930–943, Aug 2003.

[16] N. Carlevaris-Bianco, M. Kaess, and R. Eustice, "Generic node removal for factor-graph slam," *Robotics, IEEE Transactions on*, vol. 30, pp. 1371–1385, Dec 2014.

[17] A. Gee, D. Chekhlov, A. Calway, and W. Mayol-Cuevas, "Discovering higher level structure in visual slam," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 980–990, 2008.

[18] E. Eade and T. Drummond, "Edge landmarks in monocular slam," *Image and Vision Computing*, vol. 27, no. 5, pp. 588–596, 2009.

[19] L. Carlone, M. K. Ng, J. J. Du, B. Bona, and M. Indri, "Simultaneous localization and mapping using Rao-Blackwellized particle filters in multi robot systems," *Journal of Intelligent & Robotic Systems*, vol. 63, no. 2, pp. 283–307, 2011.

[20] J. Artieda, J. M. Sebastian, P. Campoy, J. F. Correa, I. F. Mondragón, C. Martínez, and M. Olivares, "Visual 3-d slam from uavs," *Journal of Intelligent and Robotic Systems*, vol. 55, no. 4-5, pp. 299–321, 2009.

[21] G. Dubbelman and B. Browning, "Cop-slam: Closed-form online pose-chain optimization for visual slam," *Robotics, IEEE Transactions on*, vol. 31, pp. 1194–1213, Oct 2015.

[22] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[23] B. Harish, K. Sivani, and M. Rukmini, "Design and performance comparison among various types of adder topologies," in *2019 3rd international conference on computing methodologies and communication (ICCMC)*, pp. 725–730, IEEE, 2019.

[24] M. D. Ercegovac and T. Lang, *Digital arithmetic*. Elsevier, 2004.

[25] D. R. Kincaid and E. W. Cheney, *Numerical analysis: mathematics of scientific computing*, vol. 2. American Mathematical Soc., 2009.