TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DIPLOMA THESIS

Acceleration of Simultaneous Localization and Mapping (SLAM) Algorithms on Graphics Processing Units (GPUs) for Unmanned Air Drones

---

Author:
Panagiotis FELEKIS

Thesis Committee:
Prof. Apostolos DOLLAS
Assoc. Prof. Michail G.LAGOUDAKIS
Assoc. Prof. Panagiotis PARTSINEVELOS (MRE/TUC)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

October 12, 2021

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Diploma Thesis

Acceleration of Simultaneous Localization and Mapping (SLAM) Algorithms on Graphics Processing Units (GPUs) for Unmanned Air Drones

by Panagiotis FELEKIS

In order to achieve fully autonomous operation in an unknown environment, many robots rely on cameras and vision algorithms to figure out where to place an object, turn a screw, or weld two pieces of metal together. Mobile robots must solve two basic problems: create a map of the environment and position themselves into this map. Simultaneous localization and mapping (SLAM) approaches can incrementally construct a map of the robot's surrounding environment, while estimating the robot's position in the map. Visual SLAM (vSLAM) uses the camera to obtain corresponding two dimensional digital images from the real three-dimensional world. Due to high computational demands of vSLAM, scaled-down versions are used with smaller resolution and less key features, resulting in poor estimations.

In this thesis, we propose an accelerated version of ORB vSLAM that uses a GPU. In our version, we use high resolution images which results in more accurate and rich results. Our system operates in NVIDIA Jetson Tx2 embedded module which is suitable for autonomous robots due to low power consumption.

In terms of performance results, our system performs almost identically to a fully-powered desktop CPU, while consuming 5$\times$ less power. We also prove that our system is as much accurate as the non-accelerated vSLAM system, by using a well-established accuracy dataset.

ΠΟΛΥΤΕΧΝΙΟ ΚΡΗΤΗΣ

Περίληψη

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Επιτάχυνση με Χρήση Κάρτας Γραφικών του Αλγορίθμου **SLAM** για Χαρτογράφηση και Εντοπισμό Θέσης σε Μη Επανδρωμένα Αεροχήματα

βψ Φελέκης Παναγιώτης

Ενα αυτόνομο ρομπότ για να λειτουργήσει σε ένα άγνωστο περιβάλλον βασίζεται σε κάμερες και αλγορίθμους όρασης. Τα κινούμενα ρομπότ πρέπει να λύσουν δύο προβλήματα: να δημιουργήσουν έναν χάρτη από το γύρο περιβάλλον τους και στη συνέχεια, να τοποθετηθούν σε αυτόν τον χάρτη. Ένα σύστημα ταυτόχρονης χαρτογράφησης και εντοπισμού θέσης (SLAM) χρησιμοποιείται για να δημιουργήσει έναν χάρτη από κάποια σημεία αναφοράς και να εκτιμήσει τη θέση του ρομπότ, σύμφωνα με αυτά τα σημεία. Το σύστημα SLAM μπορεί να εκτελεστεί χρησιμοποιώντας κάμερα, η οποία προσφέρει εικόνες δύο διαστάσεων από το τρισδιάστατο περιβάλλον. Οι κάμερες προσφέρουν εικόνες υψηλής ευκρίνειας, με πλούσιο χρώμα και επιφάνειες όπου χρησιμοποιούνται για να δημιουργηθεί ένας πλούσιος χάρτης. Λόγω των υψηλών υπολογιστικών απαιτήσεων του SLAM , χρησιμοποιούνται υποδεέστερες εκδοχές με μικρότερη ανάλυση και λιγότερα βασικά χαρακτηριστικά, κάτι που έχει ως αποτέλεσμα κακές εκτιμήσεις.

Σε αυτή την έρευνα προτείνεται ένα SLAM σύστημα που χρησιμοποιεί κάρτα γραφικών για την επιτάχυνση του. Με αυτό τον τρόπο μπορουν να χρησιμοποιηθούν περισσότερα δεδομένα από την κάμερα για πλουσιότερα αποτελέσματα. Το σύστημα λειτουργεί σε ενσωματωμένο σύστημα NVIDIA Jetson Tx2 , το οποίο είναι κατάλληλο για αυτόνομα ρομπότ, λόγω της υψηλής ενεργειακής απόδοσης που παρέχει. Συγκρίνεται η ακρίβεια, η υπολογιστική και ενεργειακή απόδοση του συστήματος που χρησιμοποιείται σε έναν προσωπικό υπολογιστή και στο ενσωματωμένο σύστημα.

Acknowledgements

I would like to express my deep gratitude to my supervisor Prof. Dollas for his guidance, devotion and help he provided for the completion of the present thesis, especially during those unseen circumstances we are currently living. I would also like to thank Prof. Partsinevelos and all the people of Sense Lab for all the inspiration, ideas and equipment they provided.

Lastly, I would like to thank my family and friends for the continuous support and love they showed me during my university years...

Contents

Abstract iii

Acknowledgements vii

Contents ix

List of Figures xiii

List of Tables xv

List of Algorithms xvii

List of Abbreviations xix

1 Introduction 1
- 1.1 Motivation and Scientific Contributions . . . 1
- 1.2 Thesis Outline . . . 2

2 Related Work 5
- 2.1 Unmanned Vehicles . . . 5
  - 2.1.1 Unmanned Ground Vehicles . . . 5
  - 2.1.2 Unmanned Underwater Vehicles . . . 6
  - 2.1.3 Unmanned aerial vehicle . . . 6
- 2.2 SLAM technologies . . . 8
  - 2.2.1 LiDar SLAM . . . 9
  - 2.2.2 Sonar SLAM . . . 10
- 2.3 Camera SLAM . . . 10
  - 2.3.1 LSD SLAM . . . 11
  - 2.3.2 OpenVSLAM . . . 11
  - 2.3.3 ORB-SLAM2 . . . 12
- 2.4 Processing platforms . . . 13
  - 2.4.1 The FPGA Perspective . . . 13
  - 2.4.2 GPU for computer vision . . . 14

2.4.3 NVIDIA Hardware . . . 15
2.5 Computer Vision Software . . . 16
2.5.1 CUDA programming . . . 16
2.5.2 OpenCv . . . 16

3 Theoretical Modeling **19**
3.1 Feature Extraction . . . 19
3.1.1 Sobel-Feldman operators . . . 20
3.1.2 Canny Edge Detector . . . 21
3.1.3 Harris Corner Detection . . . 24
3.1.4 Scale-Invariant Feature Transform . . . 26
3.1.5 Speed-up Robust Features . . . 28
3.1.6 Oriented FAST and Rotated BRIEF . . . 30
3.2 Feature Matching . . . 31
3.2.1 Brute Force Matching . . . 32
3.2.2 FLANN Based Matcher . . . 33
3.3 Three Dimensions Reconstruction . . . 33
3.3.1 Camera Calibration . . . 34
3.3.2 Zhang's camera calibration method . . . 35
3.3.3 Random Sample Consensus . . . 37
3.3.4 Structure from motion . . . 38

4 System Architecture **41**
4.1 vSLAM CPU implementation . . . 42
4.2 GPU acceleration . . . 44
4.2.1 GPU implementation of oFAST . . . 45
4.2.2 GPU implementation of BRIEF . . . 49
4.2.3 GPU brute force matching . . . 50
4.2.4 Task allocation timing . . . 51

5 System Verification and Performance Evaluation **53**
5.1 Specifications of platforms . . . 53
5.1.1 Jetson TX2 . . . 53
5.1.2 Desktop PC . . . 54
5.2 Datasets and Performance Metrics . . . 55
5.2.1 EuRoc Dataset . . . 55
5.2.2 Our Dataset . . . 61
5.3 Summary of Results . . . 65

6 Conclusions and Future Work **67**
6.1 Conclusions and Future Work 67

List of Figures

2.1 Ocado's robot warehouse 6
2.2 UUV performing coral reef repopulation. 7
2.3 Drone 7
2.4 ORB-SLAM system overview. [20] 13
2.5 CPU and GPU simple architecture model. 15

3.1 Sobel-Feldman operators 22
3.2 Step-by-step Canny edge detection 24
3.7 Matching ORB features between an image and a rotated version of the same image. 32
3.8 Zhang's calibration 36
3.9 2D RANSAC example 38
3.10 Epipolar Geometry 39

4.1 Jetson Tx2 hardware design. 42
4.2 Using a second CPU core for visualization. 43
4.3 CPU to GPU data flow 45
4.4 Building Gaussian Pyramid 46
4.5 FAST feature detection illustration. We compare the intensity of pixel P with the surrounding 16 pixels to determine if it feature. 47
4.6 Coordinate normalization and non-maximum suppression . . 48
4.7 GPU computational resources allocation 49
4.8 Comparing pair intensities around the feature point to create BRIEF descriptor. 50
4.9 A graph depicting every sub-step of feature extraction and matching. 51
4.10 Before and after comparison of task allocation. 52

5.1 GPU feature extraction and feature matching. 56
5.2 Trajectory comparisons between ground truth, CPU and GPU versions using evo package[39] 58

5.3 Execution time(seconds) of EuRoc datasets for A) PC platform, B) Nvidia Jetson. 60
5.4 Execution time(milliseconds) of A) Feature extraction, B) Feature Matching. 61
5.5 Zhang method for camera calibration 62
5.6 Execution time of our dataset for A) PC platform, B) Nvidia Jetson. 63
5.7 Execution time of feature extraction and matching for 1920x1080 pixel video with 1500 max feature point 64

List of Tables

4.1 Average desktop CPU execution time for every SLAM step and what parameters affect it. 44

5.1 NVIDIA Jetson TX2 specifications(Link). 53
5.2 NVIDIA Jetson TX2 clock configuration with power modes(Link). 54
5.3 Desktop PC specifications 55
5.4 Absolute error between estimated and true trajectories 59
5.5 Speed up for PC and Jetson platform between CPU and GPU versions 59
5.6 SLAM speed-up for PC and Jetson platform, between CPU and GPU versions. 64
5.7 Summary of various results. 66

List of Algorithms

1 Sobel-Feldman edge detection 21
2 Feature extraction using CUDA. 47
3 Feature matching using CUDA. 51

List of Abbreviations

SLAM	**S**imultaneous **L**ocalization **A**nd **M**apping
CPU	**C**entral **P**rocessor **U**nit
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
CS	**C**omputer **S**cience
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DNN	**D**eep **N**eural **N**etwork
UGV	**U**nmanned **G**round **V**ehicles
UUV	**U**nmanned **U**nderwater **V**ehicles
UAV	**U**nmanned **A**erial **V**ehicles
GPS	**G**lobal **P**ositioning **S**ystem
LiDaR	**L**aser **D**etection and **R**anging
ROS	**R**obotic **O**perating **S**ystem
ICP	**I**terative **C**losest **P**oint
RGB	**R**ed **G**reen and **B**lue
EKF	**E**xtended **K**alman **F**ilter
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPU	**G**raphic **P**rocessor **U**nit
SIFT	**S**cale-**I**nvariant **F**eature **T**ransform
SURF	**S**peed-up **R**obust **F**eatures
BRIEF	**B**inary **R**obust **I**ndepent **E**lementary **F**eatures
ORB	**O**riented FAST and **R**otated **B**RIEF
DOG	**D**ifference **O**f **G**aussians
RAM	**R**andom **A**ccess **M**emory

Dedicated to my family and friends…

Chapter 1

Introduction

Simultaneous localization and mapping (SLAM) can trace its early development back to the robotics industry in the 1980s and 1990s. This technique was originally proposed to achieve autonomous control of robots [1] in a well-know environment like a production line.

Today, SLAM technology is used in many industries and it has really opened up opportunities to better mapping and understanding of environments whether they are indoor[2][3], outdoor[4], in-air or underground[5]. SLAM gained popularity with the emergence of mobile robotics in many sectors like Medical and Healthcare, Warehousing and Defense. Mobile robots market also advance in personal, every day applications, like drones, robot vacuums or autonomous cars.

Important applications of SLAM include:

- Search and rescue scenarios where reaching with conventional means is difficult.
- Autonomous driving of vehicles.
- Planetary, oceanic and aerospace exploration.
- Surveillance systems.
- Medicine delivery or transportation.

1.1 Motivation and Scientific Contributions

Visual SLAM, also known as vSLAM, calculates the position and orientation of a device with respect to its surroundings while mapping the environment at the same time, using only visual inputs from a camera. vSLAM typically tracks points of interest through successive camera frames to triangulate the

3D position of the camera, this information is then utilized to build a 3D map. The technique change slightly, depending on the type of camera sensor we use, but the core functions and algorithm stays the same.

While visual SLAM only uses one low-cost sensor, the camera, lacks the performance. The camera usually outputs images with millions of pixels, with many layers of color information and light intensity data. Computer vision aims to exploit patterns from all this data, in order to navigate safely in an unexplored environment. Due to the dense data, the real-time operation, and the unknown environment, vSLAM technologies usually make some compromises like limiting the image resolution, using high-end expensive sensors or perform calculations remotely.

In the present study, we propose an accelerated version of visual SLAM in order to handle high resolution images while maintaining real-time frames per second. We attempt to increase the processing speed of the core functions by using commercial graphic cards on embedded systems. We support that high-resolution video will provide more information to the robot where it can extract more meaningful features resulting in more accurate results.

In terms of results, we used a visual-inertial dataset that contains synchronized images, IMU measurements, and ground truth to verify the accuracy of our system. Afterwards, we used our dataset containing high-resolution images to measure performance. After numerous testings, we concluded that our system performs almost identical to a desktop CPU while consuming 5 times less power. More information will be presented in chapter 5.

1.2 Thesis Outline

- **Chapter 2 - Related Work:** We describe the types of unmanned vehicles and popular SLAM technologies based on the sensor or the computational platform used.
- **Chapter 3 - Theoretical Modeling:** We break down the SLAM problem to smaller steps and describe and compare techniques used through time.
- **Chapter 4 - System Architecture:** Description of our system on a dual-core CPU and how we accelerate it with GPU.

- **Chapter 5 - System Verification and Performance Evaluation:** We compare our two systems in a PC and embedded platform. We use two datasets for efficiency and performance.
- **Chapter 6 - Conclusions and Future Work:** Conclusion of our work and some future extensions and thoughts.

Chapter 2

Related Work

2.1 Unmanned Vehicles

An unmanned or automated vehicle is a vehicle without a crew on board. Unmanned vehicles can either be controlled remotely or sense the environment around them and navigate their own. Unmanned vehicles are used nowadays for many applications that are dangerous or impossible for people to execute. Throughout the years unmanned vehicles became more reliable and precise. In recent years, unmanned robots infiltrate the general consumer market to assist their users. We distinguish unmanned vehicles into three types based on the field of operation: ground, water, and air.

2.1.1 Unmanned Ground Vehicles

Unmanned Ground Vehicles (UGV) as the name suggest is a piece of mechanized equipment that moves through the surface of the earth. One of the most popular forms of UGV nowadays is the self-driving cars that helped to map public roads for navigation consumer platforms. Due to safety regulations and because the environment is unpredictable, a self-driving car is equipped with state-of-art sensors like LiDar, Sonar, and cameras. A self-driving car must always have excellent knowledge of its surroundings in order to make decisions that are safe for both the passengers and other cars.

An additional field with increasing popularity for UGVs is warehouse inventory management. Hundreds or sometimes thousands of swarm robots cooperate to deliver orders much faster than humanly possible. Every robot is equipped with sensors and a communication system to move around the warehouse without crumbling one to the other. An algorithm controls the

robot swarm by sending optimal delivery paths to each system entity for efficient outcomes. uu

FIGURE 2.1: Ocado's robot warehouse(source).

2.1.2 Unmanned Underwater Vehicles

Unmanned Underwater Vehicles (UUVs) are designed for a variety of missions like intelligence gathering, mine-hunting, scientific exploration, and ship hull inspection, providing an accurate picture under the surface. UUVs are also equipped with multiple sensors, including obstacle avoidance sonars, multi-beam echo sounder, and advanced navigation/positioning sensors.

2.1.3 Unmanned aerial vehicle

Unmanned aerial vehicles (UAVs) also popularly known as drones, are the most common among the vehicles. The flight of a UAV may operate with many degrees of autonomy: either under remote control by a human operator or autonomously with onboard controllers. UAVs originated for military applications but their use is rapidly expanding to commercial, scientific, agricultural, and many more fields.

UAVs come in two form factors: rotary-wing and fixed-wing. The first one is easy to take off and maneuver in small areas with the cost of low battery life. Fixed-wing is usually used for long travel distances but need open space to operate.

FIGURE 2.2: UUV performing coral reef repopulation (source).

Drones are equipped with a variety of sensors depending on the mission they must accomplice. The most used sensors are the camera and inertial measurement unit (IMU). Depending on the mission UAVs and also have accelerometers, GPS, LiDar, and many variants of cameras(Thermal, Stereo, RGB).

FIGURE 2.3: UAVs during agriculture inspection(source).

2.2 SLAM technologies

SLAM stands for simultaneous localization and mapping and it is the task of estimating a map of the environment and at the same time localizing your sensor or your robot in that map that you're currently building. It is something that mobile robots need whenever they move into unknown or partially unknown environments. If the map is given then the position of the robot is much easier to find and if the pose estimate is given then the mapping is also relatively easy, but solving both problems together is much harder and computational challenging.

In order to solve the slam problem we have to distinguish between the front-end and the back-end:

1. The front-end is the part that takes the raw sensor data and turns them into an immediate representation such as constraints in an optimization problem or probability distribution about the location of a landmark. The front-end is a very task-specific part of slam
2. The back-end takes the intermediate representation of the front-end and solves the underlying state estimation or optimization problem, like estimating parameters that describe where objects are in the environment or where my platform is in the world coordinates.

In the back-end, we typically find three different categories of approaches:

1. Extended Kalman Filter(EKF) is the nonlinear version of the Kalman filter [6] [7]. Kalman filter is a two-step process, the prediction and correction step. The prediction step uses control commands, to estimate the position of our system at the next point in time. The correction step takes into account the sensor observations as a means to correct for potential mistakes in the prediction step. Kalman filter makes two assumptions. Firstly that the world is Gaussian and secondly all models are linear. Extended Kalmar Filter performs linearization, for every model, via Taylor series in order to use Kalman filter in real-world scenarios.
2. Particle filter is a technique for estimating the state of a dynamic system similar to the Kalman filter [8]. It updates the current belief based on so-called motion information of control commands and based on observations. It allows us to describe arbitrary probability distributions due to there is no assumption that we are in a Gaussian world. It uses

many particles or samples which are hypotheses for the system being in one single state. For every particle, we have a prediction step and a correction step similar to a common filter. Rao–Blackwellized particle filtering (RBPF), derived from particle filter, was thus applied in the algorithm proposed by Montemerlo, named FastSLAM [9].

3. Least squares or also known as Graph-based SLAM [10]. Graph-based approaches are the most popular today and as the name suggests we use a graph to represent the variables and the relations between those variables. There are different types of graphs; the two most popular are either the post-graph which is a graph that contains only the poses and marginalizes out the map information or Factor graphs, which have a vector sitting in between nodes and information coming from the front-end or other sources.

SLAM technologies differ from the other based on the sensor and approach used. We will firstly categorize SLAM based on the three common sensors used: LiDar, sonar, and camera.

2.2.1 LiDar SLAM

LiDAR SLAM implementation uses a laser sensor. Compared to Visual SLAM which used cameras, lasers are more precise and accurate. The high rate of data capture with more precision allows LiDAR sensors for use in high-speed applications such as moving vehicles such as self-driving cars and drones. The only drawback is the sensor availability and high price. The output data of LiDAR sensors often called point cloud data is available with 2D (x, y) or 3D (x, y, z) positional information.

The laser sensor point cloud provides high-precision distance measurements and works very effectively for map construction with SLAM. Generally, movement is estimated sequentially by matching the point clouds. The calculated movement (traveled distance) is used for localizing the vehicle. For LiDAR point cloud matching, iterative closest point (ICP) and normal distributions transform (NDT) algorithms are used. 2D or 3D point cloud maps can be represented as a grid map or voxel map.

The most widely used LiDAR-based SLAM libraries that have ROS wrappers - Gmapping [11], Google Cartographer [12], and Hector SLAM [13].

2.2.2 Sonar SLAM

Sonar imaging is a well-established technology mostly used for naval applications. Sound wavelengths in water are about 2,000 times longer than those of visible light. Because of its longer wavelengths, sound can go around suspended particles that would otherwise block and scatter light waves. Light can't penetrate very far in these conditions, making optical systems (like underwater cameras) ineffective. Also, optical images lack the range information found in sonar images.

The performance of an imaging sonar, from the distance at which they can detect an object, to the clarity of the image, to the number of images they can display per second, is determined by several specifications, most notably the operating frequency, acoustic beamwidth and processing power and time to form an image. Sound Metrics sonars use acoustic lens technology which forms beams instantaneously using zero power.

Using the acoustic images from the sensor we can extract some features and later on can be used to create a map. Generally speaking, a lower frequency increases the distance at which an image can be captured. A higher frequency and a smaller beamwidth used to map an object will deliver clearer images.

For all the above reasons, sonar SLAM technologies like [14] and [15] have uses for underwater environments. For ground or aerial robots, a sonar sensor is used in combination with other sensors for better mapping of nearby objects [16].

2.3 Camera SLAM

Most visual SLAM systems work by tracking set points through successive camera frames to triangulate their 3D position, while simultaneously using this information to approximate camera pose. The goal of these systems is to map their surroundings in relation to their location for the purposes of navigation.

This is possible with a single 3D vision camera, unlike other forms of SLAM technologies. As long as there are a sufficient number of points being tracked through each frame, both the orientation of the sensor and the structure of the surrounding physical environment can be rapidly understood.

All visual SLAM systems are constantly working to minimize reprojection error, or the difference between the projected and actual points, usually through an algorithmic solution called bundle adjustment. Visual SLAM systems need to operate in real-time, so often location data and mapping data undergo bundle adjustment separately, but simultaneously, to facilitate faster processing speeds before they're ultimately merged.

We will now present some popular visual SLAM systems that differ from the others in the way they process the image data.

2.3.1 LSD SLAM

LSD-SLAM [17] is a novel, direct monocular SLAM technique: Instead of using keypoints, it directly operates on image intensities both for tracking and mapping. The camera is tracked using direct image alignment, while geometry is estimated in the form of semi-dense depth maps, obtained by filtering over many pixelwise stereo comparisons.

As a direct method, LSD-SLAM uses all information in the image, including e.g. edges – while keypoint-based approaches can only use small patches around corners. This leads to higher accuracy and more robustness in sparsely textured environments (e.g. indoors), and a much denser 3D reconstruction. Further, as the proposed pixelwise depth-filters incorporate many small-baseline stereo comparisons instead of only a few large-baseline frames, there are much fewer outliers.

2.3.2 OpenVSLAM

OpenVSLAM is based on an indirect SLAM algorithm with sparse features [18]. One of the noteworthy features of OpenVSLAM is that the system can deal with various types of camera models, such as perspective, fisheye, and equirectangular. If needed, users can implement extra camera models (e.g. dual fisheye, catadioptric) with ease.

OpenVSLAM implements the FAST algorithm for keypoint detection [19] and binary vector for descriptor [20].

2.3.3 ORB-SLAM2

The ORB-SLAM2 is the state-of-the-art indirect visual SLAM algorithm, similar to OpenVSLAM. It is a complete SLAM system that works with monocular, stereo, and RGB-D cameras. It comes with map reuse, loop closing, and re-localization capabilities. It is designed to work in real-time on standard a CPU. In the past six years, hundreds of research papers have been published related to ORB-SLAM2. ORB-SLAM2 works in a wide variety of environments, ranging from small hand-held sequences indoors to a car driven on the street. Inspired by PTAM (Parallel Tracking and Mapping) [21], ORB-SLAM2 uses ORB (Oriented FAST and rotated BRIEF) feature for feature extraction, which performs better than PTAM's FAST corner detection, especially in rotation. ORB-SLAM2 has three main parallel threads, as shown in 2.4, 1: tracking, local mapping, and loop closing. The tracking thread is considered to be the bottleneck of ORB-SLAM2 because it takes most of the time, and each new frame can not be processed until the current frame is completed. However, based on the nature of the FAST detection and ORB feature extraction, there are many tasks that can be parallelized and offload to GPU from CPU if the computing device has capable GPU on it like the Nvidia Jetson boards.

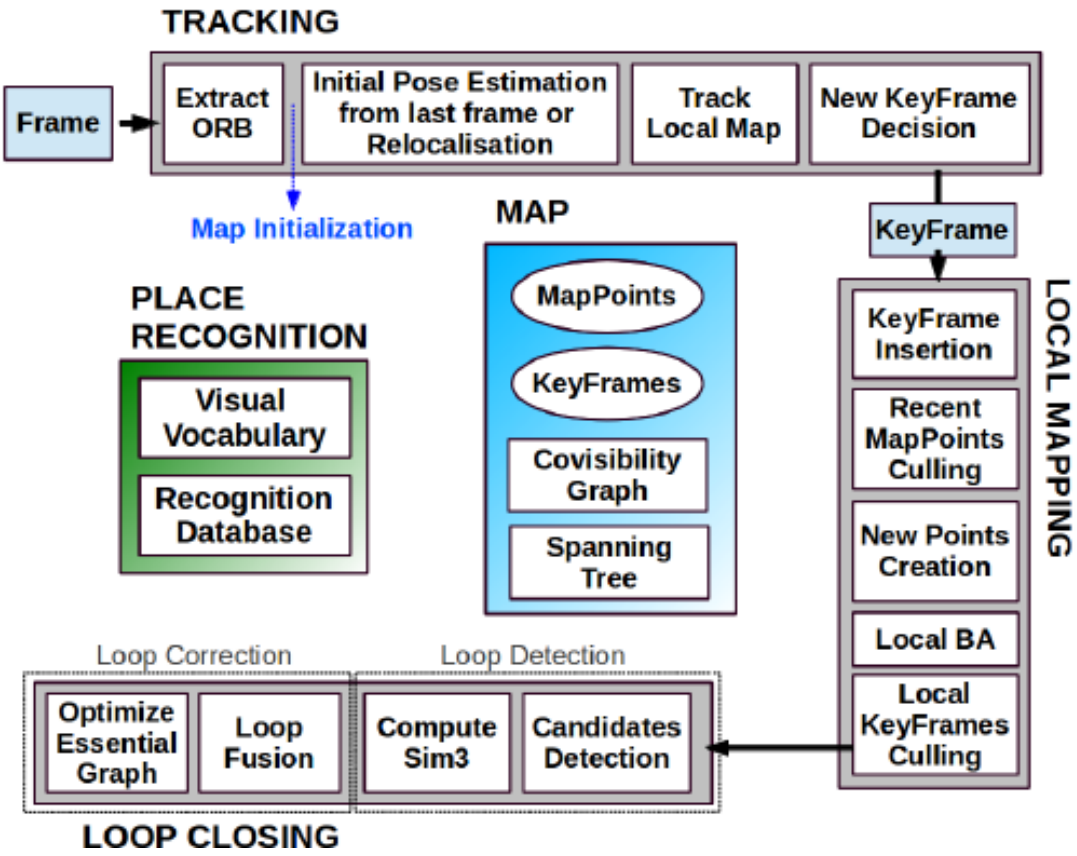


FIGURE 2.4: ORB-SLAM system overview. [20]

2.4 Processing platforms

Depending on the type of robot we have different types of processors. An assembly line robot or a small autonomous toy car uses a simple micro-controller because their tasks are limited. Since they have in-build memory and analog-to-digital converter makes them appropriate for compact and fast designs. On the other side of the spectrum, autonomous cars or humanoid robots operate with general-purpose CPUs with multiple cores and many layers of high-speed memory. For visual SLAM we need a great deal of processing power, therefore in this thesis, we will focus on high-performance processors.

2.4.1 The FPGA Perspective

Image processing is usually very computationally intensive due to complex algorithms and the sheer amount of data. For many applications, real-time image processing is difficult to achieve on a CPU. Hence, existing vision systems have dedicated hardware circuits such as ASICs, DSPs, FPGAs, or a

combination thereof, which can exploit the inherent parallelism of many image processing applications.

One popular FPGA implementation [22] offers up to four cameras input, fusion with IMU, and FAST feature extraction. The total FPGA speed-up is almost 3x times in comparison to stand-alone CPU. Similarly, [23] accelerate the ORB feature extraction, achieving up to 20fps for 640x480 input video.

2.4.2 GPU for computer vision

Symmetric multiprocessing (SMP) has been around since the 60s, and historically became mainstream architecture for parallelization. Almost every system today uses them and has Thread API like POSIX threads. Parallelization tools like Boost, OpenMP are built on top of the thread APIs. Since these tools were created, even more, abstract libraries were developed on top of them, making parallelization even easier. If you try to program using raw pthreads, CPU parallelization will not seem that easy.

When it comes to GPUs, historically they were used mainly for graphics, hence the name. Manual parallelization is not needed if you are doing common graphics tasks, you can just use libraries like OpenGL. Using general-purpose GPU programming for other tasks like HPC, and scientific computing is a relatively new trend. GPUs not only have different hardware architecture but are also designed for highly SIMD-oriented computing. This means the parallelization paradigm is not equivalent to that of CPUs. GPUs can perform specific tasks extremely fast, but they cannot perform a lot of other common programming tasks, especially ones with high branching uncertainty. This is why NVIDIA calls them accelerators, rather than CPU killers.

A modern CPU has special physical and logical parts that allow it to plan and enforce scheduling for all system components as well as generally higher single-core speeds (often upwards of 3.5–4GHz). This makes them good at performing single, complex math problems in a short time. Moreover, a CPU is equipped with multiple processing cores for parallel execution. A typical CPU has 2 to 8 cores and double the processing threads. Conversely, GPUs have low relative clock speeds (somewhere between 1–2GHz), but hundreds or even thousands of processing cores. In 2011 you had to be a Ph.D and needed to write complex CUDA code to accelerate numerical computing. Today, you can be a high school student and write a more efficient GPU accelerated program in 11 lines of Python code. There are many new GPU

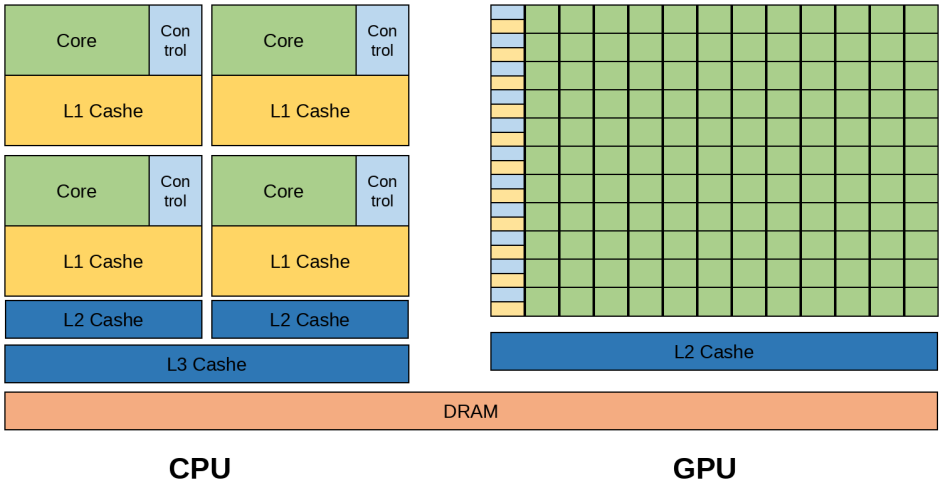


FIGURE 2.5: CPU and GPU simple architecture model.

acceleration libraries on the rise that cover more and more parallelization options, for example, CuPY.

To summarize, GPUs can't do all the tasks that CPUs can, but what they can do, they do it much faster. Because of their architecture differences, new abstractions needed to be developed specifically for GPU computing. Today, there are many tools you can use to parallelize/accelerate your code with ease, and in the future, fewer and fewer tasks will require manual low-level parallelization.

2.4.3 NVIDIA Hardware

Graphics Processing Units(GPUs) were originally designed to accelerate the graphics for computer games, a market segment in which NVIDIA has been quite strong for a long time. The gaming community kept growing and, along with it, so did NVIDIA. Now, GPUs became a widely used technology for many computer-aided procedures in different research or industry areas that require high algorithmic computation and large-scale calculations. Fields like bioinformatics, artificial intelligence, and computer vision involve processing very large datasets that are easily parreliazed in GPU.

NVIDIA in 2015 introduced the Jetson modules which are low-cost, low-power but powerful computers with integrated GPU. As shown in the table below, the cheapest and smallest is the Jetson Nano, then the TX series, and lastly the high-end Xavier modules. Due to their low weight and power consumption, they are widely used on robots to increase their computational robustness.

Jetson family is designed by NVIDIA to speed up machine learning applications with a low power system design, high-performance capability, and flexible form factors to build software-defined intelligent machines for a wide range of edge applications. These qualifications make the NVIDIA Jetson family ideal for all deep learning, AI, and visualization processes.

	Nano	**TX2**	**Xavier**
GPU	128-core Maxwell	256-core Pascal	512-core Pascal GPU with 64 Tensor cores
CPU	Quad-core ARM Cortex A57	Dual-core Denver 2 64-bit CPU and Quad-core ARM Cortex A57 MPCore processors	8-core NVIDIA Carmel ARM v8.2 64-bit CPU
Memory	4GB 64-bit LPDDR4 25.6 GB/s	8GB 128-bit LPDDR4 59.7 GB/s	32GB 256-bit LPDDR4 136.5 GB/s
Size & Power	69.6 mm x 45mm 5W I 10W	87mm x 50mm 7.5W I 15W	100mm x 87mm 10W I 15W I 30W

2.5 Computer Vision Software

Lastly, we will describe the software used in our system.

2.5.1 CUDA programming

Compute Unified Device Architecture(CUDA) is a parallel computing platform and programming model developed by NVIDIA for general computing on its own GPUs. CUDA lets developers use languages they are familiar with — C, C++, and Python — to build general-purpose applications for graphics processing units. CUDA Deep Neural Network library (cuDNN) is a library for deep neural nets built using CUDA. It provides GPU accelerated functionality for common operations in deep neural nets. You could use it directly yourself, but other libraries like TensorFlow already have built abstractions backed by cuDNN.

2.5.2 OpenCv

OpenCV is the huge open-source library for computer vision, machine learning, and image processing and now it plays a major role in real-time operation which is very important in today's systems. By using it, one can process images and videos to identify objects, faces, or even handwriting of a human.

It was introduced in 1999 by Intel Corporation and nowadays is the standard library for vision-based algorithms. Since 2010 CUDA module for OpenCV is available to speed up the algorithms using NVIDIA GPUs and achieving real-time performance.

Chapter 3

Theoretical Modeling

In this chapter, we break down the SLAM problem into its core functions. Widely accepted techniques for each sub-algorithm are described below. We also tested the functions using our own code, to verify their operation. All examples shown in this chapter were developed in a context in this thesis. For our tests we used up-to-date Python programming language.

3.1 Feature Extraction

Image features, such as edges and interest points, provide rich information on the image content. They correspond to local regions in the image and are fundamental in many applications in image analysis: recognition, matching, reconstruction, etc. Image features yield two different types of problems: the detection of an area of interest in the image, typically contours, and the description of local regions in the image, typically for matching in different images. In any case, they relate to the differential properties of the intensity function, for instance, the gradient or the Laplacian that are used to detect intensity discontinuities that occur at contours.

Every machine that loads an image stores it as a matrix. The size of the matrix depends on the number of pixels of the input image, also called image resolution. Every pixel has a value that describes how bright that pixel is, and what color it should be. Smaller numbers closer to zero represent black while larger numbers which are closer to 255 denote white. In the case of a colored image, we have three matrices, also called channels, with values from 0 to 255, describing the presence of the color in the image. The three-channel format represents Red, Green, and Blue (RGB) values for each pixel.

In most cases, we have images with many millions of pixels and want to extract specific points of interest we need for localization. One of the most

important features is the edges. Physical edges provide important visual information since they correspond to discontinuities in the physical, photometrical, and geometrical properties of scene objects. Since image intensity is proportional to scene radiance [24], physical edges are represented in the image by changes in the intensity function.

There are many methods of detecting edges [25]; the majority of different methods may be grouped into these two categories:

1. Gradient: The gradient method detects the edges by looking for the maximum and minimum in the first derivative of the image. For example Roberts, Prewitt, Sobel where detected features have very sharp edges.

2. Laplacian: The Laplacian method searches for zero crossings in the second derivative of the image to find edges e.g. Marr-Hildreth, Laplacian of Gaussian, etc. An edge has a one-dimensional shape of a ramp and calculating the derivative of the image can highlight its location.

3.1.1 Sobel-Feldman operators

In this paper, we will focus on Sobel-Feldman operators for edge detection. It is firstly introduced by Irwin Sobel and Gary Feldman in 1968 at Stanford Artificial Intelligence Laboratory. The Sobel-Feldman operator consists of two isotropic 3x3 kernels:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A \tag{3.1}$$

and

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \tag{3.2}$$

where **A** is the source digital image. Essentially Sobel-Feldman operators are trying to find out the amount of difference between a region of an image, firstly in the X-axis (3.1), and then in the Y-axis (3.2). Once we filter our image with X and Y operators we can calculate the magnitude:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3.3}$$

Lastly, using the above, we can calculate the corner direction with:

$$\Theta = \arctan \frac{G_y}{G_x} \tag{3.4}$$

Where, for example, Θ is 0 for a vertical edge that is lighter on the right side. Later on, we will use 3.4 to determine corners.

Algorithm 1 Sobel-Feldman edge detection

```
A= input(image)
B= rgb2gray(A)
[rows,columns]= size(B)
Sobelx= [1, 0, -1], [2, 0, -2], [1, 0, -1]
Sobely= [1, 2, 1], [0, 0, 0], [-1, -2, -1]
for x do=2 to rows
    for y do=2 to columns
        Gx(x,y) = B(x,y) * Sobelx
        Gy(x,y) = B(x,y) * Sobely
G(x,y) = sqrt(Gx^2 + Gy^2)
Output= G(x,y)
```

3.1.2 Canny Edge Detector

Canny edge detector [26] was developed by John F. Canny in 1986 and it is still widely used today in many different improved forms [27] or GPU accelerated [28] [29]. Canny edge detector simply takes as input, the output of Sobel-Feldman operator and makes it more useful. Here are the steps of the Canny edge detector:

1. Before we start any operation, we apply Gaussian blur to our grayscale image 3.2b 3.2c. Gaussian blur is used as a pre-processing stage in almost all computer vision algorithms [30]. The Gaussian blur feature is obtained by blurring (smoothing) an image using a Gaussian function to reduce the noise level. It can be considered as a low-pass filter and it is achieved by convolving an image with a Gaussian kernel. For our image, a 2-D Gaussian kernel is expressed as:

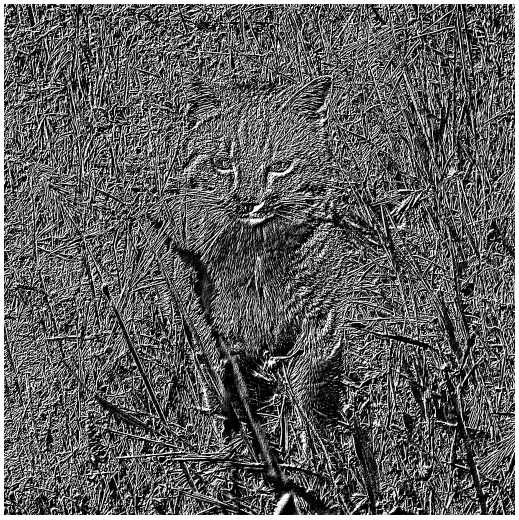
$$G(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.5}$$

   where σ is the standard deviation of the distribution and x and y are the location indices.

(A) Input image

(B) Sobel-Feldman X axis operator

(C) Sobel-Feldman Y axis operator



(D) Sobel-Feldman X Y axis operator

FIGURE 3.1: Sobel-Feldman operators

2. The next step is to use Sobel-Feldman operators to find the edge gradient strength and direction for each pixel. First the Sobel-Feldman operators 3.1, 3.2 are applied to the 3x3 pixel neighborhood of the current pixel, in both the x and y directions. Then the sum of each mask value times the corresponding pixel is computed as the Gx and Gy values, respectively. The square root of Gx squared plus Gy squared equals the edge strength 3.3. The inverse tangent of Gx / Gy yields the edge direction 3.4. The edge direction is then approximated to one of four possible values that make up the possible directions an edge could be in an image made up of a square pixel grid.

3. The following step is to trace along the edges based on the previously calculated gradient strengths and edge directions. Each pixel is cycled through using two nested for loops. If the current pixel has a gradient strength greater than its neighbor, then a switch is executed. The switch is determined by the edge direction of the current pixel. It stores the row and column of the next possible pixel in that direction and then tests the edge direction and gradient strength of that pixel. If it has the same edge direction and a gradient strength greater than a threshold we choose, that pixel is set to white and the next pixel along that edge is tested. In this manner, any significantly sharp edge is detected and set to white while all other pixels are set to black.

4. The final step is to find weak edges that are parallel to strong edges and eliminate them. This is accomplished by examining the pixels perpendicular to a particular edge pixel and eliminating the non-maximum edges 3.2d.

Canny edge detector aims to satisfy the three general criteria of edge detection

(A) Input image

(B) Gray-scale image

(C) Gaussian blur

(D) Canny edge detection

FIGURE 3.2: Step-by-step Canny edge detection

3.1.3 Harris Corner Detection

Harris Corner Detector [31] detection operator that is commonly used in computer vision algorithms to extract corners and infer features of an image. It was first introduced by Chris Harris and Mike Stephens in 1988 upon the improvement of Moravec's corner detector [32]. It is popular because it is rotation, scale, and illumination variation independent.

Corners in images represent critical information in describing object features, which play a crucial and irreplaceable role in computer vision and image processing. The difference between edge and corner is that the second has a significant change in intensity in both the X and Y-axis. For this reason, if we rotate the input image we can detect the same corners as before. Harris

Corner Detector uses Sobel-Feldman operators 3.1 to detect changes in light intensity. For this reason, steps 1,2,3 are the same as Canny edge detector 3.2d. The next steps are:

1. For each pixel in the grayscale image, consider a 3×3 window around it and compute the corner strength function. Call this its Harris value.
2. Detect all pixels that exceed a certain threshold and are the local maxima within a certain window. For each pixel, we also compute a descriptor.

The Harris value or score is described by the equation:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2 \tag{3.6}$$

where $E(u,v)$ is the Harris value, $w(x,y)$ is the pixel position and $I(x+u,y+v)$ is the intensity variation around the pixel.

(A) Input image

(B) Harris corner detector

3.1.4 Scale-Invariant Feature Transform

Scale-Invariant Feature Transform (SIFT) and was first presented in 2004, by D.Lowe, University of British Columbia [33]. The SIFT algorithm transforms the image into a collection of local feature vectors. These feature vectors are aimed to be distinctive and invariant to any scaling, rotation or translation of the image.

In the initial step, the feature locations are resolved as the local extrema of Difference of Gaussians (DOG pyramid):

$$D(x,y,\sigma) = (G(x,y,\sigma) - G(x,y,\sigma)) * I(x,y) = L(x,y,k\sigma) - L(x,y,\sigma \quad (3.7)$$

To carry out the DOG pyramid the information image is convolved iteratively with a Gaussian kernel:

$$G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} \exp^{-(x^2+y^2)/2} \quad (3.8)$$

This technique is rehashed as long as the down-sampling is conceivable. Every assortment of images of a similar size is called an octave. All octaves construct together with the alleged Gaussian pyramid, which is represented

by a 3D function:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{3.9}$$

The local extrema (maxima or minima) of DOG function are identified by contrasting every pixel and its 26 neighbors in the scale-space. The search for extrema prohibits the first and the last image in every octave since they don't have a scale above and a scale beneath individually. Scale-space extrema identification creates an excessive number of keypoint candidates, where some of which are temperamental and less helpful. In the subsequent stage, an itemized fit is performed to the close-by information to track down the precise area, scale, and proportion of head curves.

(A) Input image

(B) SIFT features with orientation

For every applicant keypoint, the interjection of the close-by information is utilized to accurately estimate its position. The insertion is finished utilizing the quadratic Taylor extension of the Difference-of-Gaussian scale-space function with the candidate keypoint as the origin. This Taylor extension is given as:

$$D(x) = D + \frac{\delta D^T}{\delta x} x + \frac{1}{2} x^T \frac{\delta^2 D}{\delta x^2} x, \tag{3.10}$$

Where $\boldsymbol{D}$ and its derivatives are evaluated at the candidate keypoint and $x = (x, y, \sigma)$ is the offset from this point.

In the following stage, for each keypoint, at least one orientation is appointed dependent on nearby image gradient directions. This is a helpful advance

in accomplishing invariance to rotations as the keypoint descriptor can be addressed comparative with this direction and subsequently accomplishes invariance to image rotation. To start with, the Gaussian-smoothed image $L(x, y, \sigma$ the scale is taken so all calculations are performed in a scale-invariant way. For an image sample $L(x, y)$ at scale σ, the angle extent, and orientation are precomputed utilizing pixel differences as:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + L(x,y+1) - L(x,y-1)} \quad (3.11)$$

$$\theta(x,y) = \tan^{-1} \frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y} \quad (3.12)$$

3.1.5 Speed-up Robust Features

There have been many attempts to improve SIFT algorithm. Speed-up Robust Features(SURF) [34] is inspired by SIFT descriptor and promises better results with a fraction of computational power. SURF is based on Hessian matrix for feature extraction and Haar wavelet responses for feature description

For feature point detection, the input image is convolved with the Gaussian kernel to obtain DOG pyramid 3.9, 3.7. The SURF operator uses the Hessian matrix to detect extreme points. For a point $X = (x, y)$ in an image I, the Hessian matrix $H(X, \sigma)$ of the point at scale σ is:

$$H(x,\sigma) = \begin{bmatrix} L_{xx}(X,\sigma) & L_{xy}(X,\sigma) \\ L_{xy}(X,\sigma) & L_{yy}(X,\sigma) \end{bmatrix} \quad (3.13)$$

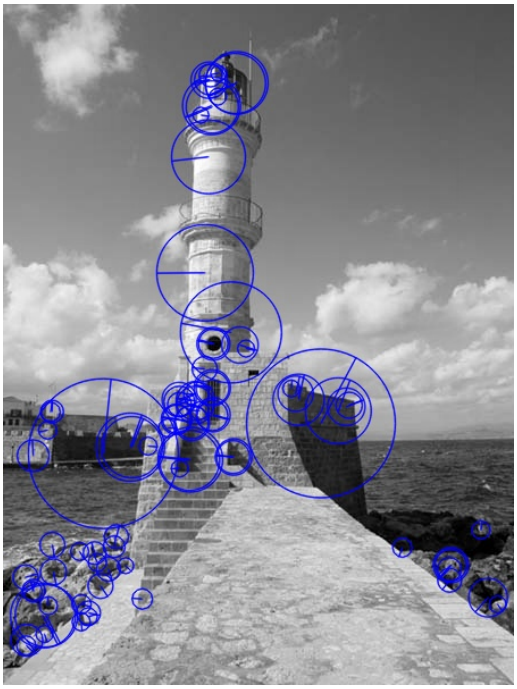
Where $L_{xx}(x, \sigma)$ indicates that the Gaussian first-order partial derivative is convolved with the image I at X and σ indicates the scale value at which the feature point is located. In order to accurately approximate the Gaussian kernel function, the H matrix discriminant is:

$$\det(H_{approx}) = D_{xx}D_{yy} - (wD_{xy})^2 \quad (3.14)$$

After using the Hessian matrix to find the extremum, the non-maximum suppression is performed in the 3×3×3 stereo neighborhood, only 9 of the upper and lower scales and 8 of the 26 neighborhoods around the scale are both large or small extreme points, they can be used as candidate feature points,

and then interpolated in scale space and image space to obtain stable feature point positions.

(A) Input image



(B) SURF features with orientation

Before we find the descriptor, we need to determine the orientation to ensure rotation invariance. SURF uses Haar wavelet responses in the horizontal and vertical direction for a neighborhood of size 6s, and then Gaussian weight coefficients are assigned to these response values, so that the response contribution to the feature points is large, thereby obtaining a series of vectors. The dominant orientation is estimated by calculating the sum of all responses within a sliding orientation window of angle 60 degrees.

Lastly, after determining the direction of the feature point a neighborhood of size 20sX20s is taken around the keypoint where s is the size. The descriptor window is divided into 4x4 sub-regions, and within each subregion, the Haar wavelet response in the range of 25s is calculated, and the Haar wavelet responses dx and dy are recorded in horizontal and vertical direction respectively. Then Gaussian weights are assigned to these responses, and finally, the response values of each sub-region and the absolute values of the responses are added to generate the feature vector V´ of the descriptor:

$$V' = (\sum dx, \sum dy, \sum |dx|, \sum |dy|) \tag{3.15}$$

3.1.6 Oriented FAST and Rotated BRIEF

Oriented FAST and Rotated BRIEF(ORB) was introduced by Ethan Rublee at Opencv labs in 2011. It is commonly used as a better alternative to SIFT and SURF due to its low computational cost. As the name suggests, it uses a slightly alternated FAST for keypoint detection and BRIEF for descriptor generator.

Features from Accelerated and Segments Test(FAST) compares the brightness of 16 pixels around a given pixel p. If more than 8 pixels are brighter or darker than the pixel p then it is selected as a keypoint.

Even though FAST has a very low cost compare to other methods, it doesn't offer orientation and multi-scale features. For this reason, ORB implements a resolution pyramid that detects features in different resolutions for the same image. After locating keypoints we set orientation to each point using intensity centroid. We expect that the corner is offset from its center and the vector is assigned for orientation. The coordinates of the centroid is computed by using moments:

$$m_{pq} = \sum_{x,y} x^p y^p I(x,y), \tag{3.16}$$

and the coordinates are calculated by:

$$c = (\frac{m_{01}}{m_{00}}, \frac{m_{01}}{m_{00}}). \tag{3.17}$$

The vector is constructed by the keypoint's center O to the centroid C with orientation:

$$\theta = atan2(m_{01}, m_{10}) \tag{3.18}$$

where $atan2$ is the quadrant-aware version of arctan. For descriptor, ORB uses rotation-aware BRIEF. The BRIEF descriptor is a bit string description of an image patch constructed from a set of binary intensity tests. Consider a smoothed image patch, p. A binary test τ is defined by:

$$\tau(p;x,y) = \begin{cases} 1 & :p(x) < p(y) \\ 0 & :p(x) \leq p(y) \end{cases} \tag{3.19}$$

where p(x) is the intensity of p at a point x. The feature is defined as a vector of n binary tests:

$$f_n(p) = \sum_{i \leq i \leq n} 2^{i-1} \tau(p;x,y) \tag{3.20}$$

In case we rotate the features slightly the performance of BRIEF falls off sharply. In that case, ORB suggests to steer BRIEF to the orientation of key-points. For any feature set of n binary test at location(x,y), we define a 2 x n matrix:

$$S = \begin{pmatrix} x1 & ..... & xn \\ y1 & ..... & y1 \end{pmatrix} \tag{3.21}$$

By using the patch orientation θ and the corresponding rotation matrix R_θ, we construct a steered version of S:

$$S_\theta = R_\theta S \tag{3.22}$$

Now the steered BRIEF operator is:

$$g_n(p, \theta) = f_n(p) | (x_i, y_i) \epsilon S_\theta \tag{3.23}$$

(A) Input image

(B) ORB features extraction

3.2 Feature Matching

Feature matching or feature correspondence serves as a core technique for image analysis and understanding. There is a wide range of applications that are closely related to it, such as object recognition, image retrieval, 3D

reconstruction, image enhancement, and so on. The problems of correspondence involve clutter background, a significant, amount of outline and occlusion. Moreover, multiple translations, orientations, and deformations also negatively affect the matching of features in terms of precision, recall, and efficiency.

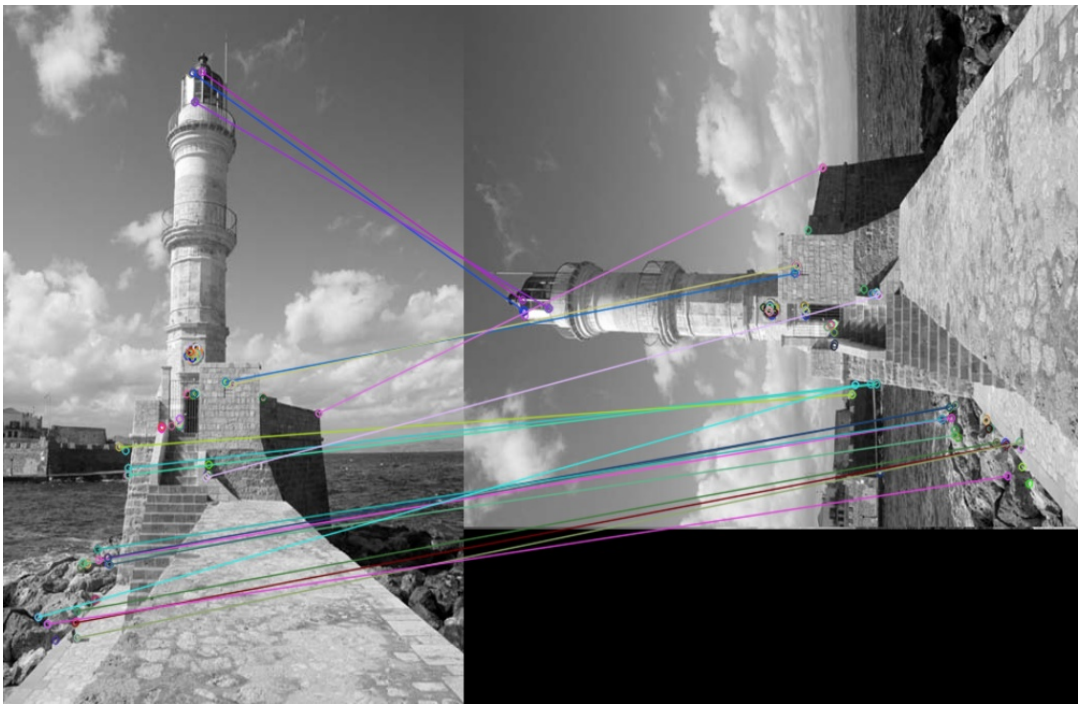


FIGURE 3.7: Matching ORB features between an image and a rotated version of the same image.

3.2.1 Brute Force Matching

Brute force matching is a simple algorithm for feature matching. It uses one descriptor from the image and calculates the Hamming or Euclidean distance for the descriptors of the second image. Binary bit-string descriptors like BRISK and ORB, utilize Hamming distance with very fast execution but low robustness. On the other hand, SIFT and SURF use Euclidean distance between descriptors and they are more robust but require more computational time. Take the two descriptors K1 and K2 from the descriptors obtained from feature extraction:

$$K_1 = x_0, x_1, x_2, ...x_{255} \quad K_2 = y_0, y_1, y_2, ...y_{255} \tag{3.24}$$

We apply XOR operation on the Hamming/Euclidean distance to determine the similarity degree of feature descriptors:

$$D(K_1, K_2) = \sum_{i=0}^{i=255} x_i \oplus y_i \tag{3.25}$$

The smaller the $D(K_1, K_2)$ the bigger the similarity between the two descriptors K_1, K_2. When the similarity reaches about 50% then it is considered to be similar.

3.2.2 FLANN Based Matcher

FLANN stands for Fast Library for Approximate Nearest Neighbors. It efficiently searches an M-dimensional data-set of ND points to find, approximately, the nearest neighbors to a set of NQ query points. For a general dataset, the only way to compute the exact nearest neighbors requires, for each of NQ query points, the computation of the distance to each of the ND data points, requiring NQ*ND such calculations. There is no known method that can significantly decrease the cost of this brute force approach. FLANN offers an approach that significantly speeds up the computation, at the price of only being able to guarantee that the results are approximate. The user specifies the degree of approximation that is acceptable.

FLANN consists of many algorithms for the nearest neighbors problem. The user inputs the features data-set and the desired degree of accuracy and FLANN will automatically choose the best algorithm for each occasion. The optimal algorithm for fast approximate nearest neighbor search is highly dependent on several factors such as the structure of the dataset(whether there is any correlation between the features in the dataset), the size, and desired precision.

3.3 Three Dimensions Reconstruction

In this section, we explore methods used to obtain three-dimension information from images. Given that we already extracted and matched features, we proceed to the localization and mapping.

3.3.1 Camera Calibration

Camera calibration is the process of estimating the parameters of a camera model. These parameters are the mathematical description of how a camera projects a 3D point of the world to the 2D plane of the image. Typically the camera model parameters are divided into two categories: ***Extrinsic*** and ***Intrinsic***.

Extrinsic describe where is the camera in the 3d world. We have the location or position matrix:

$$X_o = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.26}$$

and the orientation, or where the camera is looking, matrix:

$$\theta = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \tag{3.27}$$

So if we want to perform camera localization we have to calculate a 6 degree of freedom or 6 dimensional vector:

$$6Dvector = \begin{bmatrix} X \\ Y \\ Z \\ \alpha \\ \beta \\ \gamma \end{bmatrix} \tag{3.28}$$

.

We calculate the Extrinsic by finding the projection center of our camera. For the pinhole camera model, the projection center is the point where all rays intersect.

Intrinsics are the parameters that "sit" inside the camera. It describes how a point in the 3D world is mapped onto the 2D image plane, assuming that the camera sits in the origin and has zero orientation. To describe the intrinsic we use at least 4 or 5 parameters, depends if we have a digital or analog camera. The intrinsic parameters are:

1. The camera constant c, which is the distance of the image plane to the production center.
2. The scale difference m, in x and y. Depending on the literature sometimes one also uses focal length x fx and focal length in the y fy as two parameters, but they are basically equivalent
3. The principle point $P(x_H, y_H)$ is the pixel in the image, through which the optical axis of the camera passes. Usually, the principle point is somewhere near the center of the image but of course not precisely because the camera chip is not precisely glued into the camera lens.
4. In the case of an analog camera, we have the Sheer, S, parameter which is near zero for the digital cameras.

Using the parameters from Extrinsic and Intrinsic we can describe with a mathematical model how a point from the 3D world is mapped onto the image plane. This procedure is called Direct Linear Transform (DLT). DLT is an 11 degree of freedom transformation, taking the 6 parameters from extrinsic and the five parameters from intrinsic. DLT is an approximation because it assumes that we have a fine camera model or a camera with no distortions involved. We can compute DLT with 6 control points, which helps us estimate the intrinsic and extrinsic parameters.

In practice, we have additional non-linear parameters involved for lens distortions like barrel or pincution distortion.

Having all the above parameters we can map with high precision every point from 3D world to 2D image plane using:

$$x = PX \tag{3.29}$$

where x is the 2D pixel coordinates, X is the 3D world coordinates and P is the matrix which includes Intrinsic, Extrinsic, and distortion parameters.

3.3.2 Zhang's camera calibration method

Zhengyou Zhang developed a camera calibration algorithm using images of chess patterns [35]. The algorithm computes the elements of its 3x3 matrix K, which consists of 5 intrinsic parameters of a camera: c,m,x_H, y_H,s. We assume that we have a calibration pattern in the Z=0 of the world coordinates and we are taking images of this pattern from different viewpoints.

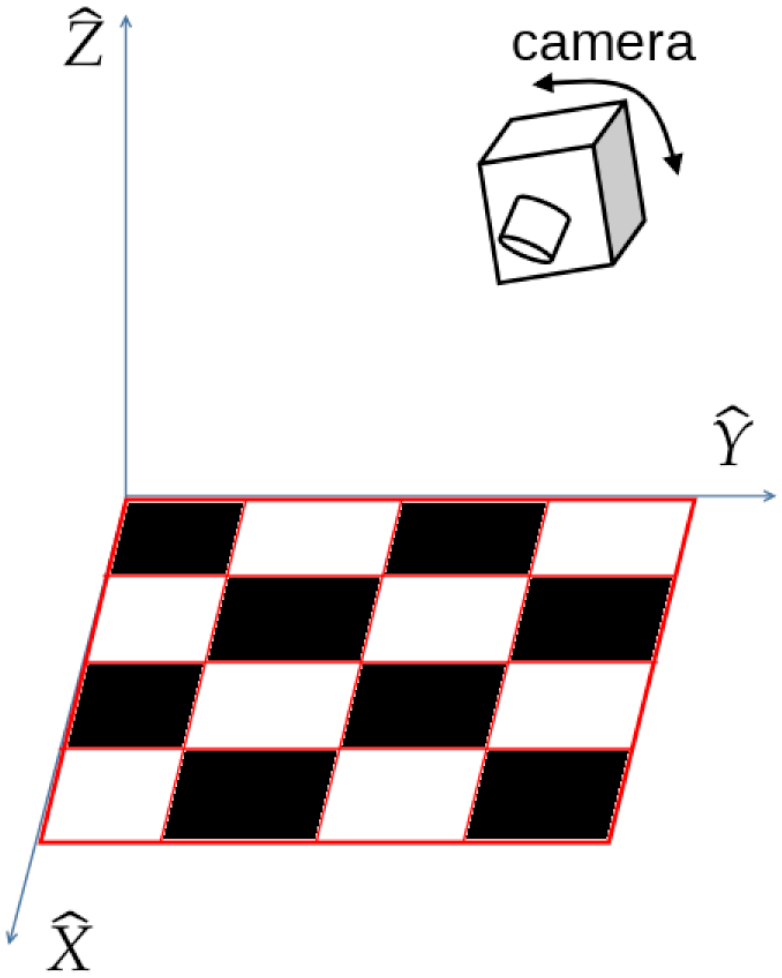


FIGURE 3.8: Zhang's calibration

For every observed camera point x:

$$x = KR[I_3 - X_o]X \tag{3.30}$$

Where R and X_o are the rotation and translation in xyz coordinates, and X is the control point in the world.

The matrix form of the equation is:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} c & cs & x_H \\ 0 & c(1+m) & y_H \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.31}$$

Since we assume that Z=0 we can simplify the above:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} c & cs & x_H \\ 0 & c(1+m) & y_H \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{3.32}$$

Similar to DLT we can estimate a 3x3 homography, by exploiting at least 4 points in our images. This provides an estimate of H:

$$H = [h_1, h_2, h_3] = \begin{bmatrix} c & cs & x_H \\ 0 & c(1+m) & y_H \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix} \tag{3.33}$$

Given the homography H we can compute K in four step procedure:

1. Exploit constrains like:

$$[r_1, r_2, t] = K^{-1}[h_1, h_2, h_3] \Longrightarrow r_1 = K^{-1}h_1 \quad \text{and} \quad r_2 = K^{-1}h_2 \tag{3.34}$$

$$r_1^T r_2 = 0 \tag{3.35}$$

$$||r_1|| = ||r_2|| \tag{3.36}$$

2. Define a matrix $B = K^{-T}K^{-1}$.
3. The matrix B has 6 unknowns that we can obtain from 3 different camera positions.
4. Solve the equations using linear least-squares method.

3.3.3 Random Sample Consensus

Random Sample Consensus(RANSAC) is a try-and-error approach to group data points into inliers and outliers. Sensor data will always be imperfect, and quite often the data points will not be explained to the real world. This will result in data associations between two images that are incorrect (outliers) and negatively affect the localization result.

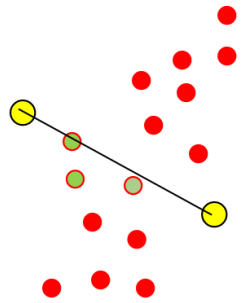
RANSAC is a 3 step procedure:

1. **Sample**: We sample a subset of data points and we consider them to be inliers.
2. **Compute**: We use the sample subset and compute the model.

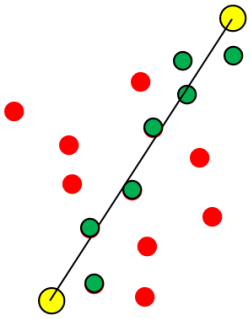
3. **Score**: We calculate a score for the sample by checking how many of the remaining data points will support this model.

This process is repeated many times and we use the solution with the highest score.

A simple 2D example in shown in 3.9:



(A) Sample with inline score=3



(B) Sample with inline score=7

FIGURE 3.9: 2D RANSAC example

3.3.4 Structure from motion

It is impossible to recover the 3D coordinates from a single image due to the loss of depth information. One of the solutions is to use multiple views of the same scene. By extracting information from one view and matching it to the other, slightly altered perspective, we can estimate with high precision the depth. This technique is called structure-from-motion and it uses epipolar geometry to find the depth of an image.

As shown in 3.10, we have P which is a real-world point mapped in P_0 at the image plane of camera C_0 and P_1 at the image plane of camera C_1. The vector or the line connecting the cameras C_0 and C_1 is called the epipolar axis and the points, e1 and e1, that meet each plane are called epipoles. P_1 lies

somewhere in the line P - P_0 and all the possible positions of P_1 create a line called epipolar line. In that case, if the P_0 is known we only need to search for P_1 in a 1D line instead of a 2D image.

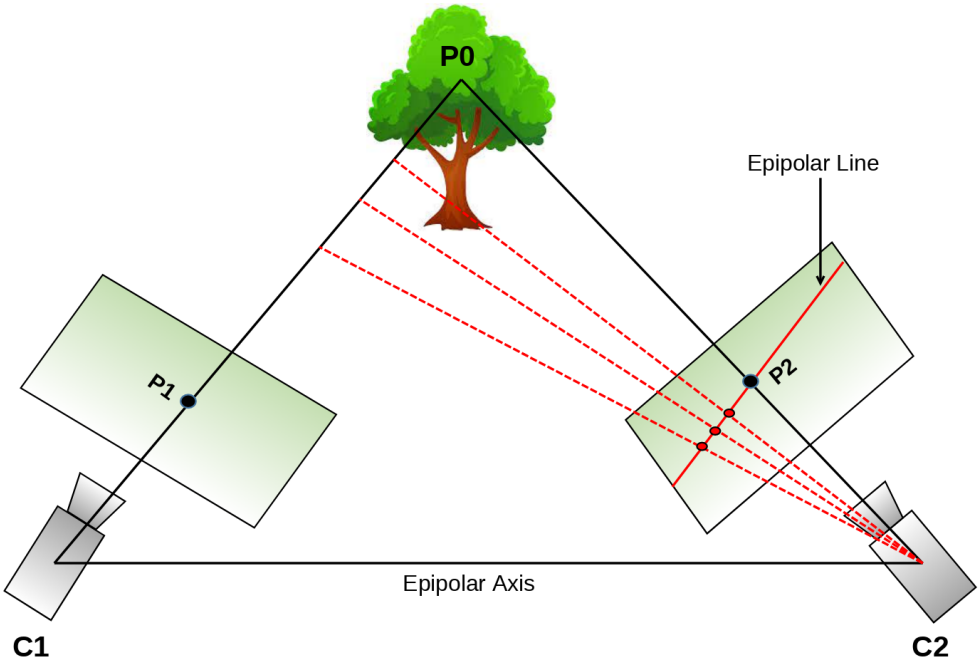


FIGURE 3.10: Epipolar Geometry

In the case of calibrated camera, we know to location and orientation of a moving camera. With the epipolar geometry, we can calculate the 3D coordinates of a real-world point with a low computational cost. This procedure is also called triangulation.

Chapter 4

System Architecture

The current development of visual SLAM has been relatively mature, and there are various types of solutions, including sparse method, semi-dense method, and dense method, as well as feature point method based on image features and direct method based on image grayscale. The execution efficiency, positioning accuracy, and robustness of these algorithms perform well in specific experimental environments. However, most of these algorithms are performed on desktop-level high-power platforms, and there is very little work to solve visual SLAM problems for embedded platforms.

Our work mainly studies how GPU parallel computing can accelerate processing for high-resolution videos input. Although there has been sufficient research on parallel computing to accelerate certain parts of SLAM [36] [37], most of them performed on desktop GPUs.

In summary, we modeled our system with certain goals:

- An efficient SLAM algorithm working in both PC and GPU embedded platforms.
- Tuned and optimized to perform best for high-resolution videos with as many as features possible.
- Fast enough to work in "real-time" scenarios.

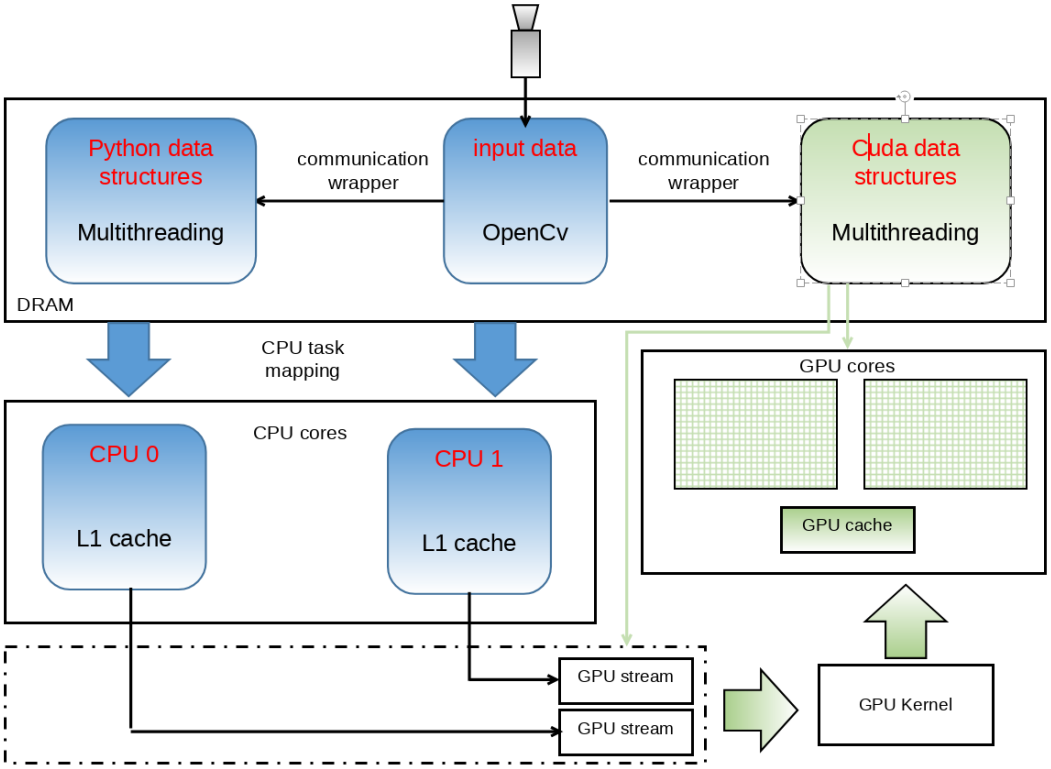


FIGURE 4.1: Jetson Tx2 hardware design.

4.1 vSLAM CPU implementation

We divided the vSLAM into 6 steps:

1. **System initialization**: The first step is to load SLAM parameters, calibrate the camera and initialize the CPU threads. Lastly, we read the input frame in a loop.

2. **Feature extraction**: For this step, we have many options. As shown in the previous chapter, ORB is the newest successor of all the other feature extraction methods. ORB take improves upon the fastest methods for keypoint detection and descriptor generation, making them more efficient while maintaining their speed advantage.

3. **Feature matching**: Feature matching is the second most computation-heavy part of our vSLAM. Here we have two options: Brute force or FLANN matching. Depending on the size of the features the results variate. Brute force matching is usually faster for a small number of features while the FLANN is better for a large number. Using GPU to accelerate our system, we found out that Brute force benefits more from parallel execution.

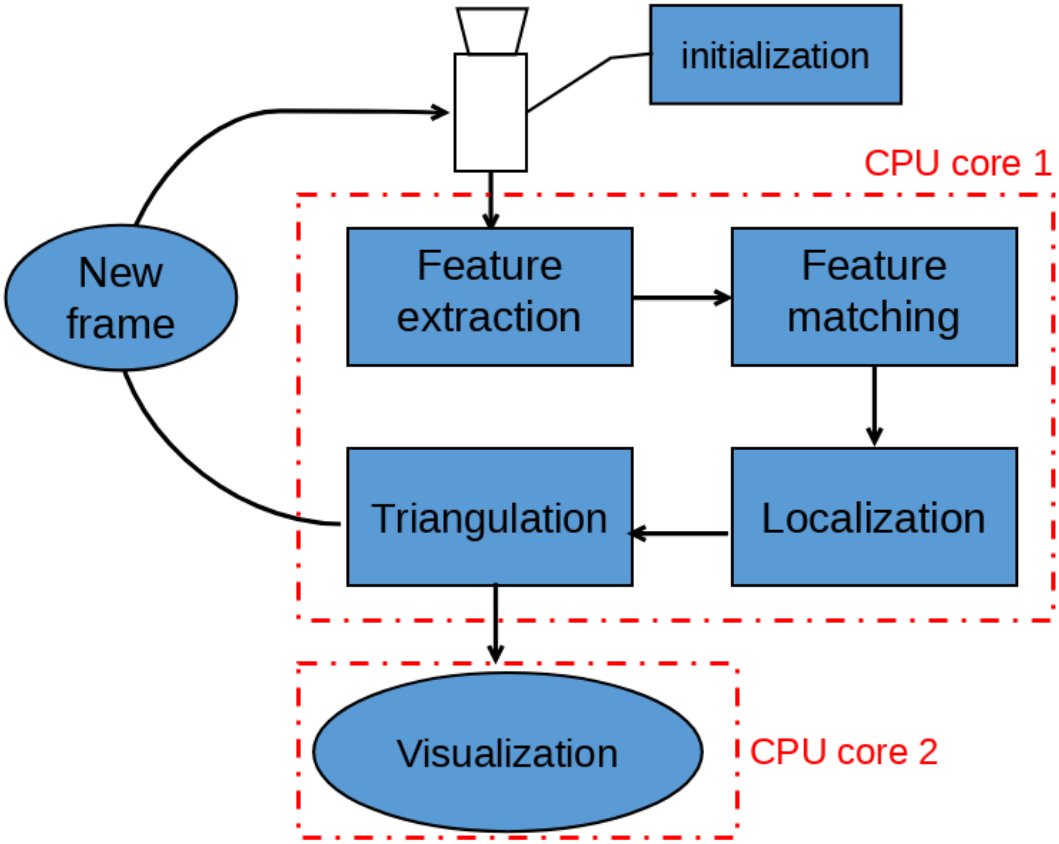


FIGURE 4.2: Using a second CPU core for visualization.

4. **Localization**: For this step, we use random sample consensus(RANSAC) algorithm to localize our camera through the environment.

5. **Mapping**: We use epipolar geometry to obtain the three-dimension coordinates of features points.

6. **Visualization**: For the last step we use a python fork of open source library, Pangolin, to visualize both the camera position and the keypoints, in a 3D environment.

Feature extraction and matching are the main focus of this work. They consume more than half of the computing resources and are particularly suitable for parallelization. Initialization runs once at the start of our system. After we obtain camera ***Intrinsics*** and distortion parameters, we just loop the video frames. Localization and Mapping consist of many branches and loops making them applicable for CPU processing.

For visualization, we dedicate a separate CPU core. We visualize the position and keypoints in a 3D environment without requiring synchronization with

the rest of the steps. For this reason, we "isolate" the visualization part and we only associate it with the three-dimension coordinates of the SLAM output.

4.2 GPU acceleration

There is an inherent overhead in the GPU processing flow due to the transfer of the images between the CPU and GPU memories. Such overhead can be minimized if all the processing operations are performed in the GPU, and only the initial and final images are transferred:

$$T_{overhead} = T_{upload} + T_{download} \tag{4.1}$$

A speed gain will be obtained if and only if:

$$T_{CPU} > T_{overhead} + T_{GPU}, \tag{4.2}$$

where T_{CPU} and T_{GPU} are the execution times for CPU and GPU respectively.

To determine which parts are worth accelerating, we experiment on some videos for the CPU model. For 1920x1080 pixel images and 500 features total, the average time of every step is:

	Desktop CPU Average time	**Dependence**
Initialization	7-10 ms	Image resolution, Image format
Extraction	100-110 ms	Image resolution
Matching	32-40 ms	Number of features
Localization	28-30 ms	Number of matches
Mapping	15-20 ms	Number of features

TABLE 4.1: Average desktop CPU execution time for every SLAM step and what parameters affect it.

In this work, we accelerated some parts of ORB feature extraction and Brute force matching.

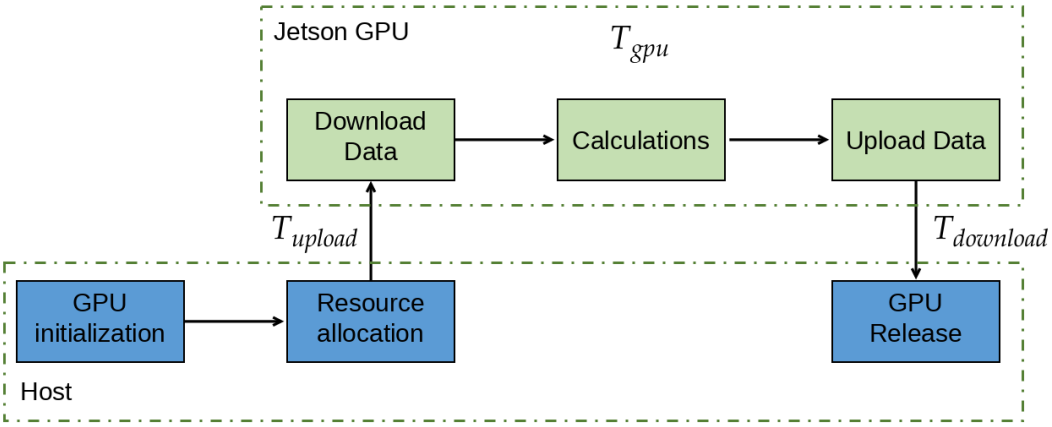


FIGURE 4.3: CPU to GPU data flow.

4.2.1 GPU implementation of oFAST

After we read a frame from our camera the first step is to build a Gaussian pyramid based on the original image first. The next step is to extract feature points and vectors from each Gaussian level of the pyramid. This step provides scale-invariant to our features. Lastly, all the keypoints will be mapped back to the original image. This step will result in keypoints being too dense and repetitive. Consequently, it is crucial to remove duplicate feature points by performing non-maximum suppression algorithm.

Gaussian pyramid. Creating a Gaussian pyramid is a repeated two-step process. Firstly we apply a Gaussian blur to our image and we sub-sample to half resolution. This process is repeated up to 4 times. As disgusted before(3.1.6), there are two benefits to performing this method. Firstly, Gaussian blur removes the camera noise, and secondly by sub-sampling the image the feature points have orientation and are multi-scaled. Performing Gaussian blur we only need data from the neighboring pixels, depending on the kernel size of the blur filter. Moreover, every level of the pyramid has no data association from the previous one, meaning we can execute Gaussian blur to all levels in parallel.

FAST features. The next step is to perform FAST feature detection in each image layer of the Gaussian pyramid. FAST algorithm compares the pixel intensity with its neighbors and calculates a score. If the score is above the threshold we require, then this pixel is a feature point. Like the previous step, we only need data for the neighbors of every pixel and all the pyramid layers are independent.

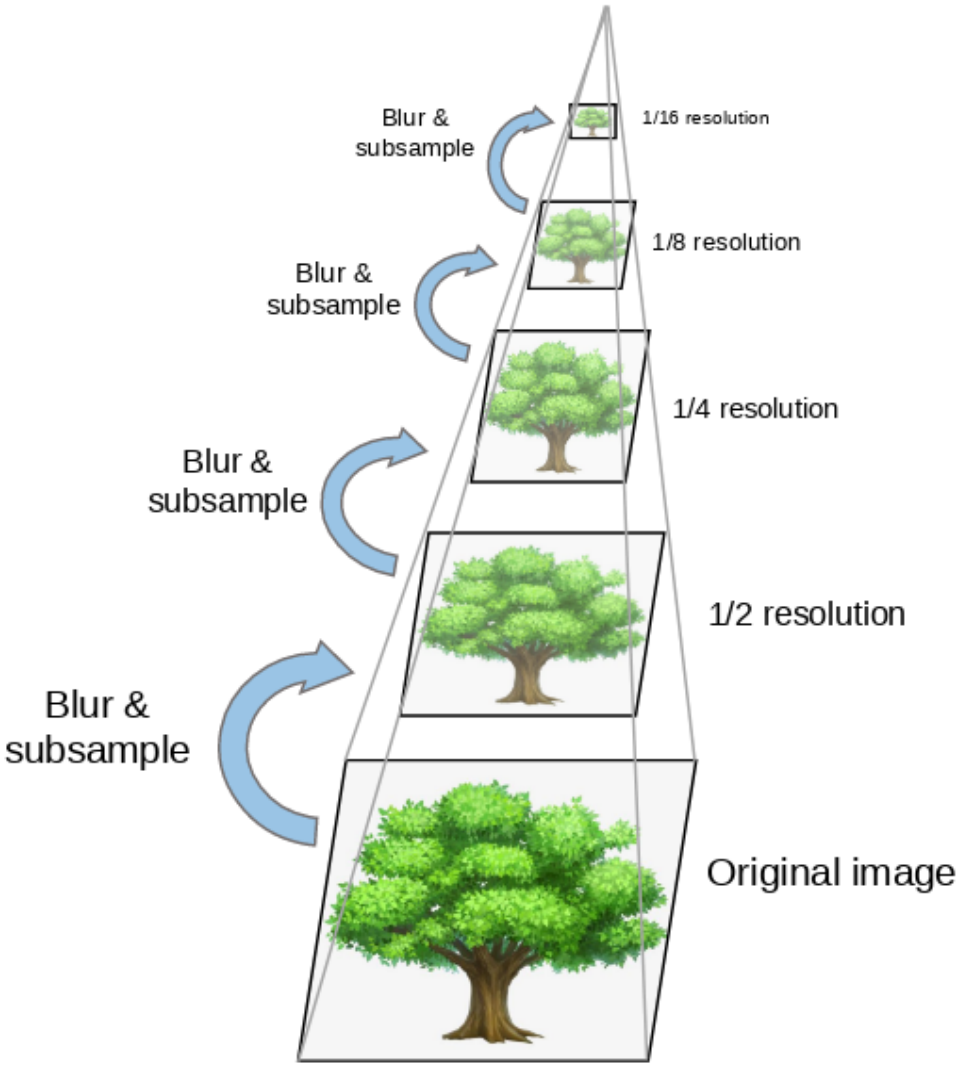


FIGURE 4.4: Building Gaussian Pyramid.

Coordinate normalization. All the feature points of every pyramid level must be mapped to the original image. The end result consists of many duplicate feature points where we must choose the one with the biggest score. Lastly, we perform Non-maximum suppression in order to spread the features points in the image. Every point is compared to all the adjacent points and we select the one with the highest response value.

After further analysis of the ORB feature extraction we concluded that:

1. There is no data communication among the Gaussian layers during the FAST feature detection. Therefore we can perform FAST to each layer independently, which can be parallelized.

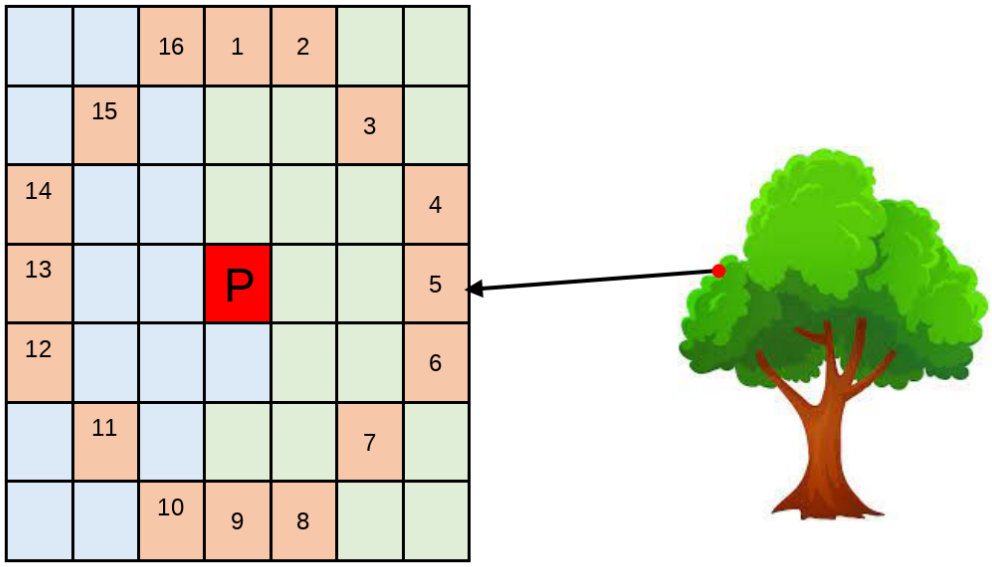


FIGURE 4.5: FAST feature detection illustration. We compare the intensity of pixel P with the surrounding 16 pixels to determine if it is a feature point.

2. For every pixel that we apply FAST, we only need data from its surrounding neighbors.

3. Non-maximum suppression has many branches and loops making it optimal for CPU to calculate.

Algorithm 2 Feature extraction using CUDA.

Input: Image frame.
Output: A list of FAST features.

1: $img \leftarrow readInput$;
2: cudaMemCopyHostToDevice(img);
3: **for** $i \leftarrow 0$ to PyramidLevel **do**:
4: $\quad layerMem[i] \leftarrow gpuBuildGaussianPyramid(img)$;
5: $\quad Pt[k] \leftarrow gpuFastFeatureExtraction(layerMem[i])$;
6: $\quad$ cudaMemCopyDeviceToHost(Pt[k]);
7: $\quad Pt[y] \leftarrow nonmaximumSuppression(Pt[k])$;
8: $Pt[y] \leftarrow nonmaximumSuppression(Pt[k])$;
9: $Output \leftarrow Pt[y]$;

The first way to speed up the GPU time is to properly allocate the computational resources, for parallel execution. When a CUDA program is running on the GPU, the user has to arrange the size of thread blocks and grids, and from there the kernel creates the threads into the streaming multiprocessor (SM) of the GPU. In our case, because the images are high resolution we can

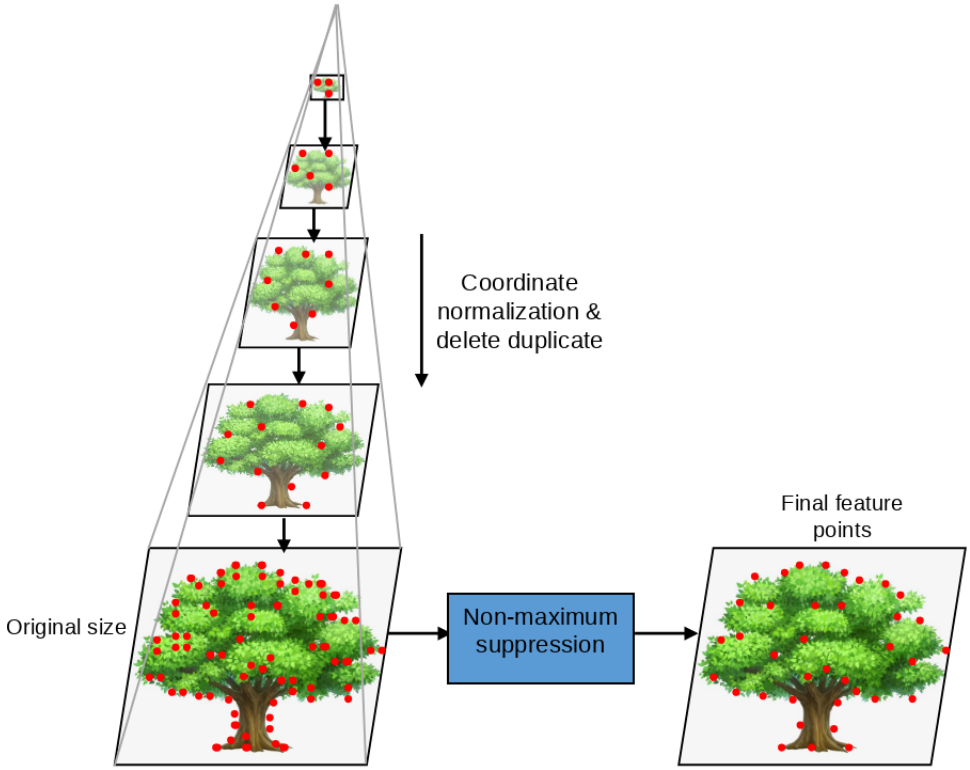


FIGURE 4.6: Coordinate normalization and non-maximum suppression

use more threads in parallel resulting in more performance boost from the SMs.

After testing multiple thread arrangements, we ended up with one thread per 4 pixels. The thread block is two-dimensional with a size of 32x8. Lastly, the grid size a dynamic size to match the resolution size of the image. For example, the original size of our input image is 1920x1080 pixels. The grid size is:

$$Grid_{feature} = \frac{img_x}{block_x} \times \frac{img_y}{block_y} = \frac{1920 pixels}{8 blocks} \times \frac{1080 pixels}{32 blocks} = 240 \times 34 \quad (4.3)$$

Allocating fewer threads in the thread block, can increase the idle time of the SM when accessing global memory. On the contrary, allocating more threads will result in extensive computational demands that the register memory can not handle, as a result some data will stored in global memory with higher latency.

Finally, we calculate the orientation of FAST features. Feature points requires data from the neighbor pixels to be involved in the calculation. For each

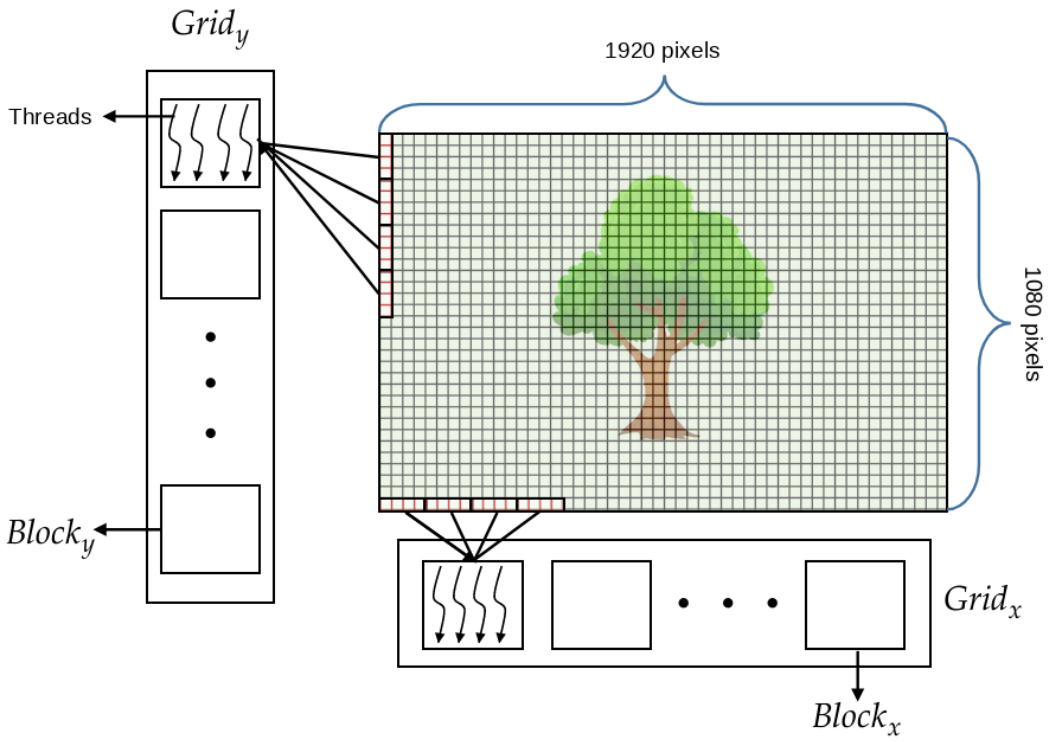


FIGURE 4.7: GPU computational resources allocation

feature point, we assign 32 threads to calculate the orientation information. Thread block size is set to 32x8 and the grid has the size of the maximum number of feature points, which can be altered by the user. Every frame has a diverse number of feature points where we allocate computational resources based on the maximum number of feature points. In case of fewer points, the allocated threads are empty.

4.2.2 GPU implementation of BRIEF

Calculating BRIEF descriptors for our feature points is a straightforward procedure with low computational cost, but very power full way to describe points in an image. As depicted in 3.1.6, we compare the intensities of fixed pairs of pixels and we generate a bit stream for each feature point. In BRIEF case we generate 32 bytes for every descriptor.

Similar to feature orientation the size of GPU grid is fixed to the maximum number of feature points. The thread block is set to two dimensions with a 32x8 size and for every keypoint we allocate 32 threads. The thread allocation scheme is the same as the feature orientation: 32 threads for each feature point to calculate orientation/descriptor, 8 features per block, and the grid

size is fixed to the maximum number of feature points.

$$Grid_{matcher} = \frac{maxFeautures}{block_y} \tag{4.4}$$

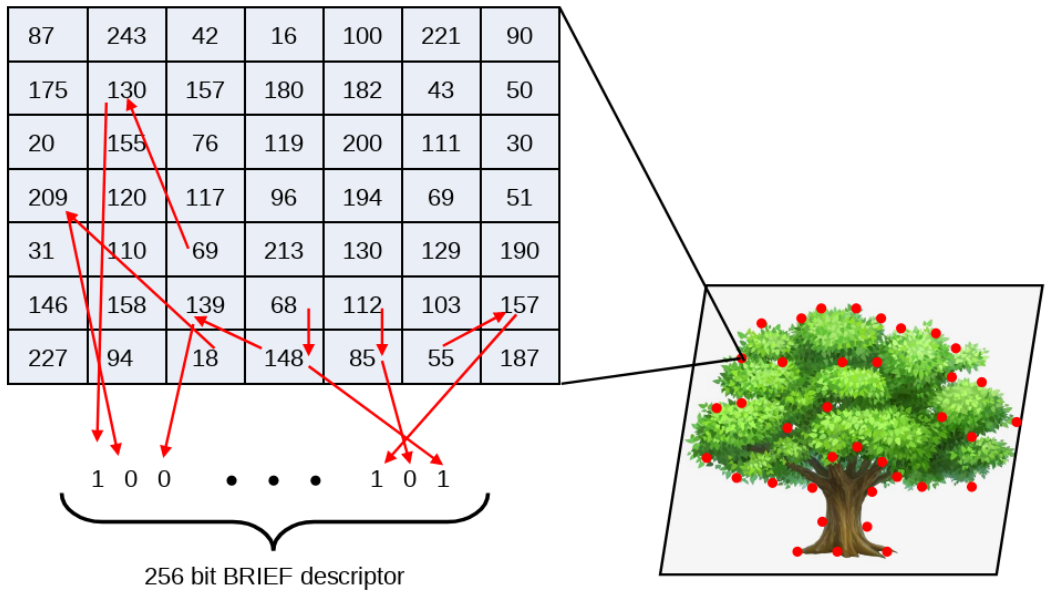


FIGURE 4.8: Comparing pair intensities around the feature point to create BRIEF descriptor.

4.2.3 GPU brute force matching

Feature matching is the second most time-consuming part of SLAM algorithm. After we extracted features from an image we compare the descriptors with a list of the previous feature points. While visual SLAM runs the list of features will gradually grow, increasing the complexity of matching. In order to reduce the computation time, we perform local point selection, meaning we only choose points that are more likely to match depending on the frame order.

In our case, the descriptors of feature points are binary, meaning that the feature matching is trivial and fast to compare. Every BRIEF descriptor is 256 bits long and compares the Hamming distance between them:

$$d_{Hamming}(B_1, B_2) = sum(xor(B_1, B_2)). \tag{4.5}$$

Therefore we allocate 256 threads for each block to compute in parallel the Hamming distance between descriptors. The thread grid is one dimension

Algorithm 3 Feature matching using CUDA.

Input: Query and train descriptors.
Output: Feature Matches.

```
cudaMemCopyHostToDevice(queryDes);
cudaMemCopyHostToDevice(trainDes);
for i ← 0 to numQuery do:
    for y ← 0 to numDes do:
        distanceTemp ← calcHammingDistance(queryDes[i], trainDes[y]);
        if distanceTemp ≤ distance then:
            distance ← distanceTemp;
            tempMatch[i] ← gpuPointsInformation();
        Matches[i] ← keepGoodMatches(tempMatch[i]);
Output ← Matches;
```

with the size of the feature list from previous feature points, divided by 256. Usually, the size of the list is fixed, depending on the maximum feature points we extract in every frame, in order to not overflow the global memory in case of long SLAM.

$$Grid_{descriptor} = \frac{maxFeautures}{block_y} \tag{4.6}$$

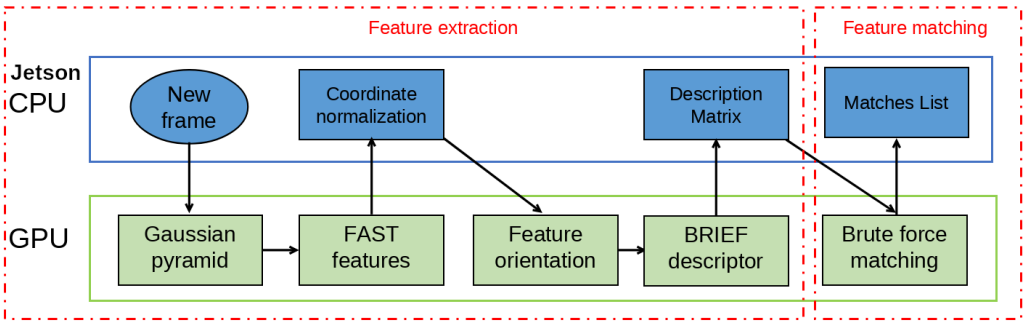


FIGURE 4.9: A graph depicting every sub-step of feature extraction and matching.

4.2.4 Task allocation timing

We can further optimize the parallel execution of SLAM, by reducing the idle time of both CPU and GPU. They operate asynchronously, and by adjusting the task allocation we can use the CPU while waiting for results in GPU. As stated before, non-maximum suppression is better operated in CPU due to it has a lot of branches and loops. Moreover, each level of Gaussian pyramid is independent, meaning we can perform feature detection while building the Gaussian pyramid meaning we can exploit more block threads. We can

also download each layer's results to the CPU to decrease the idle time of the CPU while GPU performs feature detection. Therefore here are the steps for task optimization:

1. For each layer of Gaussian pyramid we perform feature detection before moving to the next layer.
2. Each layer's results are sent back to the CPU to calculate non-maximum detection.
3. When the Gaussian pyramid and feature detection is finished, the CPU has only the last layer of the pyramid to calculate. Then the results are sent back to GPU for calculation of feature orientation.

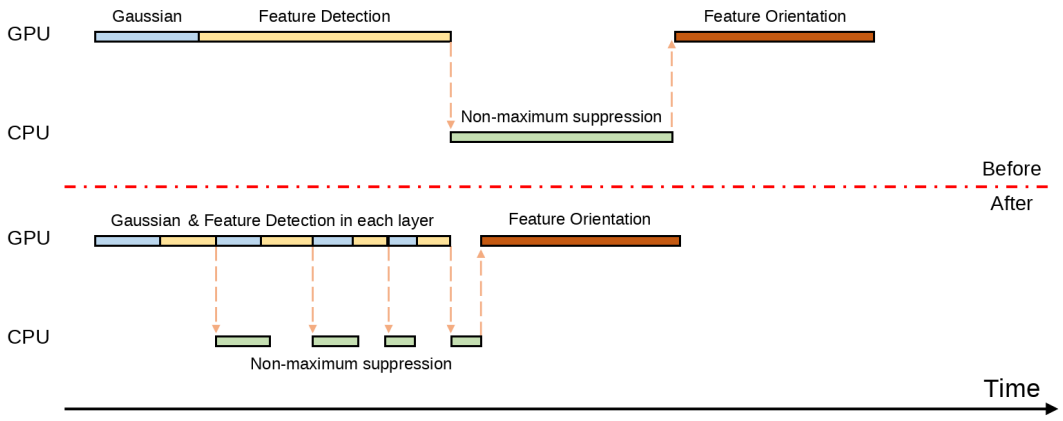


FIGURE 4.10: Before and after comparison of task allocation.

Chapter 5

System Verification and Performance Evaluation

5.1 Specifications of platforms

This study is an attempt to accelerate visual SLAM with NVIDIA GPU systems and specifically the Jetson TX2 embedded system. Moreover, we will test the same approach on a Desktop PC with far more GPU cores and CPU speed. The desktop test aims to prove that it works on different systems and occasions like remote sensing.

5.1.1 Jetson TX2

The NVIDIA Jetson series are embedded computing boards known for their low weight and power consumption. What distinguishes the Jetson series from other computing boards, is that it includes a GPU processor. Jetson TX2 is the middle-performance board between the low power-performance Jetson Nano and the high-end Jetson Xavier.

	NVIDIA Jetson TX2
GPU	256 NVIDIA Cuda cores, Pascal architecture
CPU	Dual-cor Denver 2 and quad-core ARM A57 complex CPUs
RAM	8 GB 128-bit LPDDR4
Storage	32 GB eMMC 5.1
Size	87mm X 50mm
Power	7.5W/15W

TABLE 5.1: NVIDIA Jetson TX2 specifications(Link).

Moreover, Jetson TX2 has 5 operation modes where we will test most of them. Every operation mode configures different CPU cores and clock speeds on 7,5/15W power consumption.

Property	Max-N (Mode 0)	Max-Q (Mode 1)	Max-P Core-All (Mode 2)	Max-P ARM (Mode 3)	Max-P Denver (Mode 4)
Denver 2 cores/freq (Ghz)	2 / 2	N/A	2 / 1.4	N/A	1 / 2
ARM A57 cores/freq (Ghz)	4 / 2	4 / 1.2	4 / 1.4	4 / 2	1 / 0,34
RAM freq (Ghz)	1.86	1.33	1.6	1.6	1.6
GPU freq (Ghz)	1.30	0.85	1.12	1.12	1.12
Power budget	N/A	7.5W	15W	15W	15W

TABLE 5.2: NVIDIA Jetson TX2 clock configuration with power modes(Link).

5.1.2 Desktop PC

We also tested the same vSLAM algorithm in a personal desktop computer equipped with Intel Core i7 6700 CPU, NVIDIA GTX 1060 GPU and 16GB DDR4 RAM. It's important to mention that the Jetson TX2 and the GTX 1060 have the same Pascal GPU architecture, making them slightly more fair compare in contrast to newer more sophisticated architectures. The specifications of the system are presented in table 5.3:

CPU	
Cores/Threads	4/8
Frequency base/boost	3.4 GHz/ 4.0 GHz
Cashe	8 MB
Power	65 W
GPU	
CUDA cores	1280
Frequency base/boost	1.5 GHz/ 1.7 Ghz
Memory Size	6 GB GDDR5
Memory Bandwidth	192 GB/sec
Memory Interface	192 bit
Power	120 W

TABLE 5.3: Desktop PC specifications

5.2 Datasets and Performance Metrics

We tested two types of datasets: EuRoC MAV dataset [38] and our own full high definition drone video. The first one provides videos taken from a micro aerial vehicle(MAV) with a stereo camera. Moreover, it provides extra sensors like an internal measurement unit(IMU), a motion capture system, and a laser tracker. The combination of all these sensors results in a very accurate movement and position of the drone where we will try to replicate with our algorithm. The EuRoC dataset is divided into three levels of difficulty: easy, medium, and difficult. The difficulty is based on the manoeuvres the drone does and the difference in light exposure during the video. Our goal with this dataset is to confirm that our vSLAM is accurate and reliable. We expect to have some performance improvement but our focus is on design verification.

For our dataset, we also used a MAV available in our laboratory. The difference is that we used a monocular camera set to 1920x1080 pixels video at 30 frames per second. Our footage replicates the EuRoC's levels of difficulty: from flying in a straight line to making fast-turning manoeuvres. With this dataset, we only aim to measure the performance improvements comparing the CPU-only and the GPU accelerated version of our system.

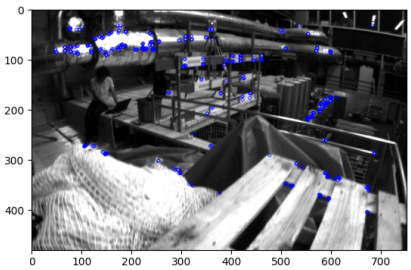
5.2.1 EuRoc Dataset

EuRoC dataset [38] uses stereo cameras during the drone recording with resolution of 752x480 pixels. The dataset format include:

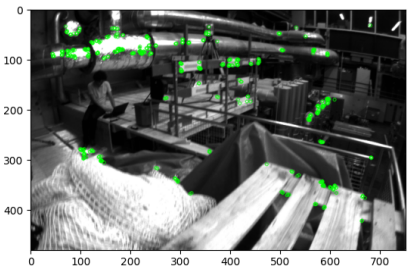
- Data collected from every sensor accompany with timestamps.

- Calibration parameters for each sensor.
- Ground truth estimation for every frame.

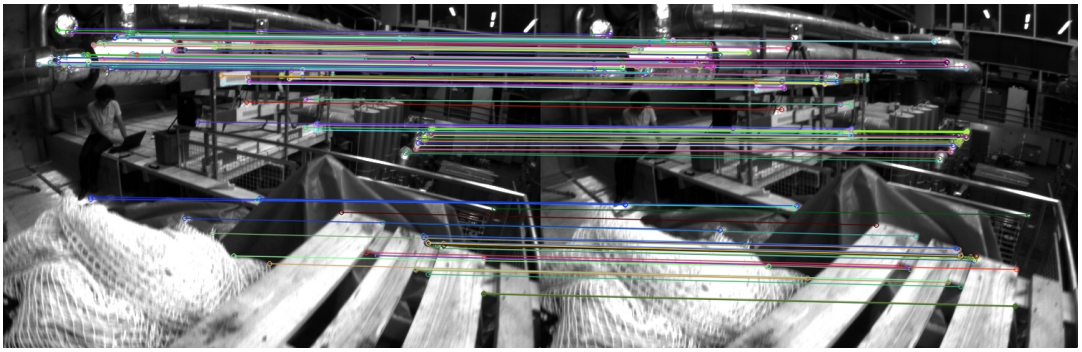
Out first test is to find features in two random consecutive images from the dataset and feature match the results. The test involves our system before and after the GPU-acceleration.



(A) Feature detection in image 1



(B) Feature detection in image 2



(C) Feature matching between image 1 and 2

FIGURE 5.1: GPU feature extraction and feature matching.

For our next experiment, we calculated the average feature matches on the different difficulties provided from EuRoc dataset. We compared the CPU and GPU average results after 10 runs:

	CPU only	**GPU accelerated**
MH01_easy	349	326
MH03_medium	299	256
MH04_difficult	278	219

Here we noticed that the GPU version has fewer matches. We believe that this happens because we perform feature detection and non-maximum suppression in each layer of Gaussian pyramid. Non-maximum suppression removes more matches in total but the difference does not reduce the efficiency notably.

For our following test, we compared the ground truth data with our camera motion trajectory estimation. We used the python package evo [39] to visualize the trajectories of our system. The comparison is between with and without GPU acceleration shown in 5.2.

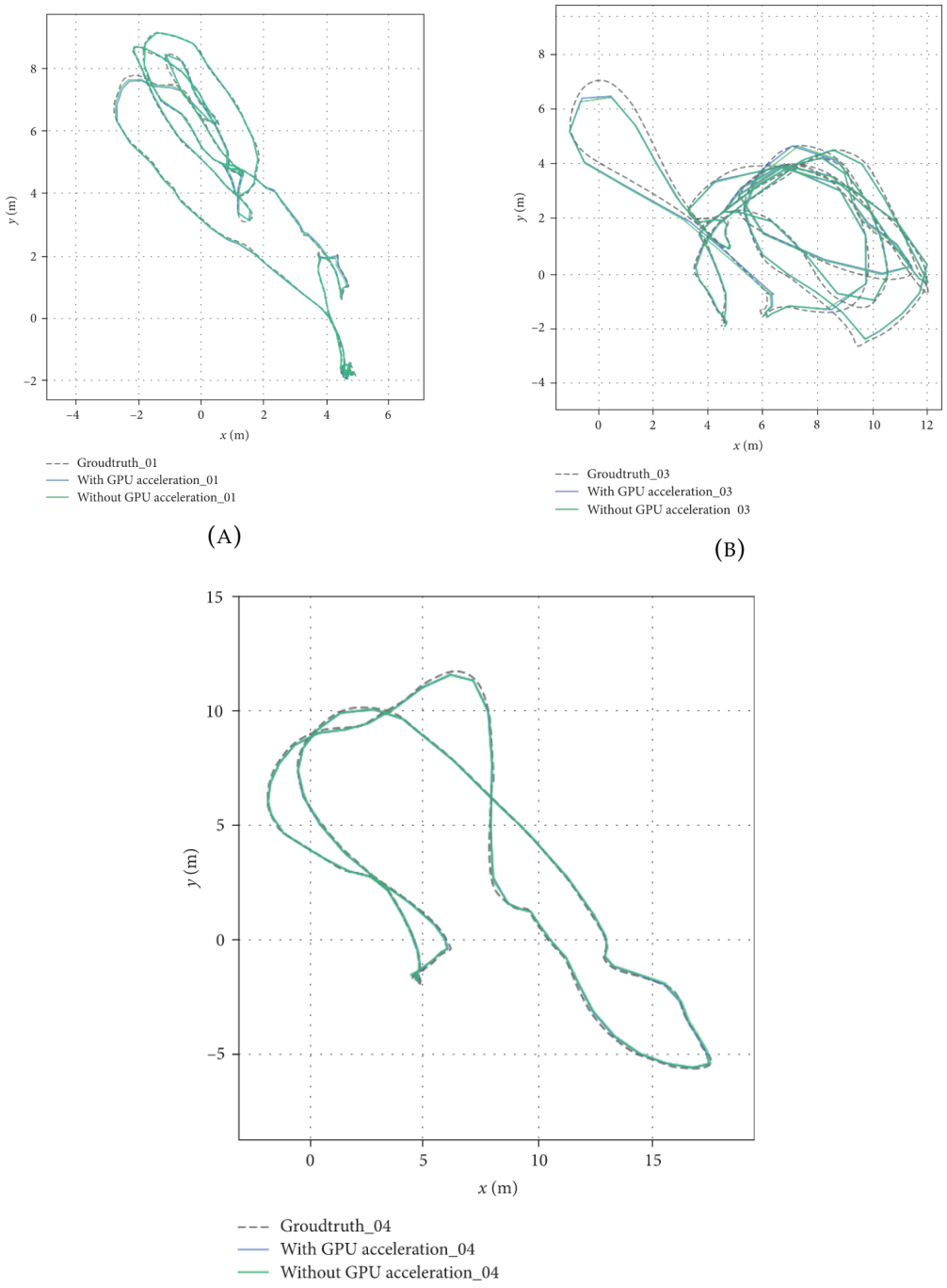


FIGURE 5.2: Trajectory comparisons between ground truth, CPU and GPU versions using evo package[39]

From the above figures, we can conclude that with GPU acceleration we do not have any negative effect on the accuracy. We can also calculate the absolute error between the ground truth provided and our GPU system. We can further examine the difference by calculating the root-mean-square error

(RMSE):

$$RMSE((\theta)) = \sqrt{MSE(\theta)} \tag{5.1}$$

Where mean square error (MSE) is calculated from the absolute error. Moreover we measure the square root of deviation, Standard Deviation (STD) between the actual value and the observed. The measurments are shown in the table 5.4:

	Mean(m)	**Median(m)**	**RNSE(m)**	**STD(m)**
MH01	0.255	0.251	0.280	0.138
MH03	0.450	0.405	0.501	0.261
MH04	0.445	0.441	0.515	0.255

TABLE 5.4: Absolute error between estimated and true trajectories

Next, we tested the performance for every EuRoc dataset. We used the PC platform for both CPU and GPU versions, and the Jetson Tx2 CPU and GPU version. According to EuRoc [38], the images have 752x480 resolution the datasets characteristics are:

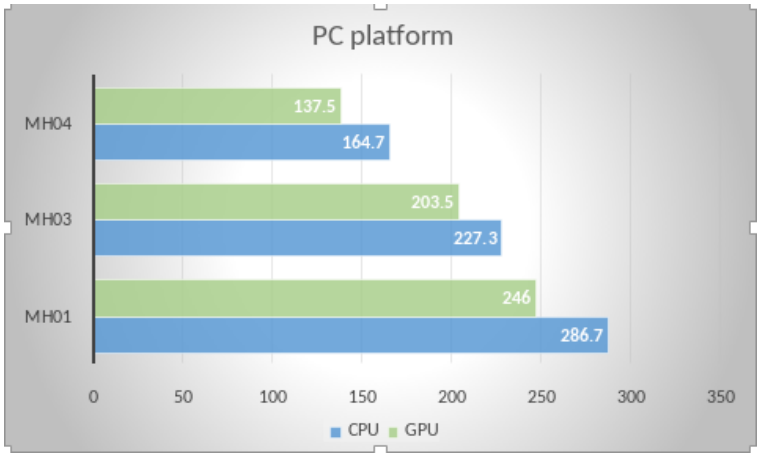
	Video Time	**Total Frames**
MH01_easy	182s	1820
MH03_medium	132s	1320
MH04_difficult	99s	990

The performance results are shown in 5.3:

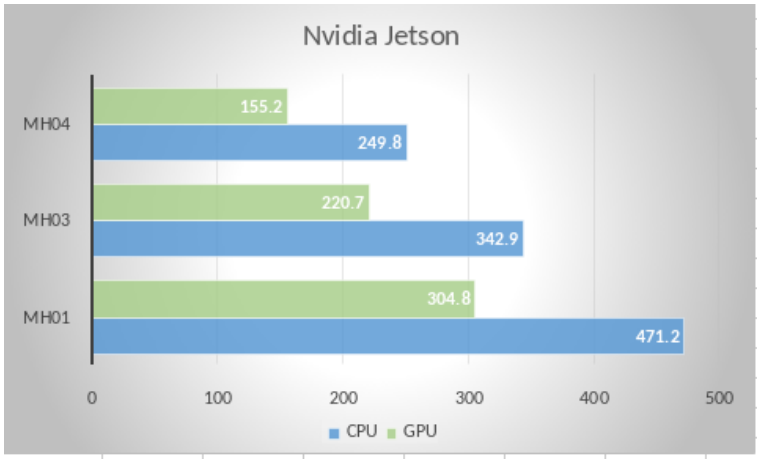
Note that for all the above tests we used 500 max features. We notice a performance boost with GPU acceleration even though it is not our primary resolution goal. The boost is even more apparent for the Jetson platform because it is designed for GPU workflows. The speed-up for every dataset are shown in 5.5

	PC speedup	**Jetson Tx2 speedup**
MH01	x1.164	x1.545
MH03	x1.197	x1.609
MH04	x1.116	x1.553

TABLE 5.5: Speed up for PC and Jetson platform between CPU and GPU versions



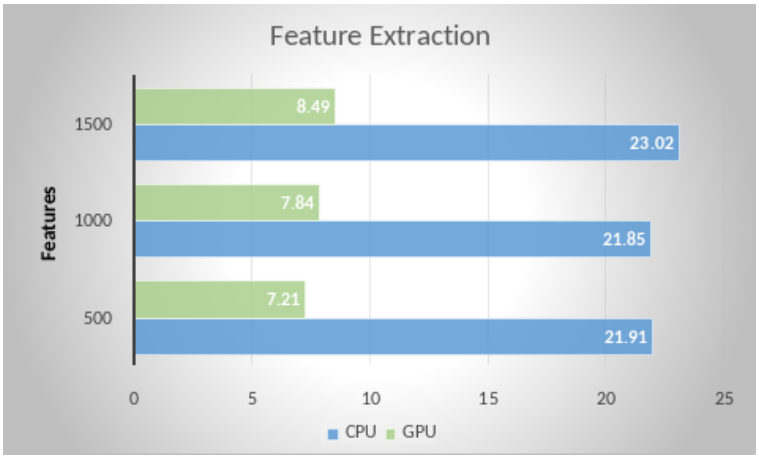
(A) PC execution times for CPU-only and GPU-accelerated versions.



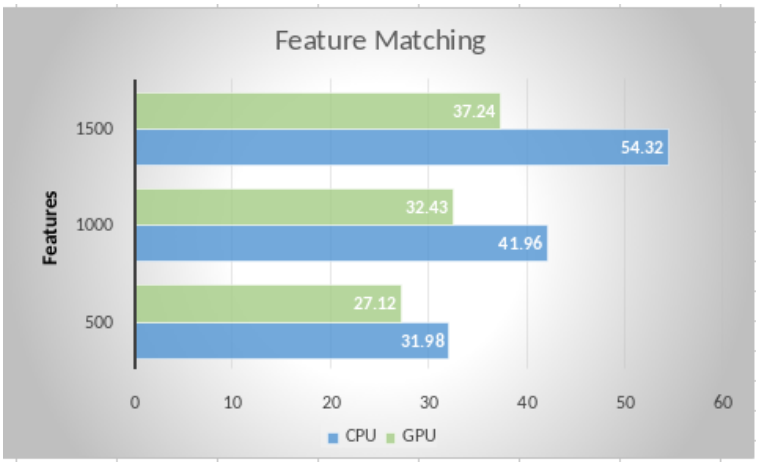
(B) Nvidia Jetson execution times for CPU-only and GPU-accelerated versions.

FIGURE 5.3: Execution time(seconds) of EuRoc datasets for A) PC platform, B) Nvidia Jetson.

Lastly, we compare the average feature extraction and matching times for 500, 1000, and 1500 features per image. According to 4.1, feature matching, localization and mapping are affected by the maximum number of features per image. For this test, we aim to measure the speed up for Brute-force step. The results are shown in 5.4



(A) Feature Extraction for 500, 1000 and 1500 total features.



(B) Feature Matching for 500, 1000 and 1500 total features.

FIGURE 5.4: Execution time(milliseconds) of A) Feature extraction, B) Feature Matching.

5.2.2 Our Dataset

For our dataset, we included three 1920x1080 pixel drone videos. We manouvred around the campus buildings. Every video has a difficulty level similar to EuRoc dataset. The difficulty is defined based on the drone movement speed and the path trajectory.

For camera calibration parameters we used 12 pictures of a 7x9 chessboard. Using the Zhang camera calibration algorithm we obtain the intrinsic matrix and the distortion parameters.

Distortion parameters we can be used to undistort and re-map our images. In our case, the results are:

- F_x: 1448

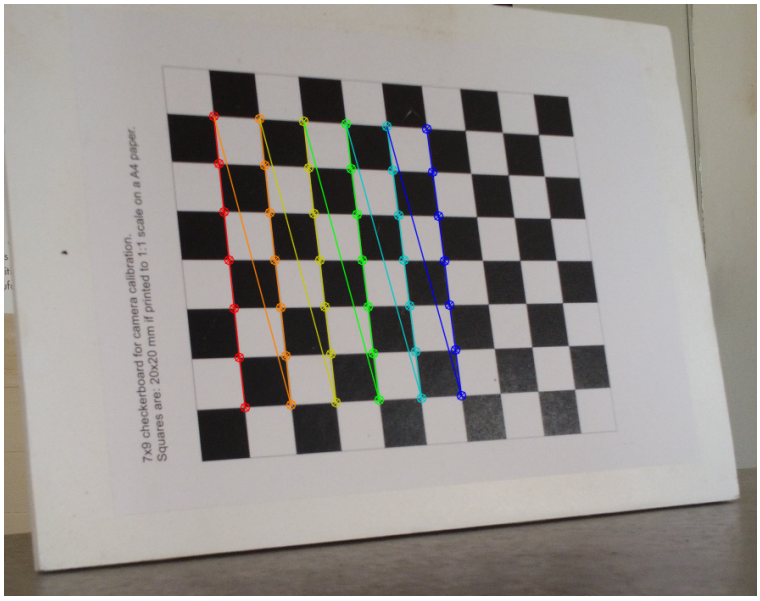
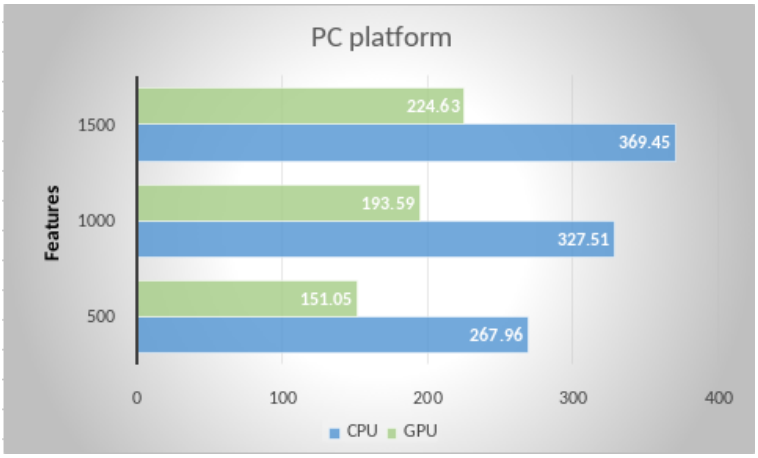


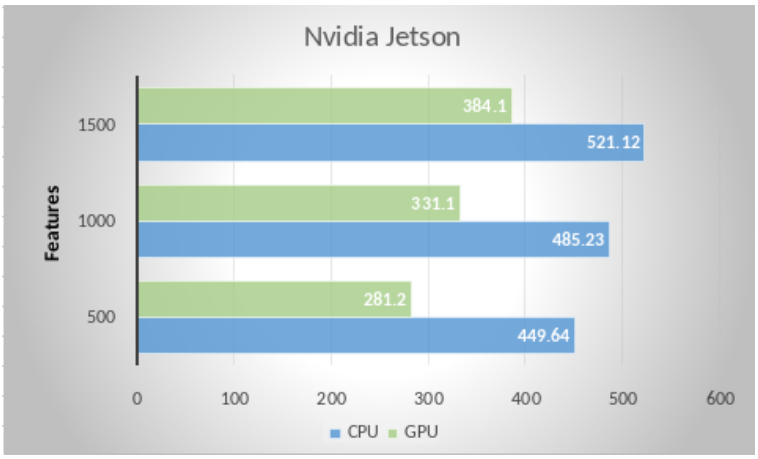
FIGURE 5.5: Zhang method for camera calibration

- F_y: 1446
- C_x: 969
- C_y: 539
- *dist*: [k1, k2, p1, p2, k3]=[0.0158, 0.038, -0.005, 0.002, -0.289]

With the calibrated camera we move on to our final test. We use a 1920x1080 pixel video captured by our own lab drone. The video is 30 frames per second and MPEG-4 format, meaning we have to get rid of some extra data in the initialization step. We tested for 500, 1000, and 1500 max features. The results are shown in 5.6



(A) PC execution time(seconds) for 500, 1000 and 1500 total features.



(B) Nvidia Jetson execution time(seconds) for 500, 1000 and 1500 total features.

FIGURE 5.6: Execution time of our dataset for A) PC platform, B) Nvidia Jetson.

The speed-up for every test is shown in table 5.6:

	PC speedup	Jetson Tx2 speedup
500 features	x1.773	x1.599
1000 features	x1.692	x1.465
1500 features	x1.644	x1.356

TABLE 5.6: SLAM speed-up for PC and Jetson platform, between CPU and GPU versions.

As the results show, we have almost similar results with lower resolutions. Even though GPU accelerates feature extraction and feature matching, we rest of SLAM operates under CPU processing. Moreover, according to 4.1 $T_{overhead}$ increase exponentially with the image resolution, moving back-and-forth from CPU to GPU memory. Jetson Tx2 due to the nature of its platform has low powered CPU with limited memory speeds compared to a desktop PC.

The results are more promising if we compare only feature extraction and matching times. For 1500 max feature points the results are shown in 5.7:

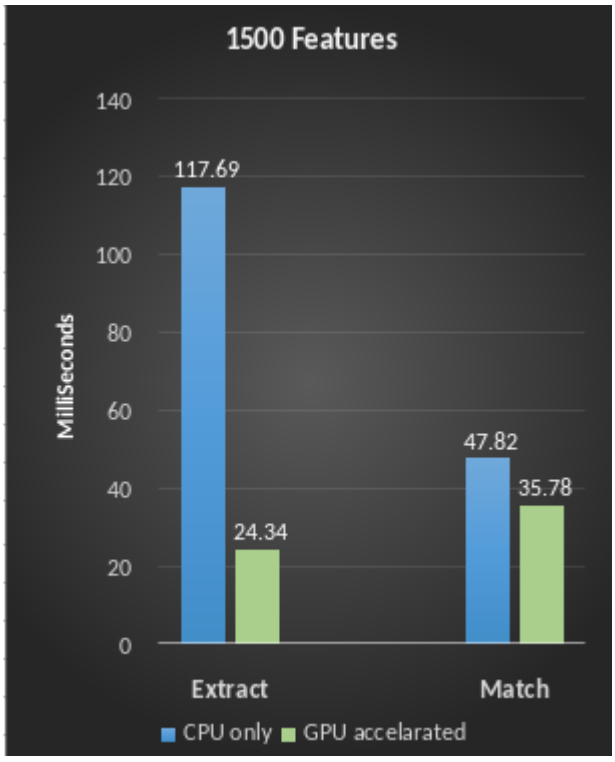


FIGURE 5.7: Execution time of feature extraction and matching for 1920x1080 pixel video with 1500 max feature point

The GPU accelerated version of our system feature extraction creates a speed-up of up to 4.8 times to the CPU-only counterpart. We use the FLANN-based

feature matching, shown in 3.2.2, because it is more efficient for more than 500 features. For the GPU matching version, we achieve x1.3 speed-up.

5.3 Summary of Results

In the previous sections, we verified that our system works with high precision using EuRoc dataset. Later on, we tested our dataset with high-resolution videos which is the goal of this thesis. Our system performed great on the SLAM parts where we used GPU to accelerated. As shown in 5.7, we achieved x4.8 speed-up for feature extraction and x1.3 speed-up on feature matching. However, our system did not achieve great results to the whole SLAM algorithm, picking at x1.6 speed up at best.

Nevertheless, it is unfair to compare a desktop PC with an embedded system only in terms of performance. Therefore, for our last benchmark, we will compare the energy consumption of each platform using CPU-only and GPU-accelerated versions.

As a benchmark, we used our own 1920x1080 pixel video with 500 max features where the results are shown in 5.6. Some metrics include:

1. **Static power:** represents the average amount of power consumed when no active computation is taking place(idle power). In both PC and Jetson platforms we have a monitor connected drawing some insignificant idle power.
2. **Dynamic power:** represents the average amount of power consumed during SLAM operation.
3. **Performance:** measured with the frames per second. As a speed-up reference we use the PC CPU-only platform.
4. **Energy per frame:** represents the amount of energy consumed in Joules per frame:

$$Energy/Frame = \frac{PowerConsumed(Watt)}{Framespersecond} \tag{5.2}$$

5. **Total Efficiency:** a combination of performance and energy efficiency:

$$TotalEfficiency = Performance \times EnergyEfficiency \tag{5.3}$$

The results are shown in the table 5.7:

	PC CPU only	PC GPU accelarated	Jetson CPU only	Jetson GPU accelarated
Name	i7 6700	Gtx 1060	Arm A57	256 core Pascal
Total On-Chip power (Watt)	65	120	∼6	∼15(6 + 9)
Idle Power (Watt)	5	8	1.2	0.5
SLAM Power Cons. (Watt)	38.1	44.6 + 36	6	6 + 9
Frames per Second (fps)	4.70	8.38	2.81	4.50
Performance	x1	x1.78	x0.60	x0.95
Energy per Frame (Joule/frame)	8.10	9.64	2.14	3.33
Energy Efficiency	x1	x0.84	x3.70	x2.43
Total Efficiency (Perf. × Energy)	**x1**	**x1.49**	**x2.20**	**x2.30**

TABLE 5.7: Summary of various results.

As expected, the PC platform is 1.5-3x faster than the Jetson counterpart but consumes 2.5-3.5x more power. As a result, our system is fast enough to be eligible but also consumes small amounts of power making it a perfect fit for an unmanned vehicle.

We want to mention here that in the PC platform we only used the 55% to 60% of CPU resources and about 25% to 30% of GPU. This means that the PC platform is capable to run many more multiple processes while we used almost the 80-90% of Jetson resources.

Chapter 6

Conclusions and Future Work

6.1 Conclusions and Future Work

Visual SLAM is the process to create a map of the environment and estimate the robot's position on the map. The only data input is from the camera sensor in form of images. In order to have optimum results, we process a high resolution of image data where the current embedded platforms struggle to accomplice. In this paper, we propose a GPU accelerated version of ORB features SLAM, aimed towards high-resolution images. Our proposed method improved the execution speed and efficiency while maintaining the accuracy need for vSLAM.

Navigating in an unknown environment creates a plethora of problems where vSLAM solves only a part of them. There will always be unexpected problems that the robot must encounter, but by increasing the efficiency of SLAM we have more time and resources available. Here are some thoughts for future autonomous mobile robots:

- Modern SLAMs has many other features that we did not include in our project. One popular feature is the loop close where we can detect if the robot is in the same position as before. Our system only stores a set amount of features, meaning after a short time we loose info of early keypoints. An additional element is an adequate pathfinder within the created map of SLAM. Lastly, there are some proposed solutions to the kidnapped problem where the robot in operation is carried to an arbitrary location and must localize itself again.
- In this thesis we proposed a solution for GPU emended system that has already exited 4 years now. By checking the NVIDIA Jetson hardware roadmap, the Jetson Orin family will be introduced in early 2022.

With newer GPU architecture NVIDIA might add hardware encoding and decoding support in GStreamer. This will hugely increase Jetson's performance and it will be more competitive to hardware-accelerated boards.

- There are more platforms to be explored like FPGA or ASIC. Both of them have more complex design flow but usually outperform convectional CPUs or GPUs in processing speed, power consumption, and cost. We think this could be interesting since there are researches [40] proving that an ASIC performs better than a GPU or a CPU in their specific case.

References

[1] R. Chatila and J. Laumond. "Position referencing and consistent world modeling for mobile robots". In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. 1985, pp. 138–145. DOI: 10.1109/ROBOT.1985.1087373.

[2] Koray Celik et al. "Monocular vision SLAM for indoor aerial vehicles". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2009, pp. 1566–1573.

[3] Seo-Yeon Hwang and Jae-Bok Song. "Monocular vision-based SLAM in indoor environment using corner, lamp, and door features from upward-looking camera". In: *IEEE Transactions on Industrial Electronics* 58.10 (2011), pp. 4804–4812.

[4] Paul Newman, David Cole, and Kin Ho. "Outdoor SLAM using visual appearance and laser ranging". In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE. 2006, pp. 1180–1187.

[5] Robert Zlot and Michael Bosse. "Efficient large-scale three-dimensional mobile mapping for underground mines". In: *Journal of Field Robotics* 31.5 (2014), pp. 758–779.

[6] Shoudong Huang and Gamini Dissanayake. "Convergence and consistency analysis for extended Kalman filter based SLAM". In: *IEEE Transactions on robotics* 23.5 (2007), pp. 1036–1049.

[7] Xiaotong Xie et al. "An EKF SLAM algorithm for mobile robot with sensor bias estimation". In: *2017 32nd Youth Academic Annual Conference of Chinese Association of Automation (YAC)*. IEEE. 2017, pp. 281–285.

[8] Arturo Gil et al. "Multi-robot visual SLAM using a Rao-Blackwellized particle filter". In: *Robotics and Autonomous Systems* 58.1 (2010), pp. 68–80.

[9] Michael Montemerlo et al. "FastSLAM: A factored solution to the simultaneous localization and mapping problem". In: *Aaai/iaai* 593598 (2002).

[10] Giorgio Grisetti et al. “A tutorial on graph-based SLAM”. In: *IEEE Intelligent Transportation Systems Magazine* 2.4 (2010), pp. 31–43.

[11] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. “Improved techniques for grid mapping with rao-blackwellized particle filters”. In: *IEEE transactions on Robotics* 23.1 (2007), pp. 34–46.

[12] Wolfgang Hess et al. “Real-time loop closure in 2D LIDAR SLAM”. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2016, pp. 1271–1278.

[13] Stefan Kohlbrecher et al. “A flexible and scalable SLAM system with full 3D motion estimation”. In: *2011 IEEE international symposium on safety, security, and rescue robotics*. IEEE. 2011, pp. 155–160.

[14] David Ribas et al. “SLAM using an imaging sonar for partially structured underwater environments”. In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2006, pp. 5040–5045.

[15] Maurice F Fallon et al. “Relocating underwater features autonomously using sonar-based SLAM”. In: *IEEE Journal of Oceanic Engineering* 38.3 (2013), pp. 500–513.

[16] Albert Diosi, Geoffrey Taylor, and Lindsay Kleeman. “Interactive SLAM using laser and advanced sonar”. In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. IEEE. 2005, pp. 1103–1108.

[17] Jakob Engel, Thomas Schöps, and Daniel Cremers. “LSD-SLAM: Large-scale direct monocular SLAM”. In: *European conference on computer vision*. Springer. 2014, pp. 834–849.

[18] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. “OpenVSLAM: A versatile visual SLAM framework”. In: *Proceedings of the 27th ACM International Conference on Multimedia*. 2019, pp. 2292–2295.

[19] Edward Rosten and Tom Drummond. “Machine learning for high-speed corner detection”. In: *European conference on computer vision*. Springer. 2006, pp. 430–443.

[20] Ethan Rublee et al. “ORB: An efficient alternative to SIFT or SURF”. In: *2011 International conference on computer vision*. Ieee. 2011, pp. 2564–2571.

[21] Georg Klein and David Murray. “Parallel tracking and mapping for small AR workspaces”. In: *2007 6th IEEE and ACM international symposium on mixed and augmented reality*. IEEE. 2007, pp. 225–234.

[22] Janosch Nikolic et al. “A synchronized visual-inertial sensor system with FPGA pre-processing for accurate real-time SLAM”. In: *2014 IEEE*

international conference on robotics and automation (ICRA). IEEE. 2014, pp. 431–437.

[23] Weikang Fang et al. "FPGA-based ORB feature extraction for real-time visual SLAM". In: *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE. 2017, pp. 275–278.

[24] Frei and Chung-Ching Chen. "Fast Boundary Detection: A Generalization and a New Algorithm". In: *IEEE Transactions on Computers* C-26.10 (1977), pp. 988–998. DOI: 10.1109/TC.1977.1674733.

[25] Djemel Ziou, Salvatore Tabbone, et al. "Edge detection techniques-an overview". In: *Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii* 8 (1998), pp. 537–559.

[26] John Canny. "A computational approach to edge detection". In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), pp. 679–698.

[27] Weibin Rong et al. "An improved CANNY edge detection algorithm". In: *2014 IEEE international conference on mechatronics and automation*. IEEE. 2014, pp. 577–582.

[28] Yuancheng Luo and Ramani Duraiswami. "Canny edge detection on NVIDIA CUDA". In: *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE. 2008, pp. 1–8.

[29] Kohei Ogawa, Yasuaki Ito, and Koji Nakano. "Efficient Canny edge detection using a GPU". In: *2010 First International Conference on Networking and Computing*. IEEE. 2010, pp. 279–280.

[30] Estevão S Gedraite and Murielle Hadad. "Investigation on the effect of a Gaussian Blur in image filtering and segmentation". In: *Proceedings ELMAR-2011*. IEEE. 2011, pp. 393–396.

[31] Christopher G Harris, Mike Stephens, et al. "A combined corner and edge detector." In: *Alvey vision conference*. Vol. 15. 50. Citeseer. 1988, pp. 10–5244.

[32] Hans P Moravec. *Obstacle avoidance and navigation in the real world by a seeing robot rover.* Tech. rep. Stanford Univ CA Dept of Computer Science, 1980.

[33] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[34] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features". In: *European conference on computer vision*. Springer. 2006, pp. 404–417.

[35] Zhengyou Zhang. "A flexible new technique for camera calibration". In: *IEEE Transactions on pattern analysis and machine intelligence* 22.11 (2000), pp. 1330–1334.

[36] Changchang Wu et al. "Multicore bundle adjustment". In: *CVPR 2011*. IEEE. 2011, pp. 3057–3064.

[37] Diego Rodriguez-Losada et al. "Gpu-mapping: Robotic map building with graphical multiprocessors". In: *IEEE Robotics & Automation Magazine* 20.2 (2013), pp. 40–51.

[38] Michael Burri et al. "The EuRoC micro aerial vehicle datasets". In: *The International Journal of Robotics Research* 35.10 (2016), pp. 1157–1163.

[39] Michael Grupp. *evo: Python package for the evaluation of odometry and SLAM.* https://github.com/MichaelGrupp/evo. 2017.

[40] Eriko Nurvitadhi et al. "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC". In: *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2016, pp. 77–84.