Implementing monocular-inertial SLAM on the Nvidia Jetson TX2 embedded platform

J Matthee

orcid.org/ 0000-0002-2139-0648

Dissertation accepted in fulfilment of the requirements for the degree *Master of Engineering in Computer and Electronic Engineering* at the North-West University

Supervisor:	Prof KR Uren
Co-supervisor:	Prof G van Schoor
Co-supervisor:	Dr CE van Daalen

Graduation: November 2023

[10] *I rejoiced greatly in the Lord that at last you renewed your concern for me. Indeed, you were concerned, but you had no opportunity to show it.* [11] *I am not saying this because I am in need, for I have learned to be content whatever the circumstances.* [12] *I know what it is to be in need, and I know what it is to have plenty. I have learned the secret of being content in any and every situation, whether well fed or hungry, whether living in plenty or in want.* [13] *I can do all this through him who gives me strength.* Philippians 4:10–13

Acknowledgments

I would like to express my special thanks of gratitude to the following persons who gave me the opportunity to complete my Masters Degree:

- To my study leaders Prof. Kenny Uren, Prof. George van Schoor, and Dr. Corné van Daalen for their guidance and support throughout my studies.
- To my mother and father for their love and support. Thank you for the way you have raised me, for all the things you have taught me, for supporting me financially, and for providing me with all the opportunities.
- To my fiancé who was the backboard for my ideas and helped me cross a few hurdles during my studies. Thank you for being there for me, motivating me to finish my degree, and helping me to deliver quality work. I look forward to sharing many more challenges and memories with you. I will always admire and appreciate your love and support.
- To the members of the McTronX research group for their feedback.
- To the unmentioned friends and family that always supported me over the years and was always ready if I needed to blow off some steam.
- To SAAB Grintek Defense, giving me the opportunity and necessary financial support over the years of completing my degree.

Abstract

SLAM is a complex computational problem where a robot or vehicle concurrently creates a representation of its environment while determining its position within the environment. State-of-the-art SLAM algorithms can achieve high accuracy at high speeds. However, these algorithms are usually developed, tested, and optimised on high-end desktop computers, which are not used in the design of the robots or vehicles that implement SLAM. These robots or vehicles use embedded platforms with limited processing power, which can cause severe degradation in the execution speed of the SLAM algorithm. Thus, since it is practically more feasible, SLAM algorithms need to be tested on embedded platforms.

The study aims to create a prediction model that can estimate the performance that a SLAM algorithm can achieve on an embedded platform. ORB-SLAM3 is a state-of-the-art visual SLAM algorithm that supports multiple camera settings and configurations. ORB-SLAM3 will be implemented on the Nvidia Jetson TX2, Raspberry Pi 3b+, and Raspberry Pi 4B embedded platform, where the performance will be measured. The EuRoC MAV dataset is used as it provides camera and inertial sensor data with accurate ground-truth measurements. ORB-SLAM3, built with the default version of OpenCV 3.2.0, was profiled, and it was identified that OpenCVs' FAST function is the bottleneck when executing on the Nvidia Jetson TX2. Investigating the OpenCV3.2.0 source code showed that the FAST function does not use SIMD instructions on ARM architectures, meaning the hardware resources of the Nvidia Jetson TX2 are not fully utilised.

ORB-SLAM3 was containerised and executed on the three embedded platforms, and the average tracking time was measured. PassMark is used to benchmark the embedded platforms to characterise the CPUs of the platforms. Since three embedded platforms are a small sample size, artificial CPUs are generated by varying the CPU frequencies and core counts. The results from the benchmarking and the execution of ORB-SLAM3 were used to create a dataset to train and test prediction models. The prediction model's target is the average tracking time that ORB-SLAM3 can achieve on embedded platforms. The inputs to the model were selected using cross-correlation and guided by the profiling results of ORB-SLAM3. The inputs used to create the prediction model are the CPU frequency, PassMark Single CPU score, PassMark NEON CPU score and PassMark Integer score.

Three modelling techniques are used to create prediction models: A simple linear regression, an Extra Trees Regressor, and a Multi-Layer Perceptron model. Two experiments are investigated for verification and validation. The first experiment, which serves as verification, is to create the models using the entire dataset of 429 unique entries, with a 75:25 ratio for training and testing. The second experiment that serves as validation is when a CPU is removed from the dataset to see how the prediction models react on an unseen CPU. The performance criteria that the models should achieve are an MAE and RMSE % of less than 10 %, and a R^2 of more than 0.9.

With the verification and validation experiments, the Extra Trees modelling technique provided the best performance. For verification, the Extra Trees model achieved an MAE of 2.84 %, RMSE of 3.93 % and an R^2 score of 0.95. The verification experiment met all the performance criteria, whereas the validation experiment could only meet the MAE % criteria. The validation experiment values were 9.12 %, 13.14 % and 0.78 for MAE %, RMSE % and R^2 score respectively. Although the validation experiment did not meet the criteria, the results were very close and could have been reached if the model had been tuned or more training data had been available.

The study showed that embedded platforms could successfully implement SLAM algorithms, although the algorithms might need to be optimised or modified to achieve real-time performance. A prediction model can be used to see what performance ORB-SLAM3 can achieve on an embedded platform.

Keywords: Monocular-Inertial, SLAM, embedded platform, ORB-SLAM3, Nvidia Jetson TX2

Contents

List of Figures

List of Equations

List of Tables

Abbreviations

ALU	Arithmetic logic unit
AMAT	Average memory access time
ATE	Absolute Trajectory Error
CISC	Complex Instruction Set Computer
CPI	Cycles per instruction
CPU	Central Processing Unit
DCO	Dynamic Code Optimization
DRAM	Dynamic Random Access Memory
FPS	Frames Per Second
FSM	Finite state machines
GPU	Graphics Processing Unit
HIL	Hardware-in-the-loop
I/O	Input or Output
ILP	Instruction-level parallelism
IMU	Inertial Measurement Unit
IPC	Instructions per cycle
ISA	Instruction Set Architecture
LRU	Least recently used
MAE	Mean Absolute Error
MCU	Microcontroller
MLP	Multi-Layer-Perceptron
MPU	Microprocessing unit
PMU	Performance Monitoring Unit
RAW	Read After Write
RISC	Reduced Instruction Set Computer
RMSE	Root Mean Square Error
ROS	Robot Operating System
SBC	Single-board computer
SIMD	Single instruction multiple data
SLAM	Simultaneous Localization And Mapping
SMT	Simultaneous multi-threading
SoC	System-On-Chip
SPE	Software Performance Engineering
SRAM	Static Random Access Memory
SWaP	Size Weight and Power
TDP	Thermal Design Power
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicles
VISLAM	Visual Inertial Simultaneous Localization And Mapping
VO	Visual Odometry
VSLAM	Visual Simultaneous Localization And Mapping

Symbols

t_{VM}	Virtual memory access time
MR_{MM}	Main memory miss rate
t_{MM}	Main memory access time
MR_{cache}	Cache miss rate
t_{cache}	Cache access time
N	Cache degree of associativity
B	Number of blocks in cache
b	Cache block size
S	Number of cache sets
C	Cache Capacity

Chapter 1 - Introduction

1.1 Background

Simultaneous Localization and Mapping (SLAM) is the process where a robot or vehicle concurrently creates a representation of its environment while determining its position within the environment. SLAM is usually developed on high-end machines and is primarily used to develop autonomous vehicles or robots. However, due to the size and power constraints of certain robots or vehicles, it would be more practical to implement SLAM on embedded platforms.

Unlike humans, robots and vehicles cannot think for themselves; thus, they are programmed to act according to their perception of the world around them. A robot gains perception of the environment around it by using different sensors. The robot's perception creates a virtual environment of its surroundings while simultaneously determining where it is within the environment. This is commonly known in the research community as simultaneous localization and mapping (SLAM). The challenges presented by SLAM are:

- To create an accurate map of the environment, the robot needs to have accurate localization.
- To localize a robot accurately in an environment, an accurate representation of the environment is needed.

Therefore, SLAM is referred to as a chicken-and-egg problem [1]. SLAM implementations are constructed using four major algorithmic components: landmark extraction, data association, state estimation, and state and landmark update. Each of the mentioned algorithmic components has multiple possible solutions, depending on the environment, sensors used, and the type of robotic platform it is implemented on. Since multiple solutions exist, the SLAM problem becomes a diverse topic. The integration and implementation of the parts can make SLAM very computationally expensive. This is said as there are algorithms (ORB [2], SIFT [3], SURF [4]) that extract features from the environment. The extracted features are used within other algorithms, such as pose graph optimization or bundle adjustment, to estimate the pose (position and orientation) and generate a map of the environment. Once all the data have been processed, the robot can move to a new location and the process restarts.

For this reason, different algorithms have been developed to be less computationally expensive. Practically, SLAM needs to be implemented on mobile platforms, which provides some restrictions such as available computational power, overall weight, payload, and physical form. Unmanned ground vehicles (UGVs) have the benefit of being more modular and adaptive to changes to the system, such as added computational power, additive sensors, or power

capacity. With unmanned aerial vehicles (UAVs), there exists a high compromise between the performance, flight time, maneuverability, and payload.

In the past, the success of SLAM was achieved by implementing it in known environments or by placing identifiable landmarks in the environment [5]. Although these methods proved to be successful, it is not entirely practical. Real-world examples that provide more practical scenarios include: search and rescue operations, military reconnaissance and attack, and automatic exploration of areas. With these examples, there is very seldom any prior knowledge of the environment, which in turn makes the proposed methods in [5]–[7] impractical. The same could be said for methods that use GPS-aided SLAM methods, as there is the possibility of electronic warfare, indoor environments, or the interference of natural elements blocking the GPS signals.

The basic building blocks needed to implement SLAM in a real-world scenario are sensors, actuators, processors, and algorithms. The interconnection between the basic building blocks is illustrated in Figure 1. The processor can be any processor powerful enough to run a SLAM algorithm, whether it is an embedded computer or a high-end desktop computer. The processor and algorithm control the flow of information between the sensors and the actuators. Different sensors can be used to implement SLAM, including sonar, LiDAR, and various types of camera configurations. Any robotic platform can be used, for example a differential robot or a quadcopter. The robotic platform that is used can influence the choice of sensors and computer systems. The most significant contributor to deciding what system and sensors to use is the cost and how computationally expensive the calculations will be. Sensors, such as LiDAR, provide accurate results, but an inexpensive camera can achieve similar results depending on the environment. Another factor that influences the feasibility of real-world applications is whether the algorithm can run in real-time. Thus, the robotic platform, from sensors to processors, must be chosen carefully to ensure real-time processing.

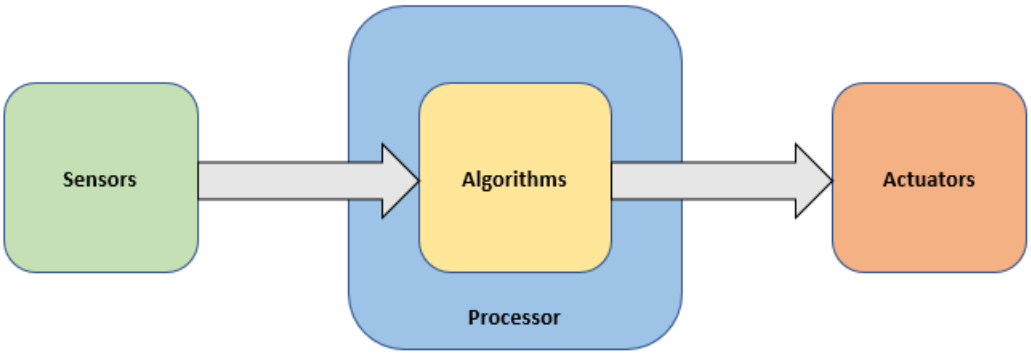


Figure 1: **Block diagram showing the interconnection between the basic building blocks needed to implement SLAM**

Visual SLAM (VSLAM) has gained popularity since it can provide a very detailed

representation of the environment in a 3D space, whereas other sensors such as LiDAR provide 2D representation. The most significant disadvantage of VSLAM is that it is more computationally expensive since images contain more data than a laser scan. The advantage of VSLAM is the ability to construct a dense representation of the environment that can be used for autonomous navigation and mapping applications. VSLAM can be divided into three categories: monocular, stereo, and RGB-D.

Monocular SLAM consists of a single camera, whereas stereo has two or more cameras. The most significant disadvantage of using monocular SLAM is the scale ambiguity problem, as the scale cannot be determined from a single camera. Otherwise, monocular SLAM is easy to implement, is the least computationally expensive, and is very low cost. Stereo VSLAM consists of two synchronized monocular cameras mounted at a known distance called the baseline. Due to the use of two cameras and the known baseline, the scale ambiguity problem is solved. Since the depth range measured by a stereo camera is related to the baseline length, autonomous vehicles using stereo cameras can be quite large in size. Stereo cameras have the disadvantage of being hard to configure and calibrate. The baseline length and camera resolution limit the depth range and accuracy, and the computational resources are high. Thus, stereo SLAM usually requires a GPU or FPGA to run in real-time. RGB-D cameras, "depth cameras", use infrared to measure the distance between objects and the camera, thus decreasing the computational complexity. RGB-D cameras suffer from a narrow measurement range, noisy data, and a small field of view. They are also susceptible to sunlight interference and are unable to measure transparent material. The disadvantages of RGB-D cameras motivate why they are mainly used in indoor environments and why they are not suitable for outdoor applications [8]. VSLAM has also been expanded into visual-inertial SLAM or VISLAM, where visual and inertial data are used to get more stable estimation results [9], or in the case of monocular SLAM to remove the scale ambiguity problem.

Different datasets are available to the research community to test developed SLAM algorithms. Some of the datasets include Kitti [10], EuRoC MAV [11], ICL-NUIM [12], and TUM dataset [13]. These datasets provide raw data from different sensor types used in SLAM, such as RGB-D camera vision, stereo vision, monocular vision, and LiDAR measurements. Kitti also provides a benchmark tool that shows the performance of different SLAM algorithms and ranks them according to translational and rotational error and the algorithm's runtime. For visual SLAM, Kitti has 124 different SLAM algorithms and provides the performance measurements that are mentioned above. Currently, their highest ranking SLAM algorithm is a hybrid visual-lidar odometry and mapping (V-LOAM) algorithm. It has an impressive 0.54 % translational error and a 0.0013 degrees per meter rotational error. It is written in a C/C++ language and has a swift runtime of 0.1 s. Although it has an impressive performance, it uses 2 cores at 2.5 GHz processing speed. Some other algorithms, such as enhanced frame-2-frame stereo odometry (ESO), need more processing power of up to 3.0 GHz using 4 cores, which has a faster runtime of 0.08 s but has a larger error in translation and rotation. This shows that there is a balance between computational complexity and the accuracy of the

algorithm.

Abouzahir et al. [14] showed that there is a large performance difference when executing certain SLAM algorithms on an embedded platform compared to a high-end desktop computer. The amount of speedup seen by the desktop is 1.8, 6.4, and 2.2 for FastSLAM2.0 [15], ORBSLAM [16] and RatSLAM [17], respectively. This not only shows that the hardware constraints of an embedded platform influence the performance of SLAM algorithms, but it also shows that different SLAM algorithms experience different performance degradation.

In conclusion, the SLAM problem is a complex research topic. The push towards full autonomy of robots will not be easy as there are two conflicting realities: the increased autonomy requires significant computing capacity, and efficient designs will restrict the limits on cost, size, weight, and power [18]. For many embedded systems, there is a need for low power consumption and high performance [19]. An example of an embedded platform that is highly restricted to the limitations of size, weight, and power (SWaP) is unmanned aerial vehicles (UAVs). The most significant problem is that most SLAM algorithms are computationally expensive, and the systems capable of running them can be costly. Algorithms that are less computationally expensive exist, but some have a lower performance. For SLAM to be effective in critical missions, search and rescue operations, military reconnaissance and attack, it should be able to run in real-time. Careful consideration must be taken when designing a robot to implement SLAM as it should still be manoeuvrable, low-cost, and power-efficient. These factors influence the choices when deciding on the sensor types, the SLAM algorithm to be used, and the computer to control it all. Thus, there is a need to investigate the feasibility of using SLAM on an embedded platform.

1.2 Research question

The study will investigate whether an accurate model can be created to predict what performance ORB-SLAM3 can achieve on embedded platforms. The research will focus on creating a model that can predict the performance of ORB-SLAM3 on embedded devices that are constrained to SWaP due to their deployment in platforms such as UAVs.

1.3 Research objectives and methodology

1.3.1 Software and hardware implementation

The core objective behind the study is to implement a monocular-inertial SLAM algorithm on multiple embedded platforms, from which a model will be created to predict the performance of the monocular-inertial SLAM algorithm.

In order to create an accurate model for the prediction of ORB-SLAM3 performance, the

hardware first needs to be investigated to determine what parameters might influence the performance of the model. Thus, the different embedded platforms will be investigated and described. Since many factors can influence software performance, the monocular-inertial SLAM algorithm will be abstracted into a Docker container so that a single image can be downloaded and executed on all platforms without the need to install it on all the platforms. That will also ensure that exactly the same environment and settings will be used across all platforms.

1.3.2 Performance profiling

The second objective is to profile the monocular-inertial SLAM algorithm to help identify any possible bottlenecks that cause performance degradation on the embedded platform.

The profiling will be done using ARM Map, which will help identify the bottleneck by investigating the function call stack of ORB-SLAM3 during execution. The function call tree will aid in determining what function/s causes the bottleneck. This information will guide the source code examination process, which should identify poorly optimized code. The source code will also be dumped using the GDB debugger to get the exact instructions that are executed on the embedded platform.

1.3.3 Experimental design

An experimental design is required in order to create a prediction model that addresses the goal of the study.

Use the information gathered during the previous objectives to create an experimental design that will be suitable in creating a successful prediction model for the performance of ORB-SLAM3 on embedded devices.

1.3.4 Modelling

A model needs to be created from the information gathered during the study that can estimate the performance of ORB-SLAM3 on embedded platforms.

The model's inputs will be decided using the information gathered during the profiling of ORB-SLAM3. The ORB-SLAM3 container will be used to generate training and testing data used in the model.

1.4 Dissertation layout

Chapter 2 - Literature: This chapter investigates the differences between desktop computers and embedded platforms to help identify what characteristics cause their performance differences. Additionally, the chapter investigates the performance of SLAM implemented on embedded platforms and the challenges associated with implementation. This is followed by the investigation of the principles of modern CPUs and how design choices influence the performance thereof. The chapter also investigates the core functions and aspects of ORB-SLAM3 to understand the working principles.

Chapter 3 - Software and Hardware implementation: This chapter introduces the embedded platforms used in this study, looking at their differences. ORB-SLAM3 and the EuRoC MAV dataset are introduced, and the containerization process is explained.

Chapter 4 - Performance profiling: This chapter discusses performance measurements and profiling of ORB-SLAM3 on the Nvidia Jetson TX2. Afterwards, ORB-SLAM3 is profiled on the Nvidia Jetson TX2 to help identify the bottleneck. The source code of ORB-SLAM3 will also be investigated. The chapter concludes with dumping the execution of ORB-SLAM3 on the Nvidia Jetson TX2 to get a better insight into the actual execution of ORB-SLAM3 on an embedded platform. The profiling results will help guide what elements will be used as input to the model.

Chapter 5 Experimental design: The chapter will use the information gathered from the previous chapters to create an experimental design to be followed to create a prediction model.

Chapter 6 - Modelling: This chapter aims to create a model that will predict the performance of ORB-SLAM3 on embedded platforms. The data-gathering process will be explained, and different models will be created. The chapter will conclude with a discussion of the performance of the different models.

Chapter 7 - Conclusion and recommendations: This chapter summarises the work done throughout the study, concluding whether the study's objectives and aims are met and highlighting the challenges and recommendations for future studies.

Chapter 2 - Literature

2.1 Introduction

This study aims to create a model that can predict the performance that monocular-inertial SLAM can achieve on an embedded platform. Since SLAM is designed and tested on desktop computers, the common differences between desktop computers and embedded platforms need to be identified. Section 2.2 investigates the common differences between desktop computers and embedded platforms.

Section 2.3 investigates what SLAM methods have been implemented on embedded platforms. This section will help identify an embedded platform and a SLAM method used in the study. Section 2.4 highlights the challenges and shortcomings that other authors experienced when implementing SLAM on embedded platforms. Section 2.5 provides an explanation of Visual SLAM and an analysis of ORB-SLAM, which is the SLAM algorithm that will be used in the study.

The first portion of the Chapter mainly looked at the software aspects of the study, the remaining Sections look into the hardware aspect of the study. Since ORB-SLAM is written to execute on CPUs, a detailed look into how CPUs work will be given as the CPU architecture and memory-subsystem dictate the performance that ORB-SLAM can achieve. Section 2.6 describes CPU architecture and memory-subsystem, which defines the components and workings of a CPU. Section 2.7 will identify how CPU performance is measured and what CPU and software characteristics influence the performance of the CPU.

Chapter 2 is concluded with Section 2.8, which provides a critical review of the literature.

2.2 Desktop computers versus embedded platforms

Since SLAM is commonly developed and tested on high-end desktop computers, it is first necessary to identify the common differences between desktop computers and embedded platforms. Embedded platforms are usually designed for a single software-controlled task, meaning their sole purpose is to execute a specific function repeatedly. Single-board computers (SBC) and microcontrollers (MCU) are also classified as embedded platforms. Embedded platforms and desktop computers share many components, with the two notable differences being that the components in embedded platforms are lower-powered and smaller in size.

As with desktop computers, the most crucial part of an embedded platform is the CPU. Depending on the design of the embedded platform, the CPU can be a basic microprocessing unit (MPU) or a complex designed CPU. Regardless of the embedded platform or desktop computer, the design of the CPU is the main factor that influences the performance that can be achieved. Other components found in desktop computers and embedded platforms are:

- Storage and RAM: They are just as essential as a CPU as they transfer and hold the data that needs to be processed by the CPU.
- I/O ports: Depending on the purpose of the system I/O ports might include USB, ethernet, audio, display, or serial communication ports.
- Accelerators: They reduce the CPU's workload by offloading work to hardware, GPUs or FPGAs, that are designed to be more efficient at processing a particular set of tasks. They are not essential to the system, but usually provide a significant performance increase.

Embedded platforms have gained popularity due to their high reliability and robustness, small form factor size, energy efficiency, and rich set of I/O. These properties make embedded platforms ideal for implementing SLAM, as they can be used on a robotic platform such as a UAV. The biggest concern is whether the embedded platforms are powerful enough to implement SLAM.

2.3 Performance of SLAM on embedded platforms

Over the last decade, SLAM algorithms have been tested on high-end machines due to the computational complexity thereof [14], [20]. SLAM algorithms need to be implemented on a robotic platform, as specific scenarios do not allow high-end machines to be used. Thus, there is a need for SLAM algorithms to be able to run on embedded devices. There exists the possibility of choosing a SLAM algorithm with low computational complexity, but it usually comes with the cost of a lower level of accuracy. In mission-critical systems, a low level of accuracy is not acceptable as it could lead to a mission's failure. The first popular method for SLAM was the EKF-SLAM, which can achieve accurate results but is unsuitable for large-scale environments as the computational complexity scales linearly with the number of landmarks [14]. This, in turn, will also make an EKF-SLAM approach unsuitable for embedded implementation. Ma et al. [21] created a large-scale SLAM algorithm, which was only tested on a high-end Nvidia Titan GPU and a quad-core CPU. Their algorithm worked for the high-end system but would have probably failed on an embedded system constrained to real-time operation.

Nardi et al. [20], are the creators of one of the first benchmarking tools for SLAM. They first created SLAMBench 1.0, which provided a portable but untuned KinectFusion [22], implementation of SLAM in C++, OpenMP, Cuda, and OpenCL. They used the ICL-NUIM RGB-D dataset [12] to compare the performance of KiniticFusion on high-end computers and embedded devices. The high-end machine had a CPU with 4 cores running at 3.5 GHz and a Nvidia Titan GPU. This system was able to achieve a frame rate of 135 frames per second (FPS). The second device worth mentioning is the Nvidia TK1 embedded platform, which

has 4 cores running at 2.3 GHz and an Nvidia Tegra K1 GPU. This device was able to achieve a frame rate of 22 FPS. It is considerably lower than the high-end device, but the authors stated that this performance could almost be classified as real-time. Although the Nvidia TK1 had a lower performance than the high-end machine, it still outperformed the Odroid XU3 embedded device due to its faster GPU. The Ordroid XU3 has 8 cores running at 1.8 GHz and has an ARM Mali-T628-MP6 GPU and achieved a frame rate of 5.46 FPS. This shows that although satisfactory results can be obtained using SLAM on high-end machines, there is still a performance gap when using them on embedded devices. Bodin et al. [23] updated SLAMBench 1.0, called SLAMBench 2.0, to include more algorithms and showed that ORB-SLAM2 performed well when considering execution time, memory and accuracy. They also updated their benchmarking tool to include new performance metrics such as frame rate, absolute trajectory error (ATE), power consumption, computational speed, algorithm accuracy, and memory usage. The newest contribution is SLAMBench 3.0 which includes 14 different SLAM algorithms and 9 different datasets [24].

Abouzahir et al. [14] evaluated the embeddability of four popular SLAM algorithms: Monocular FastSLAM2.0, ORB SLAM, RatSLAM, and Linear SLAM. They tested the four algorithms on four embedded platforms: Nvidia Tegra TX1, Odroid XU4, i.MX5 Sabre Lite and the Pandaboard ES. They also compared their results with a high-end desktop and a laptop. They used a hardware-in-the-loop configuration to provide the embedded platforms with time-stamped and synchronized input data using a serial link, Ethernet, or ROS message communication. FastSLAM2.0 was able to run under real-time constraints on the high-end machine but could not run in real-time on any of the embedded devices. The high-end machine had a processing time of 35.45 ms, and the closest embedded competitor was the Nvidia Tegra TX1 with 63.88 ms. The same results were obtained for the other SLAM algorithms, with only the high-end machine capable of providing real-time constraints. The authors stated that the performance of an embedded SLAM relies on the nature of the algorithm and the architecture of the target embedded platform, so they suggested using a hardware matching methodology to get their real-time performances. They additionally adapted the FastSLAM2.0 algorithm to run specific code segments in parallel, which created the opportunity to utilize the parallel processing power of a GPU or FPGA. The authors tested their new implementation on the same systems mentioned above and added a new platform that connected an Arria 10 FPGA board to the PCIe bus of the high-end machine. This system had the highest performance for their parallel version of FastSLAM2.0. They suspect it is due to the high transfer rate that the FPGA has with the CPU through the PCIe bus.

TaoPeng et al. [19] evaluated two popular visual SLAM techniques, ORBSLAM2 and OpenSLAM, on the 3 Nvidia Jetson platforms. Their experience is that the Nvidia Jetson TX2 is the best suitable hardware for most visual SLAM applications, as it has a powerful CPU and GPU, has a moderate amount of RAM, and has good power efficiency [19]. They found that the Nvidia Jetson TX2 could run both visual SLAM algorithms at their real-time

constraints. They also investigated an alternative ORB-SLAM2 algorithm that is modified to use the power of a GPU. The GPU-accelerated ORB-SLAM2 algorithm had an increase of 10 FPS in execution while consuming less power than the original algorithm.

Another method that is used to run SLAM on an embedded system is by using cloud robotics, which allows a robot to benefit from the clouds' powerful computational power, providing the advantage of a rapid increase in data transfer rates. Mohanarajah et al. [25] used cloud robotics to create a 3D mapping of the environment in real-time. The processes that were sensitive to network delays were connected to high-bandwidth sensors that were executed locally on the robot, while the processes without hard real-time constraints were executed in the cloud. Although cloud robotics can be very useful, it is only practical when there is sufficient infrastructure to implement it on. As mentioned, it can also only be used for processes without hard real-time constraints.

From the literature that has been reviewed, it has become apparent that most SLAM algorithms are not capable of running on embedded systems as their resource requirements are too high. Luckily, there exist a few methods to improve the feasibility of SLAM being implemented on embedded devices, including:

- Matching embedded architecture types with specific algorithms [14].
- Optimizing the algorithm.
- Making use of parallel processing [14].
- Using strongly embedded devices such as FPGA with high processing speeds [14].
- Possibly using cloud robotics [25].

2.4 SLAM implementation challenges

All of the literature that has been reviewed in terms of testing and comparing SLAM on embedded devices made use of hardware-in-the-loop (HIL) configurations [14], [24]. This means that the embedded devices processed recorded datasets. The results might begin to differ if the SLAM algorithm is implemented on a complete robotic platform. The robotic platform may be a differential drive vehicle with a mounted monocular camera. SLAMBench 3.0 provides the perfect environment to do a complete comparative and quantitative study of all their Monocular SLAM algorithms, wherefrom the best performing algorithm can be implemented on a robotic platform.

Although loop closure can be used to mitigate scale drift, it is not possible in all realistic environments. Both LSD-SLAM [26], and ORB-SLAM2 [27], are examples of algorithms that

use loop closure to correct the scale drift that is associated with Monocular SLAM. Hull [28] used LSD-SLAM to create a real-time occupancy grid mapping but did not perform any scale correction. Within Hull's study, he stated that future work might include a scale correction method. One of the problems associated with Monocular SLAM algorithms is the scale ambiguity problem. The most popular method for mitigating the problem is sensor fusion, which is where other sensor data, such as data from an IMU, is fused with the results of the SLAM algorithm to correct the scale ambiguity [29], [30]. Other authors use a model-based approach for online scale estimation [31], [32]. Convolutional Neural Networks have also been used to mitigate the scale ambiguity problem [33]. The easiest method for mitigating the scale ambiguity problem is to use prior knowledge of the environment, such as well-placed landmarks or the size of some objects. This method proves to be successful but is very impractical.

The authors [14], [19], [20] mentioned above were able to implement various SLAM algorithms on an embedded platform. None of them, however, specifically looked into the factors that cause a decrease in the execution speed and what alterations can be done to the algorithm that might increase the execution speed on embedded platforms. The performance of most SLAM algorithms is highly dependent on the performance of the CPU. Generally embedded platforms are designed to be low-cost and as efficient as possible, which contradicts most desktop computers used for developing SLAM algorithms. The difference in CPU design approaches will explain why there is a difference when executing SLAM algorithms such as ORB-SLAM on embedded platforms compared to desktop computers. Thus, later in the Chapter the CPU is discussed in detail. The next Section looks into the Visual SLAM technique ORB-SLAM.

2.5 Visual SLAM

The main idea behind visual SLAM is to localize and construct a map using images. For this process to be successful, SLAM requires an algorithm framework. Figure 2 shows how the visual SLAM framework looks like after decades of research on SLAM [8].

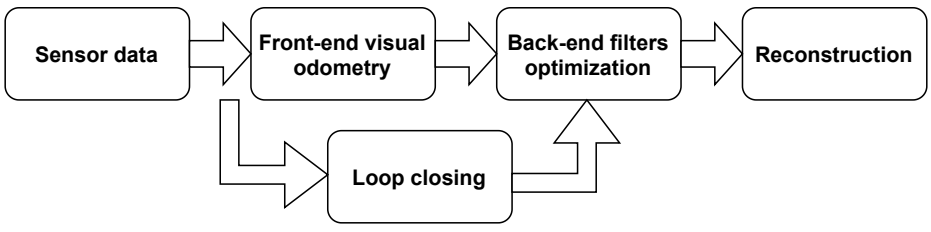


Figure 2: **Flowdiagram describing the classic visual SLAM framework [8]**

Sensor data acquisition in visual SLAM refers to the acquisition and preprocessing of camera images. Visual Odometry (VO) estimates the camera movement between adjacent frames, also

known as ego-motion, and generates a local map of the movements. VO is commonly known as the front-end of the SLAM algorithm. Back-end filtering/optimization receives camera poses at different time stamps and applies optimization to generate an optimized trajectory and map. This forms part of the back-end of the SLAM algorithm. Loop closing is used to detect loops in a trajectory and to reduce the accumulated drift. Reconstruction is responsible for reconstructing a map based on the calculated estimated camera trajectory.

2.5.1 ORB-SLAM

ORB-SLAM [16] is a feature-based monocular SLAM system that can operate in real-time in small and large indoor and outdoor environments and is very robust. It also includes loop closing, relocalization, and complete automatic initialization. ORB-SLAM uses the same features for all SLAM tasks such as tracking, mapping, relocalization, and loop closing. The point and keyframes of ORB-SLAM are selected using survival of the fittest, leading to reconstruction with excellent robustness and generating a compact map that only grows if the scene changes. ORB-SLAM uses ORB features [2] as a feature extractor, which allows for real-time performance while providing good invariance to changes in viewpoints and illumination differences.

ORB-SLAM is written to use three threads that run in parallel: tracking, local mapping, and loop closing. The tracking thread is responsible for localizing the camera with every frame and deciding when to insert a new keyframe. The local mapping thread is responsible for processing any new keyframes and performs local BA to achieve the optimal reconstruction of the camera pose. The loop closing thread searches for loops with every new keyframe.

Roughly two years later, ORB-SLAM2 [27] was released, which expanded their work beyond monocular SLAM into stereo and RGB-D SLAM. Another two years later, ORB-SLAM3 [34] was released, which is the first system to perform visual, visual-inertial, and multi-map SLAM with monocular, stereo, and RGB-D cameras, using the pin-hole and fisheye lens models. The fact that ORB-SLAM3 allows the user to use different SLAM types, with different sensor input types, and various camera models makes it an ideal SLAM algorithm to investigate.

2.5.1.1 ORB feature extraction

ORB uses FAST (Features from Accelerated and Segments Tests) keypoint detector [35] as well as a rotated BRIEF (rBrief) descriptor [36] to extract features from an image. The FAST detector finds regions in an image where features might be present, whereas the rBrief descriptor encodes a feature into a binary fingerprint used for matching.

Figure 3 aids in the explanation of how the FAST algorithm compares the brightness of a given pixel, p, to 16 pixels that are in a circle around the pixel p. Pixel p is a corner if there

exists a set of n contiguous pixels in the circle, which are all brighter or darker than pixel p. FAST has a high-speed test that first tests four pixels at 1, 9, 5, and 13. If the pixels at 1 and 9 are brighter or darker than pixel p, pixels 5 and 13 are also checked. Pixel p can only be a corner if at least three of the first four pixels are brighter or darker than pixel p. If the high-speed test is successful, the rest of the 16 pixels around pixel p is tested to determine whether pixel p is a corner or not. Since the FAST algorithm compares the brightness of the pixels in an image, the more pixels the image has, the more time is spent in the FAST algorithm.

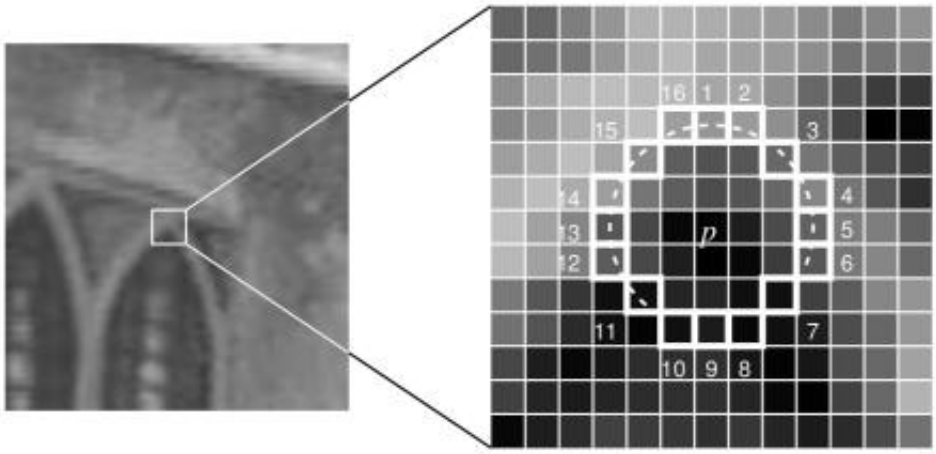


Figure 3: **FAST Test pixel comparison [35]**

FAST features do not have orientation components or multi-scale features. Thus, ORB uses a multi-scale image pyramid to make ORB more robust to scale and rotation. An image pyramid consists of a sequence of images of the same image at different resolutions. ORB then uses the pyramid to detect keypoints on the additional scale images using the FAST algorithm. The detected keypoints still need orientation, and ORB detects orientation by determining how the intensity changed around the keypoint using the intensity centroid. The intensity centroid assumes that a corner's intensity is offset from its centre, and this vector is used to suggest an orientation. Having more levels in the pyramid means that the FAST algorithm will be called more times during the execution, increasing the total feature extraction time.

Since the keypoints have some rotation and scale invariance, the feature descriptor can be determined using rBRIEF. BRIEF changes all keypoints into a binary feature descriptor, a feature vector that only contains 1's and 0's. Since BRIEF is not invariant to rotation, ORB uses rBRIEF (Rotation-aware BRIEF) to make the system more robust.

Discussing ORB-SLAM was the last part of the software aspect of the study. The rest of the chapter focuses on the hardware aspect, specifically the CPU.

2.6 The CPU

An embedded device is a computer system made up of a combination of systems. These sub-systems include processors, memory, and some input/output systems. Each sub-system plays a prominent role in the performance and efficiency of the embedded devices. This section will describe the terminology and details about the processor and memory sub-systems.

Processors are more commonly known as a Central Processing Unit (CPU). The CPU is responsible for executing instructions given to it, which can be arithmetic, logical, or controlling in nature, depending on the program being executed. How the program gets executed largely depends on the CPU's architecture.

CPU architecture consists of two major aspects: instruction set architecture (ISA) and micro-architecture as defined by Hennessy [37].

2.6.1 CPU instruction set architecture

The ISA is a well-defined interface between hardware and the lowest-level software that encompasses all the information necessary to write a machine language program. This includes instructions, registers, memory access, input or output (I/O), amongst others [38]. Hennessy [37] states that there are seven dimensions that characterize an ISA. These dimensions are:

1. **Class of ISA**: Nearly all ISAs today are general-purpose register architectures that only operate on either registers or memory locations. The x86 and ARM architecture use two different versions of this type of class, namely: register-memory ISAs and load-store ISAs, respectively. Register-memory ISAs can only access memory as part of many instructions, whereas load-store ISAs can access memory only with load-store instructions [37].

2. **Memory addressing**: Virtually all computers use byte addressing to access memory operands. Some architectures do require that the objects must be aligned to be accessed, which is generally faster than non-aligned operands [37].

3. **Addressing modes**: Specifies how the operands are addressed within an instruction. Different architectures have different types and amounts of addressing modes [37].

4. **Types and sizes of operands**: Most ISAs support operand sizes of 8-bit (ASCII characters), 16-bit (Unicode characters or half word), 32-bit (integer or word), 64-bit (double word or long integer). They also support IEEE 754 floating-point data types such as 32-bit (single precision) and 64-bit (double precision) operands [37].

5. **Operations**: The two most popular architecture designs are complex instruction set computer (CISC) and reduced instruction set computer (RISC). CISC architecture is

designed to complete a task in as few lines of assembly as possible, whereas RISC tries to utilize a small, highly-optimized set of instructions [39]. The most popular CISC and RISC architecture designs are the x86 and ARM architectures, respectively. Within the PC CPU industry, the x86 architecture has been dominant for at least 2.5 decades, while ARM has dominated the mobile and embedded market [40].

6. **Control flow instructions**: Most ISAs support conditional branches, unconditional jumps, procedure calls, and returns [37].

7. **Encoding an ISA**: There exist two types of encoding: fixed length and variable length. Fixed length encoding ensures simplified instruction decoding, whereas variable-length encoding takes fewer spaces [37].

Whereas the ISA defines the interface between physical hardware and software, the CPU micro-architecture describes the implementation and design of the physical CPU.

2.6.2 CPU micro-architecture

CPU micro-architecture includes high-level aspects of the CPU design, such as the memory system, how the memory is interconnected, and the design of the internal parameters of the CPU. These internal parameters include things such as pipeline length and layout, number and sizes of caches, cycle counts for individual instructions, and which optional features to implement [41].

The CPU micro-architecture defines the specific arrangement of registers, arithmetic logic unit (ALU), finite state machines (FSM), memories, and other logic building blocks needed to implement an architecture [42]. Any architecture can have many different micro-architectures, with different trade-offs between performance, cost, and complexity. They are all capable of running the same program, but their performance can differ due to their internal differences.

Harris [42] divides micro-architectures into two interacting parts: the datapath and the control unit. The datapath consists of a collection of functional units responsible for data processing operations on registers and buses. These functional units include the ALU, memories, registers, and multiplexers. The control unit receives the instructions from the datapath and tells the datapath how the instructions should be executed.

The three most common CPU micro-architectures are single-cycle, multi-cycle, and pipelined micro-architectures. The single-cycle micro-architecture is a straightforward design and executes a complete instruction in a single cycle. The multi-cycle micro-architecture, on the other hand, executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than more complicated instructions. This micro-architecture design has a reduced hardware cost by reusing expensive blocks such as addresses and memories. The pipelined micro-architecture is built upon the single-cycle micro-architecture with the added feature

of instruction pipelining. Instruction pipelining does increase the complexity of the micro-architecture design, but it is worthwhile since all commercial high-performance processors use pipelining today [42].

2.6.2.1 Advanced CPU features

CPU pipelining is done to increase the total utilization of the hardware by overlapping instructions during execution. It breaks down instruction processing into different stages, where each stage can process instructions simultaneously. Some situations keep an instruction from entering a specific pipeline stage, known as pipeline hazards.

There are three main types of pipeline hazards: data, structural, and control hazards. A data hazard occurs when the pipeline must be stalled because one instruction must wait for another instruction to complete, also known as instruction inter-dependency. A structural hazard is caused when two instructions need the same hardware resources at the same time. Control hazard comes from the need of the CPU to make a decision based on the results of instruction while others are executing.

When a pipeline hazard occurs, the pipeline is stalled, stopping all instructions until the stall is resolved. The length of the pipeline stall is determined by how long it takes the hazard to be resolved. Each pipeline hazard type can be avoided by using alternative techniques. A simple method to avoid data hazards is bypassing. This means that if the result of a specific instruction is needed further down the pipeline, the results are bypassed directly to that stage instead of being stored in a register. This is a common solution to prevent Read After Write (RAW) data hazards. A structural hazard can be avoided by adding more hardware resources, such as execution units. This increases the available hardware resources. Branch prediction is included in modern CPU designs to try and avoid control hazards by simply trying to predict the outcome for a branch before all necessary information is available.

Pipelining a CPU is one of the methods used to improve the performance of a CPU. Another method is to make it multiple-issue, also commonly known as superscalar. Superscalar CPUs have added hardware resources, such as execution units, to execute more than one instruction per clock cycle. This does not increase the CPU's processing speed but increases the throughput.

A superscalar CPU is defined as in-order execution if instructions are statically scheduled and out-of-order execution if instructions are dynamically scheduled [37]. The compiler is responsible for static scheduling, while hardware within the CPU is responsible for the dynamic scheduling of the instructions. Out-of-order execution can reduce stalls by deciding to execute other instructions when a dependency between instructions has been detected.

Over the years, CPUs have gotten smarter by guessing what instructions to execute next or what branch to pick. This method of guessing what instruction to execute is called speculation.

Speculation can be done by either the compiler or by the CPU itself. If the speculation is correct, it increases the CPU performance, but misspeculation requires that the pipeline be flushed or stalled, thus reducing the performance.

Almost all new CPUs are pipelined superscalar CPUs due to their ability to exploit potential parallelism between instructions. Parallelism uses hardware and software methods to force the CPU to do as much work as possible in one CPU cycle and can be divided into three primary levels: Data-level, Task-level, and Instruction-level parallelism (ILP).

Data-level parallelism is when several sets of data can be worked on simultaneously with an identical operation. Data-level parallelism is achieved through vector processing known as single instruction multiple data (SIMD).

Task-level parallelism comes from an application with multiple separate tasks that can be performed by various CPUs or threads of execution simultaneously. Thanks to the advancements in technology, CPUs evolved into multi-core and multi-threaded CPUs, which allowed us to exploit task-level parallelism. A multi-core CPU is a single integrated circuit with two or more separate CPUs called cores. Multithreading is the technique that enables the CPU to execute several tasks at the same time, also known as simultaneous multithreading (SMT). SMT increases CPU utilization by using more of the CPU's available resources. A program can only benefit from parallelism if the programmer writes the code to take advantage of multithreading. It should be noted that the nature of the program determines the level of parallelism due to inter-dependencies, the amount of communication between threads, and how well work is distributed between cores.

Instruction-level parallelism is the ability to execute multiple instructions alongside one another. Two primary methods can increase ILP in a CPU. The first method is to improve the overall pipeline depth to overlap more instructions. ILP is also increased by increasing the issue-width of the CPU.

Unfortunately, modern CPUs have far more machine parallelism than programs have available. The number of instruction dependencies, branches, and memory access all influence a program's level of parallelism. The maximum level of parallelism that a program can achieve is known as the ILP wall.

2.6.3 Memory subsystem

No matter the type of CPU micro-architecture, all CPUs are connected to a memory subsystem. The primary responsibility of the memory subsystem is to gather and store the necessary data needed by the CPU. Figure 4 shows how the memory subsystem is organized as a hierarchy.

The CPU iterates through the memory hierarchy, first looking for the data in the cache. If the data is not present in the cache, the CPU checks whether it is stored in the main memory.

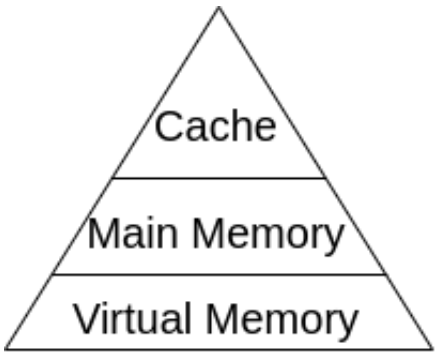


Figure 4: **Memory hierarchy**

If the data is not present in either the cache or the main memory, the CPU fetches the data from the virtual memory. Virtual memory is stored on the hard drive, which provides a large amount of capacity. Due to the slow nature of a hard drive, virtual memory has the longest access time. Main memory provides an intermediate step from virtual memory, having less capacity but faster access time. Cache has the smallest capacity and the fastest access time, but its high-speed capability is the most expensive per gigabyte.

The difference between access times for each hierarchy level is influenced by the technology used to create the memory type. Cache memory is a high-speed memory type made out of static RAM (SRAM). Main memory, on the other hand, is made out of dynamic RAM (DRAM), which is less expensive to produce and is inherently slower than SRAM, usually one or two orders of magnitude longer than the CPU processor cycle time.

Dividing the memory in a hierarchy decreases the CPU's production cost and takes advantage of the principle of locality, which increases the performance of a CPU. Locality can be described as the tendency of the CPU to access the same address space over a short period. There are two types of locality: temporal- and spatial locality. Temporal locality is the principle that states that if a location is referenced, it will be referenced again soon. Spatial locality states that if a location is referenced to, locations nearby will be referenced soon [38].

The performance of the memory system can be summarized using three measurements: miss rate, hit rate and average memory access time (AMAT). The miss rate is quantified by the number of misses divided by the total number of memory accesses. The hit rate is the total number of hits divided by the total number of memory accesses. If the cache contains the necessary data, it is referred to as a hit, if the data is not in the cache, it is a miss. Equation (2.6.1) and (2.6.2) defines miss rate and hit rate respectively [42].

$$Miss\ Rate = \frac{Number\ of\ misses}{Number\ of\ total\ memory\ accesses} = 1 - Hit\ Rate \tag{2.6.1}$$

$$Hit\ Rate = \frac{Number\ of\ hits}{Number\ of\ total\ memory\ accesses} = 1 - Miss\ Rate. \tag{2.6.2}$$

AMAT is the average time a CPU must wait for memory to load or store an instruction and is defined by equation (2.6.3) [42],

$$AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}t_{VM}), \tag{2.6.3}$$

where t_{cache}, t_{MM}, t_{VM} are the access times of the cache, main memory and virtual memory respectively, MR_{cache} and MR_{MM} are the cache and main memory miss rates.

2.6.3.1 CPU cache

The concept of main memory is straightforward; the larger and faster it is, the better performance can be expected for a CPU. On the other hand, CPU cache is a bit more complex as there exist different cache design choices that have a direct influence on the performance of the CPU. The CPU cache improves the performance by predicting what data will be needed next using temporal and spatial locality. Caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N). Unfortunately, one cache design does not guarantee the same performance for all applications, as the nature of the application influences the performance of the cache design.

The group of words that cache fetches from memory is called a cache block or cache line. The number of words in the cache block, b, is called the block size. A cache of capacity C contains $B = C/b$ blocks. A cache is further organized into S sets, each set holds one or more data blocks. Caches are classified based on the number of blocks in a set. A direct-mapped cache contains exactly one block so that the cache has $S = B$ sets. This means that a particular main memory address maps to a unique block in the cache. N-way set associative cache contains N blocks. Thus, the address maps to a unique set with $S = B/N$ sets. The data from the address, however, can go into any of the N blocks in that set. A fully associative cache has only one set, $S = 1$. This means the data can go in any of the B blocks in the set.

Direct-mapped caches are usually easier and cheaper to build but are prone to conflicts. A conflict is when two recently accessed addresses map to the same cache block, and the most recently accessed address replaces the previous one from the block. A high number of conflicts means that the cache will have a large miss rate, which is the way that the number of conflicts is quantified. The higher the cache hit rate is, the better the performance of the cache. Set associative caches generally have lower miss rates than direct maps of the same capacity as they experience fewer conflicts. However, set-associative caches are slower and more expensive to build due to additional comparators and output multiplexers. Set associative caches also have the added complexity of how to replace the data in the cache if all

the ways are full. Fully associative caches have the least conflict misses for the same capacity, but they require even more hardware. They are best suited for small caches due to the large number of comparators needed. Most modern CPUs use set-associative caches due to their balance between performance and cost [42].

Regardless of the degree of the cache's associativity, the cache's performance depends on the block size of the cache. Increasing the cache block size means more words are fetched, increasing the chance of a hit on the subsequent access due to spatial locality. However, larger block sizes mean that the cache has fewer blocks, leading to more conflicts, increasing the miss rate. Increased block size also means that the time it takes to fetch a cache block is increased. Associative caches must also decide which block to evict when a cache set is full. Most of the associative caches use the least recently used (LRU) replacement policy as temporal locality suggests that the best choice is to replace the least recently used block, as it is least likely to be used again soon [42].

Most modern systems use two to three levels of cache to try and exploit certain cache design choices that take advantage of spatial and temporal locality. The first-level (L1) cache is usually very small to provide a high-speed access time. The second-level (L2) cache is generally larger in capacity, thus being slower than the L1 cache. The same principle applies to the L3 cache.

Overall, cache misses can be reduced by changing the capacity, block size, and/or associativity. Cache misses can be divided into three groups: compulsory, capacity, and conflict. The first request to a cache block is called a compulsory miss, as the cache block is empty the first time, regardless of the cache design. Capacity miss is caused by the cache being too small to hold all the concurrently used data. Conflict misses are caused when multiple addresses map to the same set and replace blocks that are still needed. Changing the cache parameters affects one or more types of cache misses. Increasing the cache capacity can reduce conflict and capacity misses, but it does not alter compulsory misses. Increasing the block size can reduce compulsory misses but can increase conflict misses.

2.7 CPU performance analysis

The primary role of a CPU is to execute as many instructions as possible in the shortest amount of time. This measure is usually known as the performance of the CPU. The peak performance that a CPU can achieve is determined by a combination of software and hardware characteristics.

From a software point of view, ignoring all hardware characteristics, the performance of a CPU during program execution is determined by the CPU ISA and the program characteristics itself. The program characteristics are dependent on the mix of instructions needed to execute the program. This is commonly known as the instruction mix.

Instructions can be divided into three popular types: Arithmetic, Memory, Control-flow. Arithmetic instructions are responsible for performing arithmetic operations such as addition, subtraction, and multiplication. Memory instructions are responsible for loading and storing data. Control-flow is responsible for modifying the program counter. This includes branch instructions, function calls instructions, and return instructions.

On the other hand, the instruction mix is determined by ISA, the compiler used, and the type of problem the program tries to solve. Usually, programmers can not make changes to the ISA, but they can use their knowledge about the ISA to tailor the program to match the ISA they are working on. Several compilers exist, it is again the programmer's job to decide what compiler will best fit their current scenario. A significant factor that influences the software performance on the CPU is the amount of instruction dependency present in a specific program [43].

From a hardware perspective, the performance of a CPU is mainly dependent on the CPU micro-architecture and other hardware characteristics such as core count and clock speed. Other metrics included load latencies, cache miss rate, and branch prediction [43].

Before the performance of a CPU can be determined, the term "performance" needs to be defined for this context. Harris [42] states that the only true gimmick-free way to measure the performance of a CPU is by measuring the execution time of a program of interest to you. By this measure, the CPU with the best performance is the CPU that executes the program the fastest. The second solution is to measure the total execution time of a collection of programs that resemble the type of program you plan to run. This collection of programs are called benchmarks and is commonly used to define the performance of a CPU. The execution time of a program, measured in seconds, is defined by

$$Execution\,Time = \#instructions \times CPI \times Clock\,cycle\,time. \tag{2.7.1}$$

A program's number of instructions depends highly on the CPU architecture and the compiler used. Some architectures, such as CISC, have complicated instructions that do more work, thus reducing the overall instructions. However, these complex instructions are usually slower to execute. The cycles per instruction (CPI) is the number of clock cycles required to execute an average instruction. The CPI is also commonly described as instructions per cycle (IPC), which is the reciprocal CPI and represents the throughput of the CPU. CPI is primarily influenced by the CPU micro-architecture design choices, the memory subsystem it is connected to, and the workload of the program being executed. The clock cycle time is the rate at which the CPU does work. It is commonly referred to as the clock period and is usually measured in seconds. The inverse of clock cycle time is known as the clock rate of the CPU [38].

Execution time can also be defined in different ways depending on what is measured. The most common definition is wall-clock time, response time, or elapsed time, and it describes the latency to complete a task.

A common way of comparing design alternatives, whether it is software or hardware related, is to relate the performance of two designs by saying X is n times faster than Y, meaning in terms of execution time that X executes in less time than Y, as indicated by

$$n = \frac{Execution\,time_Y}{Execution\,time_X}. \tag{2.7.2}$$

2.8 Critical literature review

There is performance degradation when implementing SLAM on an embedded platform, which can be linked to the hardware limitations of embedded platforms and the difference in CPU designs. Perhaps the most considerable difference between embedded platforms and desktop computers is the use of the RISC ISA of ARM CPUs commonly used to create embedded platforms. Several authors [44]–[46] showed that RISC-based CPUs closed the performance gap between RISC and CISC CPUs with lower power consumption, thus increasing the feasibility of implementing SLAM on embedded platforms.

It has also been shown that ORB-SLAM can be implemented on embedded platforms and provides acceptable performance in speed and accuracy. ORB-SLAM also offers the opportunity to investigate multiple SLAM approaches such as monocular, monocular-inertial, stereo, stereo-inertial, and RGB-D SLAM. These properties of ORB-SLAM make it an ideal SLAM algorithm for this study. Since ORB-SLAM uses the CPU for execution, it is necessary to understand how ORB-SLAM interacts with the CPU and how the CPU works to potentially increase the performance of this SLAM algorithm on an embedded platform.

This study will focus on testing ORB-SLAM3 [34]. Since monocular SLAM is the least computationally expensive, this study will focus on monocular SLAM instead of stereo or RGB-D SLAM. The scale ambiguity problem associated with monocular SLAM will be mitigated by using monocular-inertial SLAM.

From the literature, it was decided that the Nvidia Jetson TX2 [47] will be used as the embedded platform to test the performance of ORB-SLAM3. It was chosen as it has powerful hardware with two CPU clusters and a dedicated GPU, and similar studies have been completed using this platform or the older version of it [14], [19], [20]. Thus, the results that are obtained can be compared to the previous studies and additional testing on the hardware can be done. Additional hardware includes the Raspberry Pi 3B+ [48] and Raspberry Pi 4B [49]. This additional hardware was chosen due to time and budget constraints as the were already available to use. The hardware can be classified into performance groups due to their available processing power: high performance (Nvidia Jetson TX2), average performance (Raspberry Pi 4B), and low performance (Raspberry Pi 3B+). These embedded platforms will be used to generate a dataset to create a prediction model to estimate the performance

that ORB-SLAM can achieve on embedded platforms.

Since the performance that ORB-SLAM3 can achieve on an embedded platform is linked to the CPU, the working principles of a CPU need to be investigated. Not only is it essential to understand how a CPU works, but it is also necessary to understand how software interacts with a CPU and how performance is defined. It is also necessary to investigate how ORB-SLAM3 works to help with any troubleshooting during the implementation thereof.

Several methods have been used to increase the performance of ORB-SLAM, such as using FPGA's [50], [51] or GPUs [19] for the feature extractions to decrease CPU load. Some authors [52], [53] modified portions of the ORB-SLAM code to get an increase in performance. Although using these methods provided an increase in performance, non investigate what optimizations can be done to the ORB-SLAM code to increase the performance thereof without having to rewrite any code. Thus, there is a need to profile [54] the execution of ORB-SLAM to identify what portions of code can be optimized and what parameters can be varied to increase the performance.

Chapter 2 showed that CPUs are very complex hardware, and various factors influence the performance they can achieve [37]. These factors include CPU ISA, pipeline depth, superscalar order, in-order or out-of-order execution, and the construction of the memory hierarchy. Along with the hardware characteristics, the software being executed also influences the performance that can be achieved. As the complexity of CPU designs and software increased, it became inherently more challenging to predict the performance of a CPU. Thus, CPU measurements are taken during execution to determine its performance. The best way to measure the performance of a CPU is to measure the execution time of the software being executed. This can be achieved by profiling the execution of the program. Profiling [54] is a form of Software Performance Engineering (SPE) under a measurement-based approach [55]. It can be used to measure the program's execution speed and identify any bottlenecks within a program. Software such as ARM Map [56] can easily be used to profile an application for bottlenecks, identify any poorly optimized code, or identify variables that can be adjusted to increase the performance of an application.

ORB-SLAM is a state-of-the-art SLAM algorithm that can achieve real-time execution with high accuracy. ORB-SLAM uses ORB features, made up of the FAST detector and the rBrief descriptor, to extract features from the environment using images. Although ORB-SLAM can achieve real-time execution speeds on high-end desktop computers, it fails to achieve real-time execution speeds on an embedded platform. SLAM needs to be implemented on a UAV that has limited hardware resources in the form of an embedded platform. Thus, there is a need for SLAM algorithms, such as ORB-SLAM, to execute in real-time on an embedded platform. ORB-SLAM's tracking thread is the most compute-intensive thread [16], [57]. The tracking thread needs to be investigated to see whether parameters can be changed to decrease the computational load or whether optimizations to the code can be made to increase the performance thereof.

The ORB-SLAM algorithm uses the CPU to execute its code, and it was shown that modern CPU designs are very complex and influence the performance an application can achieve. Algorithms such as ORB-SLAM are commonly developed and tested on desktop machines with a CISC ISA CPU. In contrast, the study aims to implement and test the ORB-SLAM algorithm on an embedded platform that has a RISC ISA CPU. In the past, there was a performance gap between RISC and CISC, but recently the gap has been reduced [44]–[46], making it more feasible to try an implement compute-intensive programs on RISC ISA CPUs. The difference in ISA might lead to unoptimized code in the ORB-SLAM algorithm that might influence the performance it can achieve on the embedded platform. Thus, Chapter 3 investigates the specific hardware used in the study and investigates the baseline performance of ORB-SLAM.

Chapter 2 highlighted that ORB-SLAM can be implemented on embedded platforms such as the Nvidia Jetson TX2 [14], [19], [20]. Using the technical reference manuals [58]–[60] of the Nvidia Jetson TX2 ORB-SLAM3 [34] can be profiled using ARM Map [56]. The profiling process should help to identify any bottlenecks, unoptimized code, and variables within ORB-SLAM3 that can be used to create a prediction model to estimate the performance that ORB-SLAM3 can achieve on embedded platforms. The profiling of ORB-SLAM3 is only done on the Nvidia Jetson TX2 as ARM Map is only licensed to be used on the Nvidia Jetson TX2.

Chapter 3 - Software and Hardware implementation

3.1 Introduction

Chapter 2 highlighted that modern-day CPU designs could be very complex, and the design choices influence the performance that a CPU can achieve when executing applications. How an application is written and compiled also influences the performance of the CPU. Thus, there needs to be a clear understanding of the hardware being used and how applications interact with the hardware.

This chapter covers the hardware-specific details of the Nvidia Jetson TX2, Raspberry Pi 3B+ and Raspberry Pi 4B. The chapter also covers the in-depth software setup used to implement ORB-SLAM3 on the embedded platforms, specifically pointing out software versions and settings used in the study. After the hardware and software details are discussed, ORB-SLAM3 is implemented on the Nvidia Jetson TX2 to ensure that ORB-SLAM3 works as expected and to serve as a baseline measurement for the rest of the study. ORB-SLAM3 is containerized for ease of implementation on the other platforms and to ensure the ORB-SLAM3 environment is identical, with only the hardware being different between execution on the different embedded platforms.

3.2 Hardware

This section will describe the different embedded platforms used in this study, specifically looking into the CPU architectures and memory subsystems.

3.2.1 Nvidia Jetson TX2 CPU cluster

This study will use the Nvidia Jetson TX2 developer kit as practical hardware. The Nvidia Jetson TX2 has a dual-core Nvidia Denver 2 CPU and a quad-core ARM Cortex-A57 MPCore CPU. The Nvidia Denver 2 CPU will be referred to as the Denver 2, and the Cortex-A57 CPU will be referred to as the A57. The Nvidia Jetson TX2 also has a Nvidia PASCAL GPU architecture with 256 Nvidia CUDA cores. It has 32GB eMMC 5.1 onboard storage, 8GB 128-bit 1866MHz LPDDR4 memory, and a Thermal Design Power (TDP) of 15W [47].

The Nvidia Jetson TX2 has two CPU clusters connected by a high-performance coherent interconnect fabric designed by Nvidia that enables simultaneous operation of both CPU clusters for a true heterogeneous multi-processing environment. The Denver 2 cluster is optimized for single-threaded performance, and the A57 cluster is optimized for multi-threaded applications and lighter CPU loads. Both CPU clusters include the full implementation of the ARMv8 ISA [47].

3.2.1.1 Nvidia Denver 2 CPU cluster

The Denver 2 is developed by Nvidia and is an identical implementation of the ARMv8 architecture with Nvidia's optimizations. The Denver 2 is a 7-wide in-order superscalar CPU with Nvidia dynamic code optimization (DCO). It also has dynamic branch prediction with a branch target buffer and global history buffer, a return stack buffer, and an indirect predictor. The Denver 2 has a 128 KB 4-way set associative L1 instruction cache, 64 KB 4-way set associative L1 data cache, and a 2 MB 16-way set-associative shared L2 cache [58].

It is important to note that the Denver 2 operates in one of two modes: ARM hardware-decoder mode and the optimized microcode mode. In the ARM hardware-decoder mode, the front-end fetches the instructions for the L1 instruction cache. In this mode, the Denver CPU acts as a 2-way superscalar in-order CPU and has an acceptable level of performance and efficiency for code that Nvidia's DCO has not yet optimized. Once the DCO has optimized the code, the Denver 2 CPU is in the optimized mode. The DCO extracts parallelism from the instruction stream, thus creating densely packed micro-operation bundles optimized to run on the Denver 2 CPU. This can reduce dependency stalls and eliminate redundant instructions, thus in this mode, the Denver 2 can achieve seven instructions per cycle [61].

The Denver 2 has seven execution units: two integers, two floating-point/NEON, two load/store, and one branch execution unit. The load/store units can also execute integer operations, enabling the DCO to execute up to four integer instructions when no memory instructions must be performed.

3.2.1.2 ARM Cortex-A57 CPU cluster

The A57 is developed by ARM and is the full implementation of ARMv8 architecture. The A57 is a 3-wide superscalar variable-length out-of-order CPU. It also has dynamic branch prediction with a branch target buffer and global history buffer RAMs, a return stack, and an indirect predictor. The cache is made up of a fixed 48 KB 3-way set-associative L1 instruction cache, 32 KB 2-way set-associative L1 data cache, and a 2 MB L2 16-way set-associative shared cache [59].

The A57 CPU has eight execution units: two integers, one multi-cycle, one load, one store, and two floating-point and two NEON execution units [60].

3.2.2 Raspberry Pi 3B+ (Cortex-A53)

The Raspberry Pi 3B+ is designed with the Broadcom BCM2837 system-on-chip (SoC) and includes four high-performance ARM Cortex-A53 cores running at 1.2GHz with 32kB Level 1 and 512kB Level 2 cache memory. It has a VideoCore IV GPU, a total of 1GB LPDDR2

memory and a TDP of 7W [48].

The A53 is developed by ARM and fully implements ARMv8 architecture, with an in-order pipeline with symmetric dual-issue of most instructions. The cache comprises a fixed 16 kB 2-way set associative L1 instruction cache, 16 KB 4-way set associative L1 data cache, and a 512kB L2 16-way set-associative shared cache [62].

The A53 CPU has six execution units: two integers, two load/store, one floating-point and one NEON execution unit [62].

3.2.3 Raspberry Pi 4B (Cortex-A72)

The Raspberry Pi 4B contains a Broadcom BCM2711 SOC, which consists of a Quad-core Cortex-A72 clocked at 1.5Ghz and 4GB LPDDR4-3200 SDRAM [49].

The A72 is developed by ARM and fully implements ARMv8 architecture. The A72 is a 3-wide Superscalar, variable-length, out-of-order. It also has dynamic branch prediction with a branch target buffer and global history buffer RAMs, a return stack, and an indirect predictor. The cache comprises a fixed 48 KB 3-way set-associative L1 instruction cache, 32 KB 42way set-associative L1 data cache, and a 512 KB L2 16-way set-associative shared cache [63].

The A72 CPU has nine execution units: three integers, one multi-cycle, one load, one store, and two floating-point and two NEON execution units [63].

3.2.4 CPU comparisons

Table 1 directly compares the different CPUs. Out of the four CPUs, the Denver 2 has the highest superscalar width, meaning it can execute the most instructions per cycle (seven). However, this is only possible if the Denver 2 is in DCO mode. In this mode, it shares the lowest superscalar width with the A53. The A57 and A72 both have a superscalar width of three. These instructions per cycle can not always be achieved as it is highly dependent on the program nature, but the Denver 2 does have a higher performance potential due to the larger width of the CPU.

The A72 and A57 are “out-of-order” processors, while the Denver 2 and A53 are “in-order” processors. “Out-of-order” CPUs generally cope better with dependency problems within code than “in-order” CPUs, but the Denver 2 CPU does mitigate this problem through its DCO.

The Denver 2 has a larger L1 cache due to the storage requirements of Nvidia’s DCO. In general, larger caches reduce capacity and conflict misses but increase the access time of the cache. The higher associativity of the Denver 2 should also further reduce the number of conflict misses but will have an increase in access time [37], in contrast, the A53 will

experience the most conflict misses but have the lowest access time. Overall the Denver 2 should experience lower miss rates but at the cost of a more significant miss penalty.

Theoretically, the Denver 2 should perform better than the other CPUs. However, the results might differ in practice due to other architecture designs or software implementation and optimization.

Table 1: **Comparison between embedded platform CPUs**

	Cortex-A57	**Denver 2**	**Cortex-A53**	**Cortex-A72**
ISA	ARMv8	ARMv8	ARMv8	ARMv8
Superscalar	3	7 with DCO (2 without)	2	3
Execution order	Out-of-order	In-order	In-order	Out-of-order
L1 instruction cache (set-associative)	48 KB 3-way	128 KB 4-way	2 KB 2-way	48 KB 3-way
L1 data cache (set-associative)	32 KB 2-way	64 KB 4-way	16 KB 4-way	32 KB 2-way
L2 shared cache (set-associative)	2 MB 16-way	2 MB 16-way	512 KB 16-way	512 KB 16-way
Operating frequency	2 GHz	2 GHz	1.2 GHz	1.5 GHz

3.3 Software - Development environment

This section describes the software used for this study. Specifically, looking at the EuRoC MAV dataset used in the study, the ORB-SLAM3 dependencies, and supporting software to aid in successfully implementing ORB-SLAM3.

3.3.1 EuRoC MAV dataset

This study will use the EuRoC MAV dataset [11] for testing the performance of ORB-SLAM3. The EuRoC dataset was chosen as the dataset provides synchronized stereo images, IMU measurements, and accurate ground truth measurements. The dataset has also been used in several other publications, and ORB-SLAM3 provides examples that use the EuRoC dataset. The EuRoC dataset was recorded using an AscTec Firefly hex-rotor equipped with two global-shutter monochrome cameras recording images at a rate of 20 Hz, and an IMU system recording angular rates and specific force measurements at 200 Hz. The ground truth was measured using the Leica Nova MS50 laser tracker that provides millimeter accuracy at

a rate of 20 Hz and a Vicon motion capture system that offers 6D pose measurements of a coordinate frame at a rate of 100 Hz.

The EuRoC dataset provides two types of datasets and is defined as the machine hall and vicon room datasets. The machine hall dataset contains the 3D position ground truth from the Leica system and focuses on evaluating visual-inertial SLAM algorithms in a more realistic industrial scenario. The machine hall environment is unstructured and cluttered, which renders the dataset challenging. The Vicon room dataset contains the 6D pose ground truth obtained from the Vicon motion capture system and an accurate 3D point cloud of the environment extracted using the Leica system. The Vicon room has placed obstacles to render the reconstruction more challenging and provide the scene with more visual textures.

Each dataset type has several sequences, and each sequence has a defined difficulty. Table 2, shows the eleven sequences available in the EuRoC MAV dataset with their difficulty level as defined by the authors. For more details about the sequences, please refer to the EuRoC MAV dataset article [11].

Table 2: **EuRoC MAV dataset sequences and difficulty classification**

Dataset type	Sequence name	Difficulty
Machine hall	MH01	Easy
Machine hall	MH02	Easy
Machine hall	MH03	Medium
Machine hall	MH04	Difficult
Machine hall	MH05	Difficult
Vicon room	V101	Easy
Vicon room	V102	Medium
Vicon room	V103	Difficult
Vicon room	V201	Easy
Vicon room	V202	Medium
Vicon room	V203	Difficult

3.3.2 ORB-SLAM3

The creators of ORB-SLAM3 state that it is the first real-time SLAM library to perform visual, visual-inertial, and multi-map SLAM with monocular, stereo, and RGB-D cameras, using either the pin-hole or fisheye lens models. They claim that ORB-SLAM3 is as robust as the best SLAM systems available in the literature and is more accurate in all sensor configurations. Campos et al. [34] provide examples to run ORB-SLAM3 on the EuRoC and TUM-VI datasets. ORB-SLAM3 is released under the GPLv3 license. ORB-SLAM3 was built and tested on Ubuntu 16.04 and 18.04 and recommends using a powerful CPU to ensure

real-time performance. It is written in C++11 standard and depends on other libraries such as Pangolin, OpenCV, Eigen3, DBoW2, and g2o. Additional libraries include Python and ROS.

Pangolin [64] is only used for visualization and to act as a user interface. OpenCV [65] is used within ORB-SLAM3 to manipulate images and extract features and has been tested with OpenCV 3.2.0. Eigen3 [66] is a C++ template library that is used for linear algebra, such as matrices, vectors, and numerical solvers. ORB-SLAM3 uses a modified version of DBoW2 [67] to perform place recognition. The modified g2o [68] library is used to perform non-linear optimizations throughout the entire ORB-SLAM3 system. Python [69] is only required to calculate the alignment of the estimated trajectory with the ground truth. ORB-SLAM3 can also be integrated with Robot Operating System (ROS) [70].

ORB-SLAM3 runs in three threads, and since the Nvidia Jetson TX2 has two CPU clusters, the Nvidia Jetson TX2 schedular is responsible for choosing what threads to place on a specific CPU. A user can also decide which CPU has to run a thread by setting the CPU affinity. This can be done in the program itself or by binding the threads to a particular CPU by using taskset. Taskset [71] is a Linux command that allows the user to assign the CPU affinity to a program or process.

3.3.2.1 ORB-SLAM3 performance metrics

Along with the estimated trajectory, ORB-SLAM3 outputs various metrics measured during execution. These metrics are saved in text files, which can be used for debugging and determining the performance of ORB-SLAM3.

The execution time mean text file contains the average time statistics for the major components associated with ORB-SLAM3. These components include tracking, local mapping, bundle adjustment, and place recognition. The primary objective of this study is to predict the performance of ORB-SLAM3 on an embedded platform, with a specific focus on the execution time. In mission-critical applications such as search and rescue, reconnaissance, and attack missions, SLAM applications are only valuable if they can operate in real-time.

The way ORB-SLAM3 is written, the tracking thread is the main thread of execution, and if the main thread is stalled or slow, the overall performance of ORB-SLAM3 will be slow. Thus, the performance metric important to us is the tracking thread execution time. Thus, the output of the model created in this study will focus on predicting the mean tracking time of ORB-SLAM3 on an embedded platform.

3.4 Initial implementation

ORB-SLAM3 has a set of prerequisites that need to be satisfied before being implemented on any device. These prerequisites are provided on the ORB-SLAM3 GitHub page [72], along with all the necessary steps to implement it on a system. A laptop will be used as a host computer to connect remotely to the embedded platforms through the SSH protocol. Visual Studio Code [73] will serve as an interface between the laptop and the embedded platforms as it provides a remote connection between the host and client with SSH, SFTP, and code editing functions.

The initial implementation of ORB-SLAM3 will be installed and tested on the Nvidia Jetson TX2. Once ORB-SLAM3 is up and running, it will be containerized so that the same software executes on all the embedded platforms. The initial implementation is only on the Nvidia Jetson TX2 in-order to build a containerized version of ORB-SLAM3 for ARM64 platforms. Once ORB-SLAM3 is containerized it will be used on the Raspberry Pi 3B+ and Raspberry Pi 4B to generate a dataset to be used in creating the prediction model.

Initially, ORB-SLAM3 v0.3 was configured and built to use OpenCV 3.2.0 along with Pthreads parallel framework, with the following flags enabled as suggested by Nvidia: ARM NEON (SIMD) flag enabled, optimization flag set to -O3, tune flag (-mtune) set to A57, and architecture flag (-march) set to A57. The initial test was conducted on only the A57 CPU cluster to serve as a baseline and check whether ORB-SLAM3 works and whether the output is as expected. This is achieved by calculating the ATE between the estimated and measured trajectories. The RMSE ATE between the estimated and measured trajectories of sequence V201 is 0.06m, and the comparison between the two trajectories is illustrated in Figure 5. The initial build of ORB-SLAM3 had an average total tracking time of 100.54 ms. The performance of the initial ORB-SLAM3 implementation is provided in Table 3.

Table 3: **ORB-SLAM3 initial implementation performance**

Total Tracking Time (ms)	ORB Extraction (ms)	Local Map Track (ms)	ATE RMSE (m)
73.70	34.62	24.51	0.159

3.5 Containerizing ORB-SLAM3

Since the study aims to model the performance of ORB-SLAM3 on different embedded platforms, it will help containerize ORB-SLAM3. Containerized applications run in isolated environments called containers. Containers encapsulate an application with all its dependencies, including system libraries, binaries and configuration files [74]. Docker is a

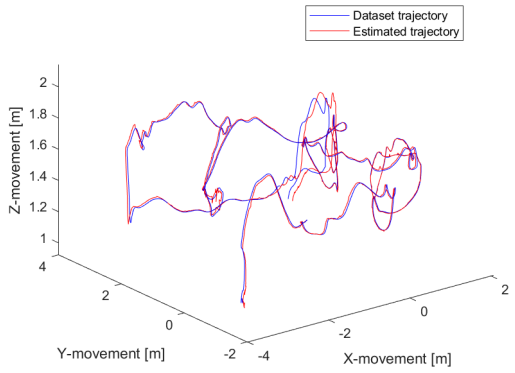


Figure 5: **Graph comparing the estimated trajectory of ORB-SLAM3 with the measured trajectory of sequence V201 of the EuRoC MAV dataset**

popular platform that is used to containerize applications. It allows the ability to create, store and share containers easily [75].

To create a container of ORB-SLAM3 with Docker, a Dockerfile has to be created. The Dockerfile contains the usual steps to install ORB-SLAM3 on any operating system or platform. The base image for the Docker container is Ubuntu 18.04 since it is the same OS used to develop ORB-SLAM3. All the necessary libraries and tools are installed as described in the ORB-SLAM3 README.md. OpenCV4.1.1 is built from source code with the minimum needed packages with SIMD enabled and the highest level of optimization for compiler settings. ORB-SLAM3 is compiled with the same settings. After everything has been built, the source code is removed, and only the generated binaries and libraries are kept to keep the size of the Docker image to a minimum. The necessary scripts and accompanying data are copied onto the image so that the same version of ORB-SLAM3 is used across all platforms and the same testing script is used. The Dockerfile and scripts used for testing are provided in Appendix A.

The created Docker container can now be used on any embedded platform with the ARM64 architecture. Unfortunately, because some compiler flags are specific to the ARM64 platform, the image can not be used without modification on an AMD64 platform. This study will only focus on using the ARM64 docker image as all of the embedded platforms have an ARM64 CPU architecture. The equivalent Dockerfile for an AMD64 image is also provided in data files for this study and it is not used in this study but can be used to run ORB-SLAM3 in a container on AMD64 platforms.

Docker also provides the ability to allocate a different amount of system resources to an image, which means that we can restrict the number of CPU cores or frequencies of the CPU and change the amount of available RAM. This provides the opportunity to emulate platforms with a different number of cores, CPU frequencies or amount of RAM.

3.6 Conclusion

This chapter highlights that the embedded platforms have complex CPUs, and it is expected that the different CPUs will achieve different performances when executing ORB-SLAM3 due to their CPU micro-architecture differences.

Building applications has become a non-trivial task as there exist multiple compilers, each with its compiler flags that can influence the application's performance. Thus, ORB-SLAM3 is built with the default compiler with the recommended flags set as provided by the Nvidia developers. Changing compilers or compiler flags can change the performance of ORB-SLAM3 but will not form part of this study.

It became clear that it is possible to implement ORB-SLAM3 on the Nvidia Jetson TX2, as shown in the previous section. The created Docker image of ORB-SLAM3 can now be used to implement it on multiple embedded platforms without going through the tedious steps to compile and install all the necessary components. The abstraction that Docker provides also ensures that from a software point of view, the execution of ORB-SLAM3 should be identical. The only differences between executing on different platforms will be the CPU micro-architecture differences.

Chapter 4 - Performance profiling

4.1 Introduction

In Chapter 3, the development environment for ORB-SLAM3 was set up and containerized, and ORB-SLAM3 was implemented on the Nvidia Jetson TX2. It was mentioned that the study aims to predict the performance of ORB-SLAM3, and the specific performance it aims to predict is the average total tracking time of ORB-SLAM3. The profiling of ORB-SLAM3 will provide insight into what metrics can be used as inputs to the model.

This chapter profiles ORB-SLAM3 on the Nvidia Jetson TX2 to identify the bottleneck. After that, the source code of ORB-SLAM3 and OpenCV is investigated. The actual execution of ORB-SLAM3 will also be dumped, using GDB Debugger, to see what instructions are executed during ORB-SLAM3

Section 4.2 describes how ORB-SLAM3 is profiled and what measurements are taken during the profiling of ORB-SLAM3. In Section 4.3, ORB-SLAM3 is profiled, and the bottleneck is identified. The program characteristics and performance of ORB-SLAM3 are measured and discussed.

Section 4.4 investigates the ORB-SLAM3 and OpenCV source code along with the dump of ORB-SLAM3 to aid in selecting inputs to the predictive model.

4.2 Performance measurements and profiling

Two of the most popular profiling types are sample-based and instrumentation. Sample-based profiling periodically records the call stack and has a very low overhead. Instrumentation, however, has a high overhead and measures the application at runtime, which provides detailed and accurate results. Linux Perf [76] and DynamoRIO [77] are examples of sample-based and instrumentation-based profiling types, respectively.

Linux Perf reads CPU performance counters that record hardware events such as instructions executed, cache misses and branch mispredictions. Each CPU has a different number of available counters as well as events that can be counted. The ARM Cortex A57 CPU has six available counters and 86 available events that can be counted according to section 11.8 of the Technical Reference Manual [59]. The Denver 2 CPU also has six available counters with 21 common performance events according to section 17.9.1 of the Parker Technical Reference Manual [58].

Linux Perf is a powerful profiling tool and can provide the user with a lot of information. Unfortunately, it can be overwhelming as Linux Perf is a command-line tool that is not always user-friendly. Luckily, there exist third-party applications such as ARM Map [56] that use

Linux Perf at its core to profile an application with ease. ARM MAP can expose performance problems and bottlenecks by measuring the computation over the time of an application, showing the thread activity, and providing access to the CPU Performance Measuring Unit (PMU) to count the performance events.

Profiling aims to identify bottlenecks in ORB-SLAM3 on the Nvidia Jetson TX2 and to determine program characteristics such as total instructions executed, instruction mix, and cache performance. The total instructions executed shows how many instructions were completed during the program execution. The total amount of instructions executed is directly linked to the execution time of a program as illustrated in (2.7.1).

Instructions are commonly divided into four groups: arithmetic, load, store, and branch. The combination of instruction types is called instruction mix and indicates what type of instructions are dominant in the program.

Cache performance indicates how many times the cache was accessed and what the cache miss ratio was. These program characteristics can be used to optimize the performance of a program.

The profiling of ORB-SLAM3 will be done on the ARM A57 CPU cluster as it provides more events that can be counted, which would lead to a more detailed analysis of how ORB-SLAM3 performs on the A57 CPU. The events that will be counted during the execution of ORB-SLAM3 are listed in Table 4. These events were chosen as the instruction mix, cache performance, and branch performance can be determined from these events.

Since the ARM A57 CPU has only six hardware counters available, the events will be measured in four different sets as listed in Table 5 during four separate executions of ORB-SLAM3. Each group is linked to a specific instruction type, such as arithmetic instructions, load and store instructions, cache performance, and branch performance for sets 1 through 4 listed in Table 5.

4.3 Profiling ORB-SLAM3

This study used ARM Map to profile ORB-SLAM3 for potential bottlenecks on the Jetson TX2. A bottleneck is the section of code where most time is spent and is also commonly referred to as a hotspot. The bottleneck of a program can be investigated to determine what can be done to increase the program's performance or, in this study's case, determine what inputs might be useful to create a prediction model. Figure 6 shows the output of ARM Map when ORB-SLAM3 v0.3 was profiled on the Nvidia Jetson TX2. Figure 6 shows the CPU activity and the number of performance events counted during the execution. The main thread activity shows that the CPU is constantly working and not waiting for any other input, indicating that ORB-SLAM3 might be compute-intensive on the Nvidia Jetson TX2. The events counted show that data processing and integer math instructions dominate

Table 4: **Jetson Tx2 PMU events**

Event name	Event description
CPU-cycles	Total cycles, beware CPU frequency scaling
instructions	Retired instructions, beware hardware interrupt counts
ASE_SPEC	Operation speculatively executed, Advanced SIMD instructions
DP_SPEC	Operations speculatively executed, integer data processing
VFP_SPEC	Operations speculatively executed, floating-point instruction
LD_SPEC	Operations speculatively executed, load
ST_SPEC	Operations speculatively executed, store
Cache-misses	Cache misses usually last level
L1D_CACHE_RD	L1D cache access, read
L1D_CACHE_WR	L1D cache access, write
L2D_CACHE_RD	L2D cache access, read
L2D_CACHE_WR	L2d cache access, write
MEM_ACCESS_RD	Data memory access, read
MEM_ACCESS_WR	Data memory access, write
Branch-misses	Mispredicted branch instructions
BR_IMMED_SPEC	Branch speculative executed, immediate branch
BR_INDIRECT_SPEC	Branch speculative executed, indirect branch
BR_RETURN_SPEC	Branch speculative executed, procedure return

Table 5: **The set of performance events that will be counted**

Set	Performance events
1	CPU-Cycles,Instructions,ASE_SPEC,DP_SPEC,VFP_SPEC
2	LD_SPEC, ST_SPEC, Cache-misses, MEM_ACCESS_RD, MEM_ACCESS_WR
3	L1D_CACHE_RD, L1_CACHE_WR, L2D_CACHE_RD, L2D_CACHE_WR, Cache-misses, Cache-references
4	Branch-misses, BR_IMMED_SPEC, BR_INDIRECT_SPEC, BR_RETURN_SPEC

ORB-SLAM3.

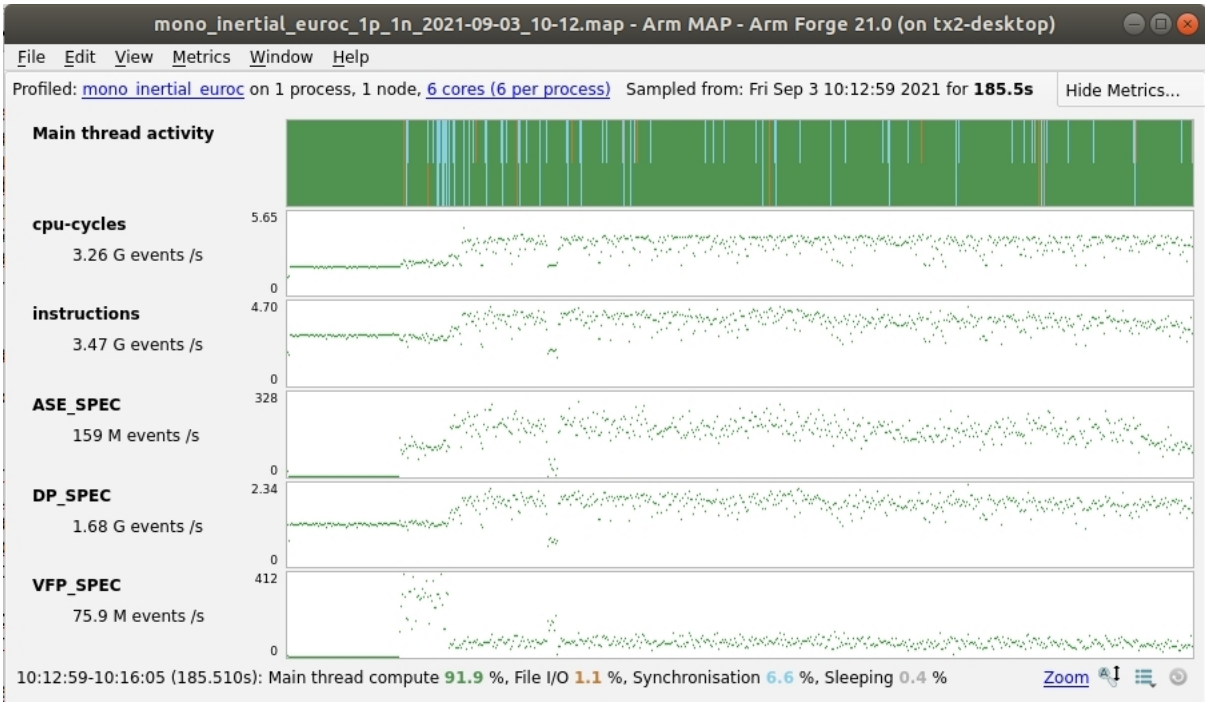


Figure 6: **Snippet of ARM Map output for profiling ORB-SLAM3 showing the main thread activity as well as the contents of the performance counters measured during the execution**

Figure 7 shows the function call stack that indicates that the OpenCV library function, FAST, is the function where most of the execution time is being spent. This is a clear indicator that the FAST function is a bottleneck of ORB-SLAM3 on the Jetson TX2, with 27.4 % of the time being spent on the FAST function with the closet function being OpenCV RowFilter with 8.2 %.

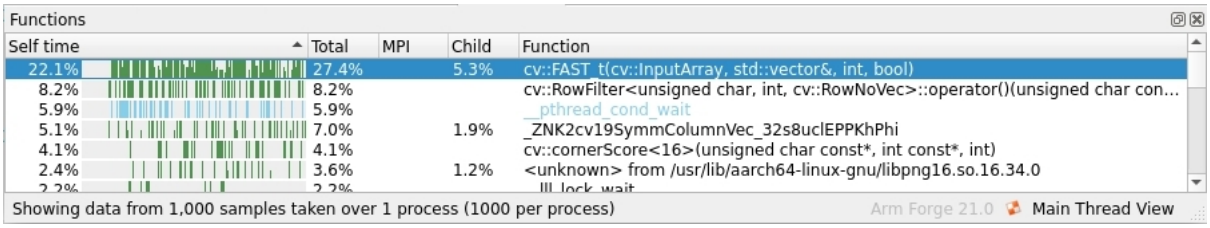


Figure 7: **Snippet of ARM Map output of the ORB-SLAM3 function call stack**

When looking at the function call stack shown in Figure 8, the FAST function is called within ORB-SLAM3's ComputeKeypointsOctTree function, which forms part of the

ORBextractorOperator that is part of the tracking thread of ORB-SLAM3. Knowing that the FAST function is the bottleneck and what procedures call it, an investigation into the source code of ORB-SLAM3 and OpenCV must be conducted to determine what parameters might be useful as input to the prediction model.

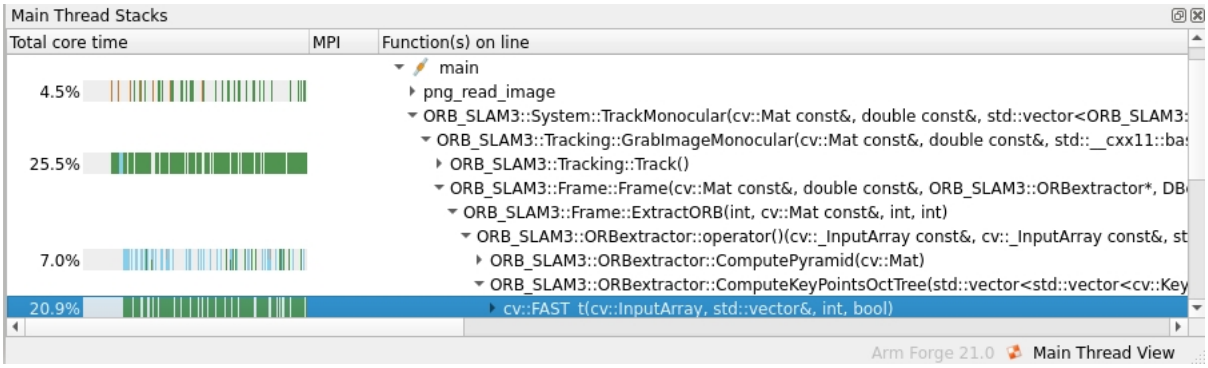


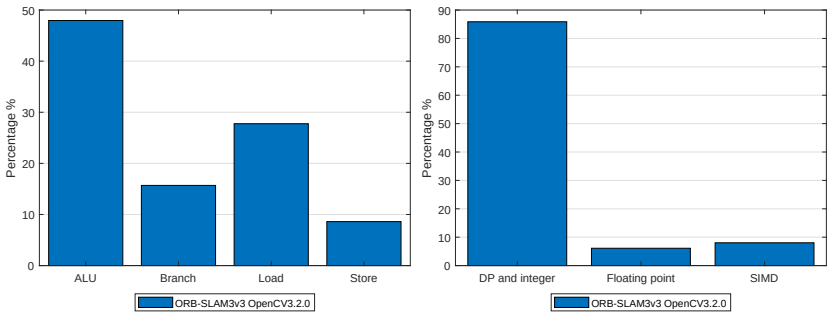
Figure 8: **Snippet of ARM Map output for profiling ORB-SLAM3 showing the function that most time is spent on**

Table 6 provides the number of instructions executed, how many times the cache was accessed, and the cache miss ratio. The number of instructions executed has a direct influence on the total execution time of ORB-SLAM3 according to (2.7.1). The number of instructions can also be used to determine the CPU's CPI during the execution of ORB-SLAM3, which is an indicator of how efficiently the CPU can execute ORB-SLAM3. The cache miss ratio forms part of cache performances, and from Table 6, it is clear that ORB-SLAM3 has an excellent cache performance on the Nvidia Jetson TX2 as it only has a miss rate of 1.47 %.

Table 6: **Intital ORB-SLAM3 profiling results**

Instructions executed	Cache access	Cache miss rates [%]
1.01E+12	3.77E+11	1.47

Figure 9a indicates that ORB-SLAM3 is dominated by ALU instructions, followed by load instructions. The high amount of load instructions is due to the sequence images loaded into the cache hierarchy to be processed by the CPU. Figure 9b breaks down the ALU instructions into data processing and integer math, floating-point, and SIMD instructions. From Figure 9b, it is clear that data processing and integer math instructions dominate ORB-SLAM3. The low value of SIMD instructions indicates that ORB-SLAM3 is poorly optimized to make use of SIMD instructions on the Nvidia Jetson TX2.



(a) **Instruction mix** *(b)* **ALU mix**

Figure 9: **Intital ORB-SLAM3 profiling results**

4.4 Investigating ORB-SLAM3 bottleneck

Looking into the results of the investigation, it has been confirmed that ORB-SLAM3 is CPU bound (Figure 6), the OpenCV function FAST is the bottleneck of ORB-SLAM3 on the Nvidia Jetson TX2 (Figure 7). The FAST function forms part of the tracking thread of ORB-SLAB3 and is called from the ComputeKeypointsOctTree function (Figure 8). ORB-SLAM3 has a good cache performance (Table 6) and is dominated by ALU instructions (Figure 9a). The ALU mix is dominated by Data Processing and integer instructions with a meagre amount of SIMD instructions (Figure 9b). This information guided the decision to investigate the source code of OpenCV to see how the FAST function is implemented. The ORB-SLAM3 source code will also be investigated to see how the FAST functions get called from the ComputeKeyPointsOctTree.

4.4.1 Investigating OpenCV source code

The OpenCV library source code should be examined to improve the efficiency and speed at which the FAST algorithm executes. The FAST algorithm of the OpenCV library is in the "fast.cpp" file under the "opencv/modules/features2d/src/"directory. Analysing the source code of OpenCV 3.2.0, it was identified that the FAST algorithm uses SIMD instructions for X86 CPU architectures, but there is no support for NEON SIMD instructions that are compatible with ARM CPU architectures.

Since there is no support for NEON instructions, the compiler is responsible for the parallelisation of the FAST algorithm for the ARM architecture. Thus, there is a possibility that the FAST algorithm executes sequentially on the Nvidia Jetson TX2. A performance increase should be gained by implementing the FAST algorithm using ARM's NEON instructions. Rewriting the FAST algorithm to use NEON instructions can be very challenging. Luckily, newer versions past OpenCV 3.2.0 have updated the FAST algorithm to

use NEON instructions on ARM architectures.

Looking at the OpenCV 4.1.1 source code it is immediately noticed on line 103 of the "fast.cpp" file under the "opencv/modules/features2d/src/" includes a new compile definition "CV_SIMD128" which is used to define that the compile and device support SSE or NEON instructions, whereas with OpenCV 3.2.0 there was only support for SSE with the "CV_SSE2" definition. ORB-SLAM3 was then re-profiled with OpenCV 4.1.1 to see how the performance is influenced.

Table 7 and 8 compares the performance differences with ORB-SLAM3 compiled OpenCV 3.2.0 and OpenCV 4.1.1. Table 7 shows that the number of instructions executed and the number of times the cache was accessed was reduced. However, the cache miss rate was slightly increased. Generally, a program will execute faster if fewer instructions are executed, and the performance is expected to increase further as there is less cache access. Thus, less time is being spent on waiting for data to be moved from the cache to the CPU. Table 7 shows that there is a decrease in load/store instructions and an increase in ALU instructions, with the branch instructions almost unchanged. The decrease in load/store instructions coincides with the decrease in cache access. Table 8 shows the breakdown of how the ALU instructions are divided according to Data processing Floating point and SIMD instructions. The percentage of floating point instructions is similar between OpenCV 3.2.0 and OpenCV 4.1.1. The largest impact that can be seen between the two OpenCV versions is that there is almost double the amount of SIMD instructions, coming from 8.02 % to 16.54 %. This clearly shows that SIMD is better utilised in OpenCV 4.1.1 compared to OpenCV 3.2.0.

Table 7: **Performance results of ORB-SLAM3 with OpenCV 3.2.0 and OpenCV 4.1.1**

Config	Instructions executed	Cache access	Cache miss rate [%]
OpenCV 3.2.0	1.01E+12	3.77E+11	1.47
OpenCV 4.1.1	9.01E+11	3.24E+11	1.80

Table 8: **Instruction mix of ORB-SLAM3 with OpenCV 3.2.0 and OpenCV 4.1.1**

Config	Load/Store [%]	ALU [%]	Branch [%]
OpenCV 3.2.0	36.36	47.95	15.69
OpenCV 4.1.1	34.47	50.02	15.51

Before receiving the Nvidia Jetson TX2, ORB-SLAM3 was tested on a desktop computer. While testing ORB-SLAM3 on the desktop computer, it became apparent that different build configurations of OpenCV also influenced the performance of ORB-SLAM3, specifically the parallel framework used to build OpenCV. OpenCV built with the OpenMP parallel framework executed 2.5 times slower than OpenCV built with the Intel TBB parallel framework on the desktop computer.

Table 9: **ALU mix of ORB-SLAM3 with OpenCV 3.2.0 and OpenCV 4.1.1**

Config	Data Processing [%]	Floating Point [%]	SIMD [%]
OpenCV 3.2.0	85.87	6.10	8.02
OpenCV 4.1.1	77.06	6.41	16.54

There is thus a need to investigate the performance of ORB-SLAM3 built with different versions and configurations of OpenCV. The performance differences between ORB-SLAM3 v0.3 beta and v0.4 beta also need to be investigated. It did not form part of the initial study, as the new version of ORB-SLAM3 was released during the study. It was decided to investigate ORB-SLAM3 v0.4 beta as the authors claim the performance of ORB-SLAM3 has been improved. According to the ORB-SLAM3, GitHub change catalogue ORB-SLAM3v4 has some minor bug fixes, has added compatibility to work with OpenCV 4.0, and allows the user to measure the running time of the system threads. The most significant change of particular interest is the change of OpenCV dynamic matrices to static matrices that speed up the code. This investigation can be found in Appendix C.

Table 10 compares the performance that ORB-SLAM3v3 has with OpenCV 4.1.1 and ORB-SLAM3v4 with OpenCV 4.1.1. The authors' changes caused yet another decrease in the total instructions executed and the number of times the cache was accessed. They were also able to further increase the percentage of SIMD instructions executed from 16.54 % to 20.09 %.

Table 10: **ORB-SLAM3v3 vs ORB-SLAM3v4**

Config	Instructions executed	Data Processing [%]	Floating Point [%]	SIMD [%]	Cache access
ORB-SLAM3v3	9.01E+11	77.06	6.41	16.54	3.24E+11
ORB-SLAM3v4	7.06E+11	72.50	7.40	20.09	2.40E+11

The process of profiling illustrated the difference in the execution of different versions of ORB-SLAM3 and OpenCV. It is noted that ORB-SLAM3 has good branch and cache performance and is dominated by ALU instructions. With the decrease in instructions executed and cache access, ORB-SLAM3v4 with OpenCV 4.1.1 should have a higher level of performance compared to the initial execution of ORB-SLAM3v3 and OpenCV3.2.0. Regardless of which ORB-SLAM3 and OpenCV version is used, the FAST function is still the bottleneck, thus, the source code of ORB-SLAM3 will be investigated to gain more knowledge on how the FAST function is called.

4.4.2 Investigating ORB-SLAM3 source code

Focusing on the section where the FAST function is being called, the source code of ORB-SLAM3 needs to be analysed, specifically, the ComputeKeyPointOctTree function that calls the FAST algorithm that was identified in Section 4.3.

The analysis shows that the FAST function is nested between three 'for' loops within the ComputeKeyPointOctTree function as illustrated in Listing 2. Listing 2 shows that the total number of times that the FAST function is called is controlled by three conditions: nlevels, nRows, and nCols. nLevels are linked to the number of pyramid levels that are used during ORB feature extraction. nRows and nCols are related to the image resolution.

From this, it is clear that the number of times that the FAST function needs to be called can be reduced by changing the number of levels of the image pyramid or by decreasing the image resolution. Lowering the image resolution should reduce FAST calls as changing the resolution changes the nRows and nCols conditions. Changing the resolution and pyramid levels might impact the accuracy of the ORB-SLAM3 algorithm.

```
1 ComputeKeyPointOctTree(nlevels,nRows,nCols)
2         for (levels=; levels <= nlevels; levels++)
3                 for (i = 0; i < nRows; i++)
4                         for (j = 0; j < nCols; j++)
5                                 FAST();
```

Listing 1: **Representation of how FAST function is executed within the ComputeKeyPointOctTree**

Further inspection of the ORB-SLAM3 source code showed that the feature extraction time could be improved by changing the limit of features to extract per frame. Limiting the features to extract per frame will cause an exit condition to be satisfied earlier. As with changing the resolution and pyramid levels, the accuracy and robustness can be influenced by changing the number of features extracted per frame.

It should be noted that changing the resolution, number of features, and number of pyramid levels will change the accuracy and robustness and will be dependent on the scenario and environment in which ORB-SLAM3 will be deployed. However, if the device that is running ORB-SLAM3 is low on resources, it might be useful to tune these variables to try and see whether better performance can be reached without losing too much accuracy and robustness. It will be useful to do a sensitivity analysis to see what variables ORB-SLAM3 is most sensitive to when changed. The sensitivity analysis is provided in Appendix D and E

4.5 Dumping FAST function

Profiling ORB-SLAM3 with ARM MAP showed that ORB-SLAM3 is bottlenecked by the single tracking thread, which is dominated by ALU instructions. Unfortunately, ARM MAP profiles the entire execution of ORB-SLAM3, and the model that will be created aims only to predict the average tracking time. Thus, a deeper investigation into the tracking thread must be done.

In order to have a better understanding of how the C/C++ human readable code executes on the physical hardware, the actual execution of the code needs to be dumped. One of the methods to achieve this is to use the GDB dump utility. Dumping the entire ORB-SLAM3 code will be unfeasible and will not help to make a more accurate prediction of the average tracking time of ORB-SLAM3 on embedded platforms, as the focus is to accurately predict the average tracking time of ORB-SLAM3. Dumping the entire code base of ORB-SLAM3 will be unnecessary as the other threads are unimportant in predicting the performance of ORB-SLAM3. Knowing that the tracking thread is the bottleneck and most of the execution time of ORB-SLAM3 is spent during the FAST algorithm of OpenCV during the ComputeOctTree function, just the FAST function of OpenCV will be dumped and investigated to help guide the choice of inputs to the predictive model.

In order to isolate the FAST function, ORB-SLAM3v4 and OpenCV 4.1.1 were compiled with debugging info. ORB-SLAM3 was then executed using a GDB debugger to set a breakpoint at the FAST function. Once the FAST function is executed, the GDB debugger is paused, and the FAST function code can be dumped. The dumped code is in the form of assembly language, as it is the direct code that will be executed on the device.

The dumped FAST code is then examined using a written MATLAB script to go through each line of code, checking the instruction and the registers used during the instruction. The goal of the script is to determine the instruction mix of the FAST function. The available registers on the ARM64 architecture are branch, float, general, NEON, special and other. Table 11 shows the percentage of registers used during the FAST function execution. It is clear that general instructions dominate the FAST function, followed by NEON and then branch. This shows a need to generalise or characterise the performance of an embedded platform in terms of the speed at which it can execute general and NEON instructions. One method that is generally used to characterise CPUs is in the form of artificial benchmarks. Using benchmarking data might be useful as inputs into the prediction model.

4.6 Conclusion

This chapter highlighted how ORB-SLAM3 and OpenCV are profiled with an investigation into their source code.

Table 11: **FAST function register usage percentage**

Register Type	Percentage [%]
general	63.71
NEON	19.93
branch	9.1
Other	4.8
float	1.6
Special	0.86

ARM Map eased the process of profiling ORB-SLAM3 on the Nvidia Jetson TX2, with easy access to the performance event counters. The output of ARM Map made it easy to identify the bottleneck of ORB-SLAM3, which is the FAST function from the OpenCV library when executing ORB-SLAM3.

The functions that call the FAST function were highlighted by analysing the function call stack. Using the profiling data, the source code of ORB-SLAM3 and OpenCV can be investigated to determine what inputs are useful for the prediction model creation.

The next chapter will describe the experimental design process that will be used to create a predictive model to estimate the performance that ORB-SLAM3 can achieve on embedded platforms.

Chapter 5 - Experimental Design

5.1 Introduction

The previous chapters explored the hardware characteristics of the embedded platforms available for this study and how software such as ORB-SLAM3 executes on them. This chapter will use the information gathered from the previous chapters to create an experimental design to aid in creating a prediction model to estimate the performance of ORB-SLAM3 on embedded platforms.

The first section will describe the selection process of inputs and outputs to the prediction model, followed by an explanation of how the necessary inputs and outputs will be generated to be used as a dataset. After that, the model creation procedure and how the study will verify and validate the models will be discussed.

5.2 Selecting prediction model inputs and outputs

Profiling ORB-SLAM3 in Chapter 4 clearly showed that ORB-SLAM3 is a CPU-bounded application when executed on an embedded platform, which means that the model inputs must focus on the CPU characteristics of an embedded platform. Perhaps the most commonly known CPU characteristic is the clock frequency of a CPU. A general notion of CPU clock frequency is that the higher the CPU clock frequency is, the higher performance the CPU can deliver. This is true. However, this is a one-dimensional performance measure, as the highest advertised clock frequency is usually only achievable for short periods and is applicable for the execution of simplistic applications. The profiling of ORB-SLAM3 supports this statement as it was shown that the FAST algorithm is the bottleneck of ORB-SLAM3 and contains 19.93% NEON instructions, which generally have more complex execution and have a higher latency than general instructions such as mov, branch, and add. The performance that a CPU can achieve during different workloads is dictated by the CPU micro-architecture accompanied by the CPU clock frequency. The literature study showed that CPU micro-architectures could be very complex, and they are continuously updated and changed with releases of new CPUs, thus it would be impractical to try and create a model based on some architecture features as they can be removed or replaced in the future.

Commonly, CPU manufacturers provide some basic CPU information, such as clock speed, cache hierarchy, number of CPU cores and what IPC it can achieve. These stats are usually important when choosing CPUs, but there are no accurate representations of the CPU capabilities in real-life situations. Therefore, benchmarking programs have been created to test a CPU's performance and give the user a more quantitative performance value. However, some benchmark programs are purely synthetic, which means they are designed to provide the best theoretical performance of the CPU without a realistic program being executed. Thus,

programs such as PassMark [78], GeekBench [79], CineBench [80], and 3DMark [81] have been created, which are collections of benchmarks to provide an excellent overall CPU performance across a set of applications that accounts for micro-architectural differences in CPU designs.

PassMark is benchmarking software that can measure the performance of computer hardware components such as CPUs, GPUs, hard drives and RAM. PassMark provides a score for different elements of the tested computer hardware and has a free and paid version. PassMark also has a database where individuals can post their scores for their computer hardware [78]. Thus, a User can search the PassMark database and see what performance a particular CPU can achieve [82]. This can be beneficial for the predictive model as a user can search for a CPU on PassMark, enter the scores into the prediction model, and see what kind of performance they can expect to achieve when executing ORB-SLAM3. Due to the free use of PassMark and their database, along with the way PassMark breaks down different aspects of CPU performance PassMark was used in this study.

Table 12 shows the categories that are tested by the PassMark benchmark. The CPU mark is an overall score that combines all the other categories. The rest of the categories are individual benchmark test scores where specific programs are executed to test the CPU performance in a specific category. All tests are done with multithreading except for the CPU single-threaded test. These benchmark scores will be investigated and reduced to be used as inputs to the prediction model.

Table 12: **PassMark benchmark elements**

PassMark benchmark elements
CPU Mark
Integer Math
Floating Point Math
Prime Numbers
Sorting
Encryption
Compression
CPU Single Threaded
Physics
Extended Instructions (NEON)

PassMark will thus be used to characterise the performance of the CPUs of the embedded platforms to be used as inputs to the prediction model.

Chapter 4 showed that the FAST algorithm forms part of the tracking thread and is the bottleneck of ORB-SLAM3. Thus, the maximum performance that ORB-SLAM3 can achieve on an embedded platform is determined by the speed at which the platform can execute the tracking thread. Thankfully, the authors of ORB-SLAM3 already measure the amount of

time that ORB-SLAM3 spends in the tracking thread and calculate the average time spent on tracking per frame. Thus, the prediction model will predict the average tracking time that ORB-SLAM3 can achieve on a given platform.

5.3 Generation dataset for model creation

With the inputs and output defined, the data must be gathered to train and test the model. The model output will be generated by executing the 11 EuRoC MAV sequences mentioned in Section 3.3.1 on the embedded platforms. The study has access to three embedded platforms with a total of four different CPUs, the Nvidia Jetson TX2 (Denver 2 and A57 CPUs), the Raspberry Pi 3B+ (A53) and the Raspberry Pi 4B (A72), all the hardware details are provided in Section 2.6. If all 11 sequences are executed on the four different CPUs, the dataset will only contain 44 data pairs, which can limit the prediction model's performance. Thus, the number of available CPUs will artificially be increased by setting up different CPU configurations to generate enough data pairs for the model creation.

Generating the different CPU configurations will be achieved by over-clocking and under-clocking the different CPU cores and limiting the amount available CPU cores. The embedded platforms CPUs can easily be over and under-clocked by altering their respective configuration files. Chapter 3 described that TaskSet could be used to set the CPU affinity to specific CPUs cores for execution. This will force the program to only use the set CPU cores during execution.

The A57 and A72 CPUs will be tested using four and two cores with a CPU frequency from 1 GHz to 2 GHz with 200 MHz intervals. The Denver CPU has only two cores, so it will only be tested with two cores, from 1 GHz to 2 GHz with 200 MHz intervals. The A53 CPU will be tested with four cores but only at frequencies of 1.3 GHz, 1.2 GHz and 1.0 GHz. The A53 processor is tested at these different intervals as 1.3 GHz is the maximum clock frequency of the Raspberry Pi 3B+. The last configuration is not setting any CPU affinity on the Nvidia Jetson TX2 to allow the scheduler to determine what processes should be pinned to what CPU, so all six CPU cores are available for execution and will also be tested from 1 GHz to 2 GHz with 200 MHz intervals. Table 13 shows the 39 CPU configurations that will be used to generate the prediction model dataset.

So by executing the 11 EuRoC MAV sequences on the 39 available CPU configurations, the dataset will have 429 data pairs. All of the fields for each of the data pairs in the dataset are displayed in Table 14. It combines the CPU name, clock frequency, PassMark benchmarking results, the executed sequence, and the average tracking time that the specific CPU configuration achieved while executing ORB-SLAM3. Thus, the entire dataset will be of size 429 x 15, with a possibility of 14 inputs and a single output.

The next section will explain the process of how the dataset is populated.

Table 13: List of CPU configurations to be used to generate dataset

CPU	Number of cores	Core frequencies (GHz)	Number of CPU configurations
A57	2, 4	1.0, 1.2, 1.4, 1.6, 1.8, 2.0	12
A72	2, 4	1.0, 1.2, 1.4, 1.6, 1.8, 2.0	12
A53	4	1.0, 1.2, 1.3	3
Denver 2	2	1.0, 1.2, 1.4, 1.6, 1.8, 2.0	6
TX2	6	1.0, 1.2, 1.4, 1.6, 1.8, 2.0	6

Table 14: Data pairs fields

Data fields	Explanation
CPU	CPU Name
Cores	Amount of CPU cores
Frequency	CPU clock frequency
Mark	PassMark overall CPU score
Int	PassMark integer score
Float	PassMark float score
Prime	PassmMark prime score
Sort	PassMark sort score
Encryp	PassMark encryption score
Comp	PassMark compression score
Single	PassMark singel CPU performance score
Phys	PassMark physics score (GPU)
NEON	PassMark SIMD score (NEON)
Sequence	EuRoC MAV sequence
Tracking	ORB-SLAM3 average tracking time

5.4 Data collection and pre-processing

The data collection process will happen in two separate steps: benchmarking the CPU configuration and executing ORB-SLAM3 with the 11 EuRoC MAV sequences. As stated in Section 5.3, the CPU clock frequency is modified by changing the embedded platform's CPU configuration file. Once the CPU frequency is set, the PassMark benchmark is executed to characterize the performance of the CPU configuration. Afterwards, the 11 EuRoC MAV sequences are executed on the platform using the created Docker container mentioned in Section 3.5. The results of the PassMark benchmarking and ORB-SLAM3 execution are combined in a table format shown in Table 14. Figure 10 shows the procedure that will be followed to generate the necessary dataset.

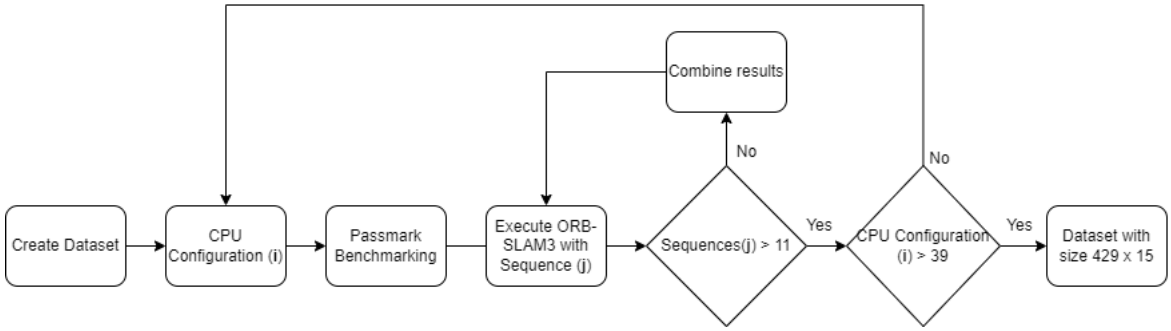


Figure 10: **Data gathering procedure**

Out of the 15 fields in the dataset, 14 are possible inputs, and the last field is the output or target of the model, the average tracking time that ORB-SLAM3 achieved for the different EuRoC MAV sequences on the different CPU configurations. It is important to note that the 14 fields are immediately reduced to 13, as it was decided to always include the EuRoC MAV sequence as an input to ensure that the model will have 429 unique data pairs. If the EuRoC MAV sequence had been left out, there would have been only 39 unique entries in the dataset. To further reduce the complexity of the prediction model, two feature selection methods will be used: the correlation between the target value and the remaining inputs and a user-selection method guided by profiling done in the previous chapters.

5.5 Experimental Design: Model Creation

After the dataset has been generated, the prediction models can be created. The study will investigate the creation of different prediction models to see what model can achieve the best performance. With the dataset created, feature selection will be applied to it to select the inputs that have a performance impact on the average tracking time on embedded platforms. The feature selection process will be guided by calculating the correlation between the target

and all the inputs and by the knowledge gained through the literature study and the profiling of ORB-SLAM3. After the feature selection process, the dataset will be standardized using Python's StandardScalar module to ensure that the dataset has zero mean and unit variance.

The study will investigate two different experiments. Experiment 1 will serve as verification and Experiment 2 as validation. The only difference between the two experiments is the data pairs used to train and test the models. Experiment 1 will use the entire dataset with a ratio of 75:25 split ratio for training and testing. Experiment 2 will be trained on only a subset of the dataset to see how the models perform on an unseen embedded platform. The subset of training data will include the data pairs of the A72, A53, and Denver 2 CPUs, with the test data being the data from the A57 CPU. The data from the TX2 CPU is removed as it combines the A72 and A57 CPUs performances. This dataset has 363 data pairs, from which 231 pairs are used for training and 132 for testing, effectively creating a training test set with a 63:37 split ratio. If the model can achieve adequate performance for an unseen embedded platform, the model has been validated.

Three modelling techniques will be used: a simple Linear Regression model, an Extra Trees Regressor, and a Multi-Layer Perceptron model, MLP. The last step in the model creation process is to compare the performances of the different models. Figure 11 shows the steps of the experimental design.

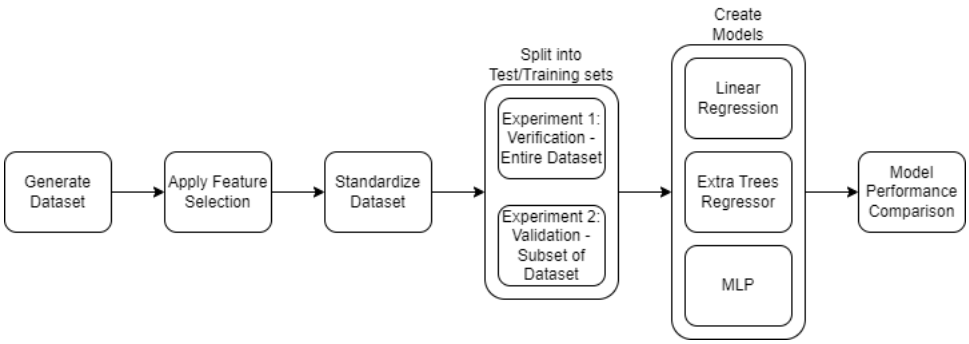


Figure 11: **Experimental Design method**

The performance metrics that will be measured are the Mean Absolute Error, MAE, Root Mean Square Error, RMSE, and the Coefficient of Determination, R^2. The MAE will be calculated using equation 5.5.1, where y is the predicted value and $\hat{y}$ is the actual value, and N is the size of the test set. The RMSE is calculated using equation 5.5.2, where y, $\hat{y}$, and N is the same as used in the MAE. The R^2 score will be calculated using equation 5.5.3.

$$\text{MAE}(y, \hat{y}) = \frac{\sum_{i=0}^{N-1} |y_i - \hat{y}_i|}{N} \tag{5.5.1}$$

$$\text{RMSE}(y, \hat{y}) = \sqrt{\frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N}} \tag{5.5.2}$$

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y}_i)^2}, \; where \; \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i \tag{5.5.3}$$

Table 15 shows the performance metrics criteria that the models should achieve for verification and validation.

Table 15: **Performance metrics for prediction models**

MAE %	**RMSE %**	R^2
less than 10 %	less than 10 %	greater than 0.9

Table 16 summarizes the two experiments with their names and their performance criteria.

Table 16: **Summary of experiments**

Experiment	**Name**	**Dataset**	**MAE %**	**RMSE %**	R^2
1	Verification	Entire Dataset	< 10 %	< 10 %	> 0.9
2	Validation	Subset	< 10 %	< 10 %	> 0.9

5.6 Conclusion

The Chapter described the experimental design method, specifically looking at how the dataset is generated and how the models are created. Two experiments will be done. For experiment 1, verification, the entire generated dataset is used to create the model. For Experiment 2, validation, an embedded platform is removed from the dataset to see how well the model performs on data it has not seen before.

By using the proposed experimental design method, the next Chapter will focus on creating the prediction model to estimate the performance that ORB-SLAM3 can achieve on embedded platforms.

Chapter 6 - Modelling

6.1 Introduction

Chapter 5 described the experimental procedure that will be followed to create the prediction model. This Chapter will use the proposed experimental design to create a model to estimate the performance of ORB-SLAM3 on embedded platforms.

As explained in the previous Chapter, the model-creating process comprises a couple of phases. The first phase is to generate the dataset that will be used for training and testing the prediction model. Afterwards, the feature selection method is applied, and the dataset is split into training and test sets. The data is then used to create the prediction models. For verification, the entire dataset is used for training and testing, whereas for validation, a CPU is removed from the dataset to see how the models perform for unseen CPUs.

6.2 Benchmarking and executing ORB-SLAM3

The first step in the data gathering process is to benchmark the 39 CPU configurations mentioned in Section 5.3 Table 13. PassMark's Performance Test application is downloaded and extracted from the PassMark website [78], whereafter, it is executed on the different CPU configurations. The benchmarking results of the different CPU configurations are provided in Appendix B and are stored in an Excel file on a host computer. After running the PassMark benchmark, ORB-SLAM3 is executed with the 11 EuRoC MAV sequences using the created Docker image of Chapter 3. When the Docker image of ORB-SLAM3 was created, an automated script was also added to the Docker image to run ORB-SLAM3 on the 11 EuRoC MAV dataset sequences automatically. As mentioned earlier, ORB-SLAM3 saves the performance results of each execution time, saving essential stats such as the average tracking time in a text file. All these files were saved and moved to the host laptop for further processing. The results of executing ORB-SLAM3 on the different CPU configuration are not provided as an Appendix as it is too large. However, it is provided with the Dissertation files as the complete dataset.

A Python script compiles a dataset that combines the generated data into a table saved as a CSV. The CSV file contains the CPU name, number of cores, CPU frequency, the PassMark benchmark results, the EuRoC MAV sequence, and the average time for the tracking thread, which is the target. This CSV file contains a table of size 429 x 15, as explained in Chapter 5 and will be used to create the models.

6.3 Experiment 1: Verification

With the dataset created, the feature selection method must be applied to the data to determine what data fields will be used as input for the prediction models. Firstly, the correlation between all the inputs of the dataset is correlated to the target of the model, which is the average tracking time that the embedded platforms achieved during the execution of ORB-SLAM3. The inputs with the highest correlation to the target will be selected as inputs to the model. Table 17 shows the top five correlation scores of the different inputs to the average tracking time. All returned scores show a negative correlation between the input scores and the target value. This is expected, for example, if the CPU frequency is increased, there should be a decrease in the average execution time of ORB-SLAM3. The same is true for all the scores shown in the table. If the scores are higher, the CPU should have better performance, thus decreasing execution time. The higher the correlation coefficient, the stronger the relationship between input and target. As expected, the most significant correlation coefficient belongs to the CPU frequency, as higher clock frequencies usually mean applications can execute at faster speeds. The second most significant contributor is the Single Threaded CPU score of PassMark, which also makes sense since the tracking thread is limited to a single CPU. The fact that the correlation coefficients show that the Float score has a more significant influence on the target value, average tracking time, is strange as profiling showed that the FAST function only contains 1.6% Float instructions and 19.93% NEON instructions according to Table 11 in Section 4.5.

Table 17: **Experiment 1: Correlation coefficient for dataset inputs against the average tracking time**

Parameter	Correlation value
CPU Frequency	-0.73
Single Thread	-0.71
Float score	-0.64
NEON	-0.57
Sorting	-0.56

The appropriate inputs for the prediction models will be determined by combining the results of the correlation coefficients and the knowledge gained during Chapter 4, which involved profiling ORB-SLAM3. From the PassMark category results, the following inputs have been selected: Single CPU score, NEON Score, and integer score. The Single CPU score has been chosen because the tracking thread executes on a single core, and it exhibits the second-highest correlation coefficient with our target variable. It's important to note that a CPU with more than three cores should be considered the minimum requirement for running ORB-SLAM3 since it employs three threads. If there are fewer than three cores, the CPU time will be divided among these threads. The NEON score has also been included in the inputs because

the FAST algorithm executes using NEON instructions, constituting 19.93% of its instructions. The correlation coefficient indicates a strong relationship between the NEON score and our target variable. The integer score is the final input chosen because it represents the largest portion of executed instructions during the FAST algorithm, accounting for 63.71%. As a reminder, the sequence of EuRoC MAV is also included as part of the inputs, as discussed in the previous chapter. Table 18 provides a summary of the inputs that will be used to create the models.

Table 18: **Experiment 1: Inputs selected to create the prediction models**

Input
CPU Frequency
CPU Single Threaed
Extended Instruction (NEON)
Integer Math
EuRoC MAV Sequence

Python was used to create the prediction models. The data is first standardized using Python's StandardScalar module to ensure that the dataset has zero mean and unit variance. The dataset is also split into a training and test set with a ratio of 75:25 to be used for the models' verification.

Different models will be created to see what model can provide the best results. Modelling techniques such as linear regression, ExtraTrees and a simple MLP regressor will be used to create the models. All the models are used from the Scikit-learn library, including LinearRegression from the linear model module [83], ExtraTreesRegressor from the ensemble module [84], and the MLPRegressor from the neural network module [85].

The Linear regression model is an Ordinary least squares linear regression model with all the default Python settings. This model has been chosen for its simplicity. The ExtraTrees ensemble method was chosen as ensemble methods can provide better performance as they are typically made up of more than one model, and the models learn from each other. ExtraTrees are also known for their robustness [86], the default settings were also used. The MLPRegressor neural network was chosen as it has the ability to learn non-linear relationships in the dataset, the hidden layer size was increased from 100 to 200, and the max iterations to 20000 to allow for convergence.

This study will look at the following performance characteristics of the models. The coefficient of determination or R^2 of the model using the model.score() method, the Mean Absolute Error MAE of the predictions as well, and the Root Mean Square Error RMSE. The MAE and RMSE will also be discussed as a percentage value.

Table 19 shows the performance of the prediction models. The Extra Trees model is the best-performing model with an MAE and RMSE percentage of 2.84% and 3.93% and a coefficient of

determination score 0.95. The low MAE and RMSE percentage indicates that the model can make accurate predictions with a small relative error. Both the MAE and RMSE percentages were calculated by dividing their respective value by the range of the target training data multiplied by 100. The high coefficient of determination score shows that input variables are good predictors for the target variable. The Linear Regression model has the worst performance, and the MLP regressor has a comparable performance to the Extra Trees model. Both the Extra Trees and MLP models were able to achieve the performance criteria listed in Table 15, with both of them having an MAE and RMSE percentage well below 10 %. Their R^2 scores are above the required value of 0.9. Thus, the Extra Trees and MLP models are verified to achieve the performance that was defined for the study.

Table 19: **Experiment 1: Prediction models results**

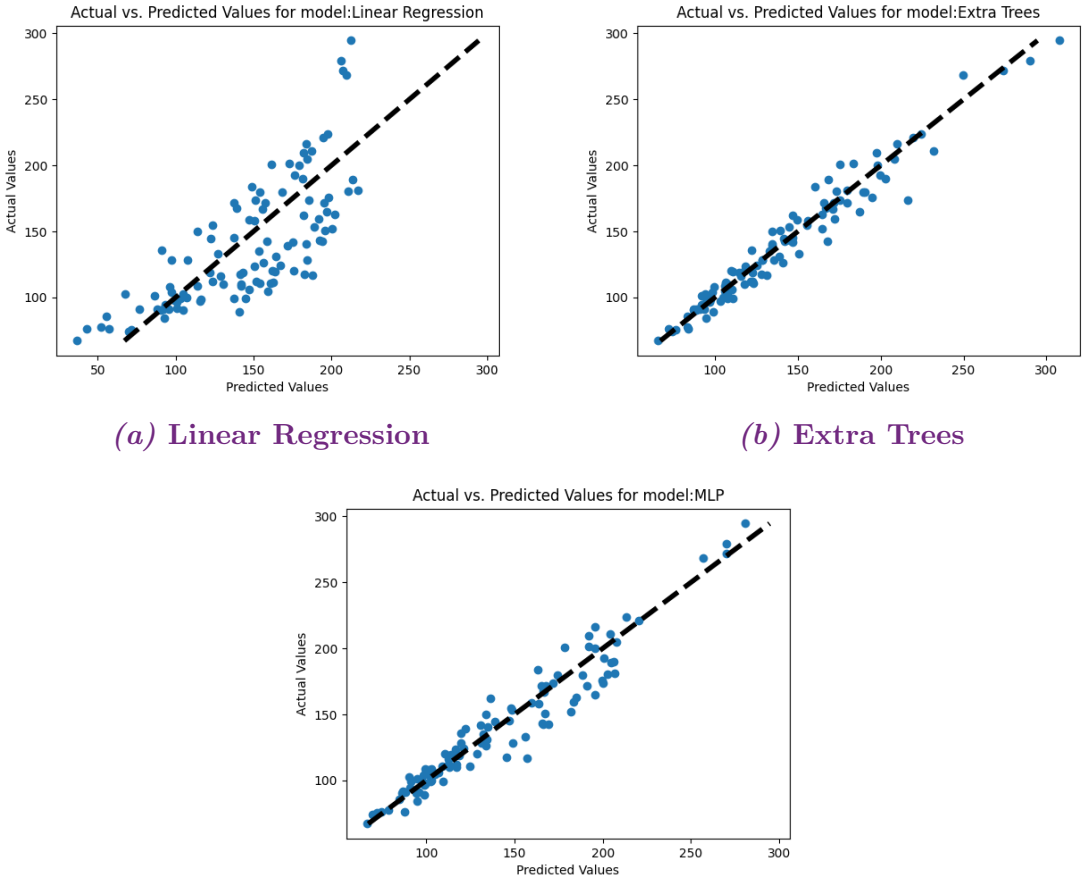
Modelling technique	MAE (ms)	MAE %	RMSE (ms)	RMSE %	R^2
Linear Regression	27.48	10.62	32.76	12.66	0.51
Extra Trees	7.34	2.84	10.15	3.93	0.95
MLP	9.49	3.67	12.86	4.97	0.93

Figures 12a to 12c show scatter plots for the three prediction models plotting the actual vs. the predicted values. Figure 12a shows that the Linear Regression model is loosely clustered around the diagonal line and cannot make the most accurate predictions. It should also be noted that it struggles to predict targets with a longer average tracking time accurately. This might be due to the low frequency of these occurrences as the A53 CPU was the CPU with the lowest performance and had the least data pairs in the dataset. Figures 12b and 12c are more tightly clustered around the diagonal line, showing their strength in accuracy. The Extra Trees, Figure 12b, and MLP models, Figure 12c, achieve a better performance predicting the longer average tracking time of the A53 CPU.

The performance that the Extra Trees model achieves verifies the model, and it shows the robustness and performance capabilities of Extra Trees models. However, the models are still trained and tested with all of the embedded platforms. In the next section, the performance of the models will be investigated to predict the performance of an embedded platform that is not present in the dataset and will be used to validate the model.

6.4 Experiment 2: Validation

This section will validate the models by predicting the average tracking time of ORB-SLAM3 for an embedded platform that is not present in the training set. To achieve this, the dataset has been reduced to only include the data gathered from the Denver 2 CPU, A72 CPU and the A53 CPU. The unknown CPU that will be used for testing is the A57 CPU. The TX2



(a) **Linear Regression**

(b) **Extra Trees**

(c) **MLP**

Figure 12: **Experiment 1: Scatter plots for the three prediction models showing the Actual vs. the predicted values**

CPU data was removed as it is a combination of the A57 and Denver 2 CPU and might skew the results.

The correlation coefficient was also recalculated, and the results are provided in Table 20. The same top five inputs are present in this training set as with the entire dataset. The order is also the same, but the values have changed. Since the correlation table did not change, the same inputs will be used to create the validation models as used for verification.

The same three modelling techniques were used, and the performance of the models is displayed in Table **??**. The best-performing model is still the Extra Trees model, with the lowest MAE and RMSE percentages of 9.12 % and 13.14 %, respectively and a coefficient of determination score of 0.78. Table **??** shows that the Linear regression model now performs better than the MLP model. Neither the Linear Regression nor MLP models achieved the performance

Table 20: **Experiment 2: Correlation coefficient for dataset inputs against the average tracking time**

Parameter	Correlation value
CPU Frequency	-0.72
Single Thread	-0.66
Float score	-0.56
NEON	-0.48
Sorting	-0.46

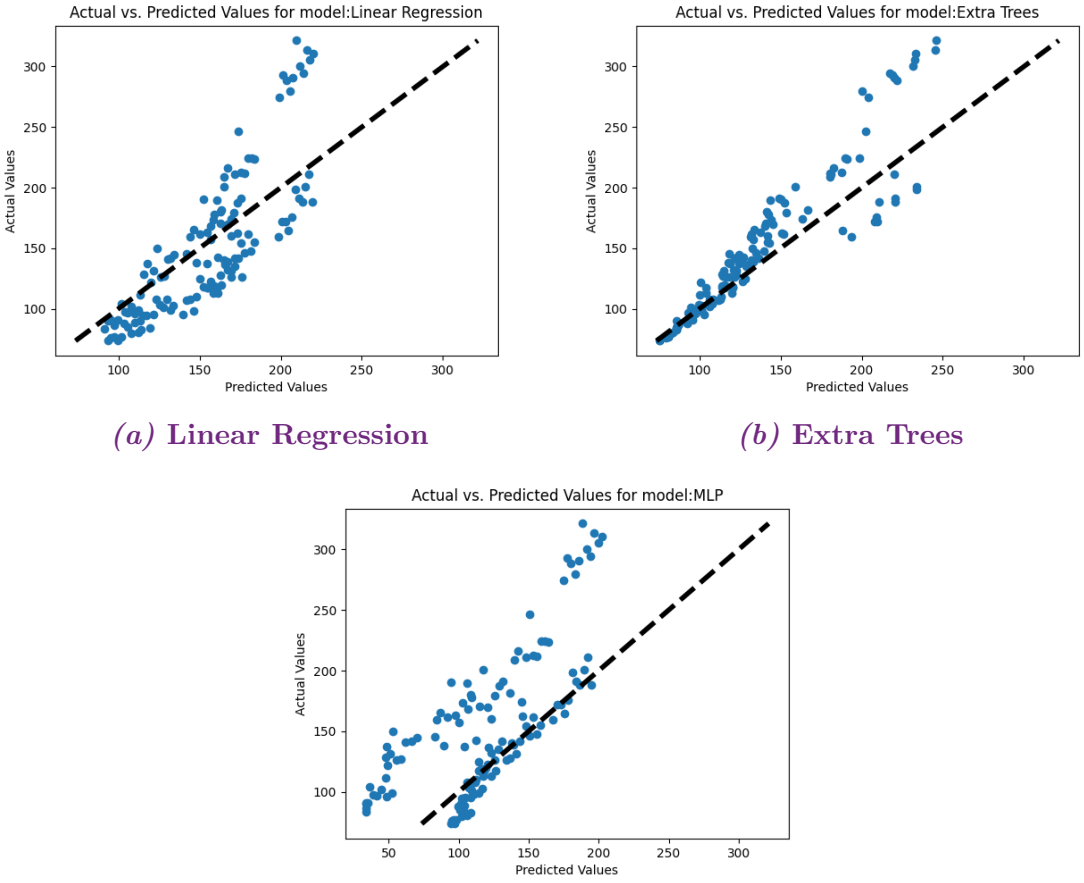
criteria listed in Table **??**. However, the Extra Trees Model is the only model that was able to achieve a MAE % of below 10 %, and it only fulfils the MAE criteria and not the RMSE or R^2 criteria. The RMSE criteria is only 3.14 % over the specified value. The R^2 score of 0.78 is well below the required value of 0.9. This means there is room for improvement in the models. One approach that can be taken is to try and fine-tune the models to achieve better performance For the Extra Trees Model parameters such as the number of trees in the forest, the maximum depth of a tree, the number of features to consider when looking for a split, can be altered to try and fine-tune the model. For the MLP parameters such as hidden layer size, activation, and solver, can be used to fine-tune the MLP model. Another approach will be to increase the training and test set size by profiling more embedded platforms.

Table 21: **Experiment 2: Prediction model results**

Modelling technique	MAE (ms)	MAE %	RMSE (ms)	RMSE %	R^2
Linear Regression	29.39	13.72	36.89	17.22	0.61
Extra Trees	19.55	9.12	28.16	13.14	0.78
MLP	40.25	18.79	53.40	24.92	0.19

Figures 13a to 13c show the scatter plots for the three prediction models plotting the actual vs. the predicted values for experiment 2. Figure 13a, Linear Regression Model, shows poor performance in predicting the correct values. As with Experiment 1, it had the most challenging time predicting longer average tracking times. The Extra Trees Model, Figure 13b, performed best. It can very accurately predict the performance of the average tracking time if below 150ms. Afterwards, the error between the predicted and actual values increases. This could mean that the model is over-fitted to that portion of the dataset as there is a larger frequency of results in that range. Figure 12c shows that the MLP model could accurately predict only a portion of the dataset, with the rest of the predictions forming another distinct pattern. A further investigation into the training data needs to be done to determine why the MLP model reacted this way.

Changing the training data influenced the model's results. A contributor to the decrease in performance can be caused by the decrease in available data from 321 down to 231 to train the



(a) **Linear Regression**

(b) **Extra Trees**

(c) **MLP**

Figure 13: **Experiment 2: Scatter plots for the three prediction models showing the Actual vs. the predicted values**

model with. The MLP had a drastic performance decrease, which the training data decrease could have caused. On the other hand, the Linear Regression had a similar performance to that of the entire dataset. Extra Trees is still the best-performing model. Experiment 2 did validate the possibility of creating a model that can predict the performance of ORB-SLAM3 on embedded platforms. However, the models were not able to achieve the required performance criteria, with the Extra Trees model missing the RMSE criteria with 3.14 % and the R^2 score with 0.12.

6.5 Conclusion

This Chapter showed how the data for the prediction model is collected and how it is used in creating several prediction models. Experiment 1 verified the models by using all the data pairs in the dataset and achieving the defined performance criteria. Whereafter, Experiment 2 validated the models by removing the data pairs of a particular CPU from the dataset to see how well the model can predict the performance of ORB-SLAM3 on an unknown platform. Unfortunately, Experiment 2's models did not reach the performance criteria.

The best-performing model was the Extra Trees model for both the Verification and Validation experiments. This showed that the profiling of ORB-SLAM3 was a helpful method in determining what inputs to use for creating the model.

Table 22 summarizes the Experiments results and compares the results of the best-performing Extra Trees modelling technique. From this, it is clear that Experiment 1 was verified by meeting the defined performance criteria. Experiment 2 validated the approach, although not all the performance criteria were met.

Table 22: **Performance comparison between verification and validation Extra Trees modelling technique**

Experiment	MAE %	RMSE %	($R\hat{2}$)
Experiment 1: Verification	2.84	3.93	0.95
Experiment 2: Validation	9.12	13.14	0.78

This Chapter showed that the model is feasible in estimating the performance of ORB-SLAM3 on embedded platforms. The model can achieve better results if more embedded platforms are added to the training set. The training set was also limited to only the EuRoC MAV dataset. Future work can expand on adding more datasets and creating a more complex model that accounts for the environment in which ORB-SLAM3 will be executed.

Chapter 7 - Conclusion and recommendations

7.1 Introduction

This chapter discusses the findings of this study according to the objectives and methodology provided in Chapter 1.

The first section briefly overviews the study's objectives and discusses whether they were met. The second section highlights some problems and challenges experienced during this study and provides recommendations to mitigate the problems and challenges for future studies. The chapter concludes with the final remarks on this study.

7.2 Research objectives reflection

The research aim of this study was to create a prediction model that can predict the performance of ORB-SLAM3 on embedded platforms. The study's objectives were outlined in Chapter 1, and whether the objectives were met is discussed in this section.

Software and hardware implementation: For this study 3 embedded platforms were available: the Nvidia Jetson TX2 (2 onboard CPUs the Denver 2 and Cortex A57), the Raspberry Pi 3B+ (Cortex A53 CPU), and the Raspberry Pi 4B (Cortex A72 CPU). ORB-SLAM3 was identified as the VISLAM to be implemented on the embedded platforms, and the EuRoC MAV dataset was used for evaluating the ORB-SLAM3 algorithm. Using the ORB-SLAM3 documentation, the development environment was set up and built on the Nvidia Jetson TX2, from which a Docker container was created to be implemented on all the embedded platforms, as discussed in Chapter 3.

Performance profiling: In order to create an accurate prediction model, ORB-SLAM3 was profiled to identify the bottleneck of the algorithm and what possible parameters can be used to create the prediction model. ORB-SLAM3 was profiled on the Nvidia Jetson TX2 using ARM MAP due to licensing constraints, and it was identified that the FAST function of the OpenCV library is the bottleneck, as shown in Chapter 4. ORB-SLAM3 is also a CPU-bound application and has a good cache performance. Dumping the execution of ORB-SLAM3, it became clear that when executing the FAST algorithm, it is dominated by general instructions and NEON instructions, concluding that a CPU with a strong capability of executing NEON instructions will be beneficial to execute ORB-SLAM3.

Experimental design: An experimental design was created to generate the necessary dataset to create a prediction model to estimate the performance of ORB-SLAM3 on embedded platforms. The process discussed how the embedded platforms are virtually expanded by creating artificial CPU configurations by varying the CPU cores and frequency to get 39 CPU configurations. The 11 EuRoC MAV sequences was used during the ORB-SLAM3 execution

to create a dataset that has 429 entries. The entire dataset consisted of 429×15 elements, with the 15th element the target of the models, which is the average tracking that ORB-SLAM3 achieved on the embedded platforms. The other 14 data fields are the PassMark result scores, the CPU frequency and the executed data sequence. The feature input selection is described, followed by the process of creating the respective models.

Modelling: PassMark was used as a benchmarking application to try and characterize the CPUs of the different embedded platforms. The Docker container created in Chapter 3 was used to execute ORB-SLAM3 on all the embedded platforms to gather training data. The results from executing ORB-SLAM3 on the embedded platforms and the results from executing PassMark were combined into a new dataset that was used to create the prediction models, as described in Chapter 5. Following the dataset's creation, three different models were created to see which model could achieve the best performance in predicting the performance of ORB-SLAM3 on embedded platforms. To reduce the complexity of the models, feature selection was applied to the inputs. The three created prediction models are the Linear Regression, Extra Trees, and MLP models. The Extra Tree model had the best performance over the entire dataset and was verified as it met the performance criteria. For validation, a CPU was removed from the dataset to see how the model performs when predicting the performance of a CPU that is not present in the dataset. The best model was the Extra Trees model, with a slight degradation in performance. Although not all the performance criteria were met, the experiment was still validated as it missed the performance criteria by a small margin.

7.3 Recommendations

The most significant limitation of this study is the need for more embedded platforms to generate more training data, along with the use of only the EuRoC MAV dataset. The study can thus be expanded by adding more embedded platforms to the study and by adding more sequences from the TUM dataset with Visual Inertial sequences needed to ORB-SLAM3 in the same configuration as used in this study. It should also be noted that the model was trained on CPU architectures of the ARMv8a architecture, so the model will perform poorly on other architectures.

The FAST function is the bottleneck when executing ORB-SLAM3 on the embedded platforms. This shows that there is still an opportunity to increase the performance of ORB-SLAM3 on the embedded platforms. OpenCV does provide the opportunity to increase the speed of ORB-SLAM3 as it does have a GPU implementation of the FAST function. This means ORB-SLAM3 can achieve even greater performance on embedded platforms with a dedicated GPU if ORB-SLAM3 was rewritten to use the GPU for the FAST algorithm instead of the CPU. This, however, does come with its challenges, as the whole ORB-SLAM3 library might need changing. There might also be a limiting factor when transferring the data between CPU and GPU that can mitigate the extra performance gained by implementing the

FAST algorithm on the GPU.

Although ORB-SLAM3 could execute on all the embedded platforms, the performance they achieved still needs to be closely comparable with what can be achieved on desktop computers. Appendix D shows the investigation of the sensitivity analysis on what ORB-SLAM3 parameters can be changed to decrease the computational load on the CPU to achieve better performance while executing.

Profiling ORB-SLAM3 was also limited to the available performance events on the Nvidia Jetson TX2 CPU clusters. If more events were available, a more detailed analysis could have been done while profiling ORB-SLAM3 on the Nvidia Jetson TX2. Helpful information might also be obtained if ORB-SLAM3 was profiled on the other embedded platforms, but due to licencing issues, profiling using ARM MAP was only possible on the Nvidia Jetson TX2.

7.4 Closure

SLAM is a complex problem and can be used for various tasks such as search and rescue operations, military surveillance and attack, and automatic explorations of areas. Different SLAM techniques exist, and in this study, there was a focus on ORB-SLAM3, a form of visual SLAM. ORB-SLAM3 is a versatile SLAM technique as it provides the ability to use different configurations of camera setups and the recent addition of using inertial data along with the images taken by the camera. One of the significant problems with SLAM is that it is very computationally expensive and is usually developed, tested, and optimized on a high-end desktop. Practically, SLAM will be implemented on an embedded platform that does not have the same computational power as a high-end desktop machine. Thus, this study aimed to create a model that can predict the performance of ORB-SLAM3 on embedded platforms.

This study showed that the performance that ORB-SLAM3 can achieve on embedded platforms can be predicted. Through profiling and investigation into the execution of the code, different models were created to estimate the performance of ORB-SLAM3 on embedded platforms. The predictive models' target was the average tracking time of ORB-SLAM3, and the inputs to the models were a selection of the PassMark benchmarking results. The models could predict the performance of ORB-SLAM3 with satisfactory results, with the best model for scenario 1 having 2.84%, 3.93%, 0.95 for the MAE, RMSE and the R^2 score, respectively. For the second scenario, the MAE, RMSE, and the R^2 scores were 9.12% ,13.14% ,0.78.

References

[1] D. Kumiawan, A. N. Jati, and U. Sunarya, "A study of 2D indoor localization and mapping using FastSLAM 2.0", in *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, 2016, pp. 152–156. DOI: 10.1109/ICCEREC.2016.7814991.

[2] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF", in *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints", *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[4] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features", in *European conference on computer vision*, Springer, 2006, pp. 404–417.

[5] J. Blom, C. E. Van Daalen, and P. G. Wiid, "Accurate Localisation of a Multi-rotor Using Monocular Vision", *Stellenbosch University*, 2018.

[6] T. Lee and T. Hollerer, "Hybrid Feature Tracking and User Interaction for Markerless Augmented Reality", in *2008 IEEE Virtual Reality Conference*, 2008, pp. 145–152. DOI: 10.1109/VR.2008.4480766.

[7] S. B. Knorr and D. Kurz, "Leveraging the User's Face for Absolute Scale Estimation in Handheld Monocular SLAM", in *2016 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2016, pp. 11–17. DOI: 10.1109/ISMAR.2016.20.

[8] X. Gao, T. Zhang, Y. Liu, and Q. Yan, *14 Lectures on Visual SLAM: From Theory to Practice*. Publishing House of Electronics Industry, 2017.

[9] T. Taketomi, H. Uchiyama, and S. Ikeda, "Visual SLAM algorithms: a survey from 2010 to 2016", *IPSJ Transactions on Computer Vision and Applications*, vol. 9, 2017. DOI: 10.1186/s41074-017-0027-2.

[10] A. Geiger, P Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: the KITTI dataset", *The International Journal of Robotics Research*, vol. 32, pp. 1231–1237, 2013. DOI: 10.1177/0278364913491297.

[11] M. Burri, J. Nikolic, P. Gohl, *et al.*, "The EuRoC micro aerial vehicle datasets", *The International Journal of Robotics Research*, vol. 35, 2016. DOI: 10.1177/0278364915620033.

[12] A. Handa, T. Whelan, J. McDonald, and A. J. Davison, "A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM", in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 1524–1531. DOI: 10.1109/ICRA.2014.6907054.

[13] J. Engel, V. Usenko, and D. Cremers, "A Photometrically Calibrated Benchmark For Monocular Visual Odometry", *ArXiv*, 2016.

[14] M. Abouzahir, A. El Ouardi, R. Latif, S. Bouaziz, and T. Abdelouahed, "Embedding SLAM algorithms: Has it come of age?", *Robotics and Autonomous Systems*, vol. 100, 2017. DOI: 10.1016/j.robot.2017.10.019.

[15] M. Montemerlo and S. Thrun, "FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges", *IEEE Rob. Autom.*, vol. 13, pp. 99–110, 2007. DOI: 10.1007/978-3-540-46402-0_4.

[16] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM: A Versatile and Accurate Monocular SLAM System", *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015. DOI: 10.1109/TRO.2015.2463671.

[17] M. Milford, G. Wyeth, and D. Prasser, "RatSLAM: a hippocampal model for simultaneous localization and mapping", in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, vol. 1, 2004, 403–408 Vol.1. DOI: 10.1109/ROBOT.2004.1307183.

[18] N. Otterness, M. Yang, S. Rust, *et al.*, "An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads", in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 353–364. DOI: 10.1109/RTAS.2017.3.

[19] T. Peng, D. Zhang, D. L. N. Hettiarachchi, and J. S. Loomis, "An Evaluation of Embedded GPU Systems for Visual SLAM Algorithms", vol. 2020, 2020, pp. 325–1–325–6.

[20] L. Nardi, B. Bodin, M. Z. Zia, *et al.*, "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM", in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 5783–5790. DOI: 10.1109/ICRA.2015.7140009.

[21] L. Ma, J. M. Falquez, S. McGuire, and G. Sibley, "Large scale dense visual inertial slam", in *Field and Service Robotics: Results of the 10th International Conference*, D. S. Wettergreen and T. D. Barfoot, Eds. Cham: Springer International Publishing, 2016, pp. 141–155, ISBN: 978-3-319-27702-8. DOI: 10.1007/978-3-319-27702-8_10. [Online]. Available: https://doi.org/10.1007/978-3-319-27702-8_10.

[22] R. A. Newcombe, S. Izadi, O. Hilliges, *et al.*, "KinectFusion: Real-time dense surface mapping and tracking", in *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, 2011, pp. 127–136. DOI: 10.1109/ISMAR.2011.6092378.

[23] B. Bodin, H. Wagstaff, S. Saecdi, *et al.*, "SLAMBench2: Multi-Objective Head-to-Head Benchmarking for Visual SLAM", in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 3637–3644. DOI: 10.1109/ICRA.2018.8460558.

[24] M. Bujanca, P. Gafton, S. Saeedi, *et al.*, "SLAMBench 3.0: Systematic Automated Reproducible Evaluation of SLAM Systems for Robot Vision Challenges and Scene Understanding", in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 6351–6358. DOI: 10.1109/ICRA.2019.8794369.

[25] G. Mohanarajah, V. Usenko, M. Singh, R. D'Andrea, and M. Waibel, "Cloud-Based Collaborative 3D Mapping in Real-Time With Low-Cost Robots", *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 423–431, 2015. DOI: 10.1109/TASE.2015.2408456.

[26] J. Engel, T. Schoeps, and D. Cremers, "LSD-SLAM: large-scale direct monocular SLAM", *European Conference on Computer Vision*, vol. 8690, pp. 1–16, 2014. DOI: 10.1007/978-3-319-10605-2_54.

[27] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras", *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017. DOI: 10.1109/TRO.2017.2705103.

[28] G. Hull, "Real-time occupancy grid mapping using LSD-SLAM", *Stellenbosch University*, 2017.

[29] S. Ullah, B. Song, and W. Chen, "EMoVI-SLAM: Embedded Monocular Visual Inertial SLAM with Scale Update for Large Scale Mapping and Localization", in *2018 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, 2018, pp. 880–885. DOI: 10.1109/ROMAN.2018.8525794.

[30] S. García, M. E. López, R. Barea, L. M. Bergasa, A. Gómez, and E. J. Molinos, "Indoor SLAM for Micro Aerial Vehicles Control Using Monocular Camera and Sensor Fusion", in *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2016, pp. 205–210. DOI: 10.1109/ICARSC.2016.46.

[31] M. Ludhiyani, V. Rustagi, R. Dasgupta, and A. Sinha, "Multirotor dynamics based online scale estimation for monocular SLAM", in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 6475–6481. DOI: 10.1109/ICRA.2019.8794372.

[32] L. Nieto-Hernández, A. A. Gómez-Casasola, and H. Rodríguez-Cortés, "Monocular SLAM Position Scale Estimation for Quadrotor Autonomous Navigation", in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2019, pp. 1359–1364. DOI: 10.1109/ICUAS.2019.8797951.

[33] Y. Li, C. Xie, H. Lu, X. Chen, J. Xiao, and H. Zhang, "Scale-aware Monocular SLAM Based on Convolutional Neural Network", in *2018 IEEE International Conference on Information and Automation (ICIA)*, 2018, pp. 51–56. DOI: 10.1109/ICInfA.2018.8812582.

[34] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM", *IEEE Transactions on Robotics*, pp. 1–17, 2021. DOI: 10.1109/TRO.2021.3075644.

[35] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection", *Computer Vision–ECCV 2006*, pp. 430–443, 2006.

[36] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary Robust Independent Elementary Features", *Computer Vision – ECCV 2010 Lecture Notes in Computer Science*, 778–792, 2010. DOI: 10.1007/978-3-642-15561-1_56.

[37] J. Hennessy and D. Patterson, *Computer Architecture – A Quantitative Approach 5th edition*. Elsevier Science, 2012, ISBN: 978-0-12-383872-8.

[38] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface, ARM edition*. Elsevier, 2016.

[39] RISC, "What is RISC", *[Online]. Available: https://cs.stanford.edu/people/eroberts/-courses/soco/projects/2000-01/risc/whatis/*, [Accessed: 12- Jun- 2021].

[40] J. Hruska, "Get Ready for the Most Interesting CPU Market We've Seen in Decades", *[Online]. Available: https://www.extremetech.com/computing/315354-get-ready-for-the-most-interesting-cpu-market-weve-seen-in-30-years*, [Accessed: 7- Apr- 2021].

[41] ARM, "Architecture and Microarchitecture", *[Online]. Available: https://developer.arm.com/documentation/102404/0200/Architecture-and-micro-architecture*, [Accessed: 12- Jun- 2021].

[42] S. L. Harris and D. M. Harris, *Digital design and computer architecture (ARM edition)*. Morgan Kaufmann, 2016.

[43] J. R. C. Patterson, "Modern Microprocessors A 90-Minute Guide!", *[Online]. Available: http://www.lighterra.com/papers/modernmicroprocessors/*, [Accessed: 12- Jun- 2021].

[44] K. Gupta and T. Sharma, "Changing Trends in Computer Architecture : A Comprehensive Analysis of ARM and x86 Processors", *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 2021.

[45] A. A. Abudaqa, T. M. Al-Kharoubi, M. F. Mudawar, and A. Kobilica, "Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with Gem5 simulator", *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, pp. 317–322, 2018.

[46] K. Sankaralingam, J. Menon, and E. R. Blem, "A Detailed Analysis of Contemporary ARM and x86 Architectures", 2013.

[47] NVIDIA, "Jetson TX2 Module", *[Online]. Available: https://developer.nvidia.com-/embedded/jetson-tx2*, [Accessed: 7- Apr- 2021].

[48] R. Barnes, "Raspberry Pi 3: Specs, benchmarks and testing", *[Online]. Available: https://magpi.raspberrypi.com/articles/raspberry-pi-3-specs-benchmarks*, [Accessed: 16- Feb- 2023].

[49] R. P. T. Ltd, "Raspberry Pi 4 Model B specifications", *[Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/*, [Accessed: 16- Feb- 2023].

[50] Mamri, Ayoub, Abouzahir, Mohamed, Ramzi, Mustapha, and Latif, Rachid, "ORB-SLAM accelerated on heterogeneous parallel architectures", *E3S Web Conf.*, vol. 229, p. 01 055, 2021. DOI: 10.1051/e3sconf/202122901055. [Online]. Available: https://doi.org/10.1051/e3sconf/202122901055.

[51] W. Fang, Y. Zhang, B. Yu, and S. Liu, "FPGA-based ORB feature extraction for real-time visual SLAM", in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 275–278. DOI: 10.1109/FPT.2017.8280159.

[52] Q. Fu, H. Yu, X. Wang, Z. Yang, H. Zhang, and A. S. Mian, "FastORB-SLAM: a Fast ORB-SLAM Method with Coarse-to-Fine Descriptor Independent Keypoint Matching", *ArXiv*, vol. abs/2008.09870, 2020.

[53] J. Cheng, L. Zhang, Q. Chen, K. Zhou, and R. Long, "A Fast and Accurate Binocular Visual-Inertial SLAM Approach for Micro Unmanned System", in *2021 IEEE 4th International Conference on Electronics Technology (ICET)*, 2021, pp. 971–976. DOI: 10.1109/ICET51757.2021.9450932.

[54] Smartbear.com, "Fundamentals of Performance Profiling", *[Online]. Available: https://smartbear.com/learn/code-profiling/fundamentals-of-performance-profiling/*, [Accessed: 12- Jun- 2021].

[55] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering", in *Future of Software Engineering (FOSE '07)*, 2007, pp. 171–187. DOI: 10.1109/FOSE.2007.32.

[56] ARM, "Arm MAP: Scalable Profiler for Server/HPC Developers", *[Online]. Available: https://www.arm.com/products/development-tools/server-and-hpc/forge/map?ARMMAP*, [Accessed: 12- Jun- 2021].

[57] W. Ye, Y. Zhao, and P. A. Vela, "Characterizing SLAM Benchmarks and Methods for the Robust Perception Age", *ArXiv*, vol. abs/1905.07808, 2019.

[58] NVIDIA, "NVIDIA Parker Series SoC Technical Reference Manual", *NVIDIA*, 2017.

[59] ARM, "ARM Cortex-A57 MPCore Processor Technical Reference Manual", *ARM*, 2013.

[60] ARM, "Cortex-A72 Software Optimization Guide", *ARM*, 2016.

[61] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's First 64-bit ARM Processor", *IEEE Micro*, vol. 35, no. 2, pp. 46–55, 2015. DOI: 10.1109/MM.2015.12.

[62] ARM, "Cortex-A53", *[Online]. Available: https://developer.arm.com/Processors/Cortex-A53*, [Accessed: 16- Feb- 2023].

[63] ARM, "Cortex-A72 Software Optimization Guide", *ARM*, 2016.

[64] S. Lovegrove *et al.*, "Pangolin", *GitHub repository*, 2021.

[65] OpenCV, "Opencv.org, 2020", *[Online]. Available: https://opencv.org*, [Accessed: 12- Jun- 2021].

[66] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3", *[Online]. Available: http://eigen.tuxfamily.org*, [Accessed: 12- Jun- 2021].

[67] D. Galvez-López and J. D. Tardos, "Bags of Binary Words for Fast Place Recognition in Image Sequences", *IEEE Transactions on Robotics*, vol. 28, no. 5, pp. 1188–1197, 2012. DOI: 10.1109/TRO.2012.2197158.

[68] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o: A general framework for graph optimization", *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, pp. 3607 –3613, 2011. DOI: 10.1109/ICRA.2011.5979949.

[69] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697.

[70] ROS, "Ros.org, 2020", *[Online]. Available: https://www.ros.org*, [Accessed: 12- Jun- 2020].

[71] "taskset(1) - linux manual page", *[Online]. Available: https://man7.org/linux/man-pages/man1/taskset.1.html*, [Accessed: 7- Apr- 2021].

[72] UZ-SLAMLab, "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM", *[Online]. Available: https://github.com/UZ-SLAMLab/ORB_SLAM3*, [Accessed: 7- Apr- 2021].

[73] Microsoft, *Visual Studio Code - Code Editing. Redefined*, 2016. [Online]. Available: https://code.visualstudio.com/.

[74] Datadog, "What are containerized applications? — datadog", Feb. 2022. [Online]. Available: \url{https://www.datadoghq.com/knowledge-center/containerized-applications/}.

[75] Docker, "Enterprise Application Container Platform — Docker", 2018. [Online]. Available: \url{https://www.docker.com/}.

[76] "Linux Perf - Perf Wiki", *[Online]. Available: https://perf.wiki.kernel.org/index.php/-Main_Page*, [Accessed: 7- Apr- 2021].

[77] DynamoRIO, "Existing DynamoRIO-based tools", *[Online]. Available: https://dynamorio.org/*, [Accessed: 7- Apr- 2021].

[78] PassMark, "Download PassMark PerformanceTest - PC Benchmark Software", *[Online]. Available: https://www.passmark.com/products/performancetest/index.php*, [Accessed: 16- Feb- 2023].

[79] Geekbench, "Geekbench 5 - Cross-Platform Benchmark", *[Online]. Available: https://www.geekbench.com/*, [Accessed: 16- Feb- 2023].

[80] Cinebench, "Maxon - 3D for the Real World", *[Online]. Available: https://www.maxon.net/en/cinebench*, [Accessed: 16- Feb- 2023].

[81] 3DMark, "3DMark.com - Share and compare scores from UL benchmarks", *[Online]. Available: https://www.3dmark.com/*, [Accessed: 16- Feb- 2023].

[82] PassMark, "PassMark Software - List of Benchmarked CPUs", *[Online]. Available: https://www.cpubenchmark.net/cpu_list.php*, [Accessed: 16- Feb- 2023].

[83] Scikit-learn, "sklearn.linear_model.LinearRegression — scikit-learn 0.22 documentation", *[Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html*,

[84] Scikit-learn, "sklearn.ensemble.ExtraTreesRegressor", *[Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesRegressor.html*, [Accessed: 16- Feb- 2023].

[85] Scikit-learn, "sklearn.neural_network.MLPRegressor", *[Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html*, [Accessed: 16- Feb- 2023].

[86] J. Brownlee, "Why Use Ensemble Learning?", *[Online]. Available: https://machinelearningmastery.com/why-use-ensemble-learning/*, Oct. [Accessed: 16- Feb- 2023].

Appendices

A Dockerfile

The results and the code used to obtain the results are provided online and can be found here.

```
FROM ubuntu:18.04

ARG TARGETOS
ARG TARGETARCH
ARG TARGETVARIANT

RUN echo "I'm building for $TARGETOS/$TARGETARCH/$TARGETVARIANT"

RUN apt-get update && apt-get upgrade -y

RUN apt-get install build-essential cmake git nano -y

RUN apt-get install libc6-dbg gdb valgrind -y

RUN apt-get install libgl1-mesa-dev libglew-dev libpython2.7-dev
   libeigen3-dev libboost-all-dev libssl-dev -y

COPY ./opencv-4.1.1 /home/dusr/repos/opencv-4.1.1
WORKDIR /home/dusr/repos/opencv-4.1.1
RUN mkdir build \
        && cd build \
        && cmake .. \
        && cmake
           -DCMAKE_INSTALL_PREFIX=/home/dusr/installs/opencv/Opencv-4.1.1
           -DCMAKE_CXX_FLAGS_RELEASE:STRING="-O3 -DNDEBUG -g"
           -DCMAKE_C_FLAGS_RELEASE:STRING="-O3 -DNDEBUG -g"
           -DENABLE_PROFILING:BOOL=ON -DBUILD_ITT:BOOL=OFF
           -DBUILD_JAVA:BOOL=OFF -DBUILD_JPEG:BOOL=ON
           -DBUILD_PACKAGE:BOOL=OFF -DBUILD_PERF_TESTS:BOOL=OFF
           -DBUILD_PROTOBUF:BOOL=OFF -DBUILD_TESTS:BOOL=OFF
           -DBUILD_opencv_apps:BOOL=OFF
           -DBUILD_opencv_calib3d:BOOL=ON
           -DBUILD_opencv_dnn:BOOL=OFF -DBUILD_opencv_flann:BOOL=ON
           -DBUILD_opencv_gapi:BOOL=OFF
           -DBUILD_opencv_imgcodecs:BOOL=ON
           -DBUILD_opencv_highgui:BOOL=ON
           -DBUILD_opencv_imgproc:BOOL=ON
           -DBUILD_opencv_java_bindings_generator:BOOL=OFF
           -DBUILD_opencv_ml:BOOL=OFF -DBUILD_opencv_photo:BOOL=OFF
```

```
            -DBUILD_opencv_python_bindings_generator:BOOL=OFF
            -DBUILD_opencv_python_tests:BOOL=OFF
            -DBUILD_opencv_stitching:BOOL=OFF
            -DBUILD_opencv_ts:BOOL=OFF -DBUILD_opencv_video:BOOL=ON
            -DBUILD_opencv_videoio:BOOL=OFF
            -DENABLE_PRECOMPILED_HEADERS:BOOL=OFF
            -DWITH_FFMPEG:BOOL=OFF -DWITH_GSTREAMER:BOOL=OFF
            -DWITH_GTK:BOOL=OFF -DWITH_IMGCODEC_HDR:BOOL=OFF
            -DWITH_IMGCODEC_PFM:BOOL=OFF -DWITH_IMGCODEC_PXM:BOOL=OFF
            -DWITH_IMGCODEC_SUNRASTER:BOOL=OFF -DWITH_ITT:BOOL=OFF
            -DWITH_JASPER:BOOL=OFF -DWITH_JPEG:BOOL=OFF
            -DWITH_LAPACK:BOOL=OFF -DWITH_OPENCLAMDBLAS:BOOL=OFF
            -DWITH_OPENCLAMDFFT:BOOL=OFF -DWITH_OPENEXR:BOOL=OFF
            -DWITH_QUIRC:BOOL=OFF -DWITH_TIFF:BOOL=OFF
            -DWITH_WEBP:BOOL=OFF -DBUILD_opencv_python2:BOOL=OFF
            -DBUILD_opencv_python3:BOOL=OFF .. \
23          && make -j16 \
24          && make install -j4
25  WORKDIR /home/dusr/repos
26  RUN rm -rf opencv-4.1.1
27
28  ENV
       PATH="/home/dusr/installs/opencv/Opencv-4.1.1/bin:/home/dusr/installs/ope
29
30  COPY ./Pangolin /home/dusr/repos/Pangolin
31  WORKDIR /home/dusr/repos/Pangolin
32  RUN mkdir build \
33          && cd build \
34          && cmake .. \
35          && make -j16
36
37  COPY ./workspaceORB-master /home/dusr/workspaceORB-master/
38  WORKDIR /home/dusr/workspaceORB-master/
39  RUN mkdir build \
40          && cd Thirdparty/DBoW2/ \
41          && mkdir build \
42          && cd build \
43          && cmake -DLOC=/home/dusr/installs/ORBThreads/OpenCV4.1.1 ..
               \
44          && make -j16
45
46  WORKDIR /home/dusr/workspaceORB-master/
```

```
47 RUN cd  Thirdparty/g2o/ \
48         && mkdir build \
49         && cd build \
50         && cmake -DLOC=/home/dusr/installs/ORBThreads/OpenCV4.1.1 ..
              \
51         && make -j16
52
53 WORKDIR /home/dusr/workspaceORB-master/
54 RUN cd build \
55         && cmake -DLOC=/home/dusr/installs/ORBThreads/OpenCV4.1.1 ..
              \
56         && make -j16
57 WORKDIR /home/dusr
58 RUN rm -rf workspaceORB-master
59
60 COPY ./data /home/dusr/installs/ORBThreads/data
61 COPY ./Run.sh /home/dusr/installs/ORBThreads
62 COPY ./Auto_Tests.sh /home/dusr/installs/ORBThreads
63 WORKDIR /home/dusr/installs/ORBThreads
64 RUN chmod +x Run.sh
65 RUN chmod +x Auto_Tests.sh
```

Listing 2: **Representation of how FAST function is executed within the ComputeKeyPointOctTree**

B CPU benchmarking results

The results and the code used to obtain the results are provided online and can be found here. Since the Table is too large to display it can be found in the above link under “Results Results_Table.csv”

C Experiment 1 - Results tables

This Appendix contains the results that were obtained for the investigation into the performance enchancemnet of OpenCV and ORB-SLAM3 as mentioned in Section D.2.1. The results and the code used to obtain the results are provided online and can be found here.

C.1 ORB-SLAM3 performance

These tables represents the execution performance of ORB-SLAM3 during execution for the OpenCV and ORB-SLAM3 configurations.

Table 23: **ORB-SLAM3v3 beta version results with OpenCV configurations**

Configuration	Mean Tracking Time [ms]	ATE RMSE [m]	ORB Extraction [ms]	Local Map Track [ms]
OpenCV3.2.0PT3	100.54	0.060	53.18	34.43
OpenCV3.2.0MP3	99.35	0.065	53.96	33.13
OpenCV3.2.0TBB3	99.53	0.058	52.66	33.89
OpenCV4.1.1PT3	99.96	0.068	39.26	43.88
OpenCV4.1.1MP3	102.57	0.071	41.45	44.44
OpenCV4.1.1TBB3	99.65	0.070	40.40	42.63
OpenCV4.1.1Nvidia3	102.47	0.052	41.30	44.09

Table 24: **ORB-SLAM3v4 beta version results with OpenCV configurations**

Configuration	Mean Tracking Time [ms]	ATE RMSE [m]	ORB Extraction [ms]	Local Map Track [ms]
OpenCV3.2.0PT4	87.35	0.062	51.93	23.67
OpenCV3.2.0MP4	87.22	0.062	52.54	23.47
OpenCV3.2.0TBB4	86.24	0.048	51.30	23.35
OpenCV4.1.1PT4	73.70	0.159	34.62	24.51
OpenCV4.1.1MP4	75.50	0.265	36.75	24.37
OpenCV4.1.1TBB4	74.47	0.152	35.91	24.25
OpenCV4.1.1Nvidia4	75.08	0.051	36.62	23.77

C.2 ORB-SLAM3 Profiling

This section contains the profiling information while executing different ORB-SLAM3 and OpenCV configurations.

Table 25: **Experiment 1 - Instruction mix**

Configuraion	Instructions executed	Percentage Load/Store [%]	Percentage ALU [%]	Percentage branch [%]
ORB-SLAM3v3 OpenCV3.2.0	1.01E+12	36.36	47.95	15.69
ORB-SLAM3v3 OpenCV4.1.1	9.01E+11	34.47	50.02	15.51
ORB-SLAM3v4 OpenCV4.1.1	7.06E+11	32.76	51.97	15.27

Table 26: **Experiment 1 - ALU mix**

Configuraion	Percentage data processing/inte [%]	Percentage floating point [%]	Percentage SIMD [%]
ORB-SLAM3v3 OpenCV3.2.0	85.87	6.10	8.02
ORB-SLAM3v3 OpenCV4.1.1	77.06	6.41	16.54
ORB-SLAM3v4 OpenCV4.1.1	72.50	7.40	20.09

Table 27: **Experiment 1 - Cache performance**

Configuraion	Cache accesses	Cache miss rate [%]
ORB-SLAM3v3 OpenCV3.2.0	3.77E+11	1.47
ORB-SLAM3v3 OpenCV4.1.1	3.24E+11	1.80
ORB-SLAM3v4 OpenCV4.1.1	2.40E+11	1.89

D Sensitivity Analysis

D.1 Introduction

In Chapter 4 two aspects were identified that might increase the performance of ORB-SLAM3 on the Nvidia Jetson TX2. The first aspect is to increase the performance of the FAST algorithm of OpenCV by changing the version and build settings. Secondly, three ORB-SLAM3 parameters were identified that should reduce the number of times the FAST algorithm is called.

This Appendix will describe the two methods used to investigate the performance that ORB-SLAM3 achieves when implementing the changes. The first section will describe the experiment that investigates the OpenCV versions and build settings, whereas, the second section will highlight the experiment used to investigate the parameter changes.

D.2 Experiment 1 - Increasing OpenCV perfromance

ORB-SLAM3 is highly dependent on the OpenCV [65] library. Thus ORB-SLAM3 will be tested with different OpenCV build configurations to get the configuration of OpenCV that provides the fastest results. Two versions of OpenCV will be tested: version 3.2.0 and 4.1.1. These versions were chosen as ORB-SLAM3 was initially developed for OpenCV 3.2.0, OpenCV 4.1.1 was chosen to compare it with another optimized OpenCV 4.1.1 version provided by Nvidia for the Nvidia Jetson TX2. OpenCV 4.1.1 also uses ARM's NEON SIMD instructions for the FAST algorithm , whereas OpenCV 3.2.0 does not. OpenCV will also be built with three different parallel frameworks: Pthreads, OpenMP, and TBB. The following settings will be set for all OpenCV build configurations as suggested by Nvidia developers:

1. ARM Neon (SIMD) flag enabled.
2. Optimization flag set to -O3.
3. Tune flag (-mtune) set to A57.
4. Architecture flag (-march) set to A57.

Table 28 shows the seven OpenCV build configurations that will be tested with ORB-SLAM3.

At the start of the study ORB-SLAM3 0.3 beta was the latest version; as time progressed the authors released a new version of ORB-SLAM3 0.4 beta. It was decided that both versions of ORB-SLAM3 will be tested with the OpenCV build configurations mentioned in Table 28.

Table 28: **OpenCV Build configurations**

OpenCV configuration name	OpenCV version	Parallel framework
OpenCV3.2.0PT	3.2.0	PThreads
OpenCV3.2.0MP	3.2.0	OpenMP
OpenCV3.2.0TBB	3.2.0	TBB
OpenCV4.1.1PT	4.1.1	PThreads
OpenCV4.1.1MP	4.1.1	OpenMP
OpenCV4.1.1TBB	4.1.1	TBB
OpenCV4.1.1Nvidia	Nvidia Optimized	TBB

Table 29: **ORB-SLAM3 versions**

ORB-SLAM3 configuration name	ORB-SLAM3 version
ORB-SLAM3v3	ORB-SLAM3 version 0.3 beta
ORB-SLAM3v4	ORB-SLAM3 version 0.4 beta

Table 29 indicates how each ORB-SLAM3 version will be referred to for the rest of the study, and their differences are discussed in Section 4.4.1.

Both ORB-SLAM3 versions will be built with each OpenCV build configuration from which the performance will be measured to determine what configuration provides the best results.

As mentioned in Section 3.3.2.1 the performance metrics are defined as speed, accuracy, and robustness. However, for this experiment the focus is only on speed and accuracy. The details on how the speed and accuracy metrics are measured and calculated are discussed in Section 3.3.2.1.

D.2.1 Experiment 1 - Methodology

The aim of this experiment is to see how changing the OpenCV version and build settings alters the performance of ORB-SLAM3 on the Nvidia Jetson TX2. The changes of the OpenCV settings will also be tested with two versions of ORB-SLAM3, v0.3 and v0.4. The method can be broken into the following steps:

1. Build all OpenCV configurations listed in Table 28.
2. Build the two versions of ORB-SLAM3 listed in Table 29 with all OpenCV configurations listed in Table 28.
3. Run all ORB-SLAM3 configurations on EuRoC MAV V201 sequence for five repetitions.

4. Calculate the average total tracking time and the absolute trajectory error for all sequences.

5. Compare results of all configurations to determine what configuration provides the best results

After the comparison is made, the execution of the initial ORB-SLAM3 configuration and the best performing ORB-SLAM3 configuration will be profiled. The configuration that provides the best performance will be used in the second experiment that investigates the parameter changes of ORB-SLAM3.

D.3 Experiment 2 - Increase ORB-SLAM3 performance

The experiment aims to see how parameter changes will influence the performance of ORB-SLAM3. The parameters that have been identified to be investigated are the:

1. Input image resolution

2. The number of features to extract per frame

3. The number of pyramid levels

The reason why these three parameters where chosen is discussed in Section 4.4.2.

D.3.1 Experiment 2 - Methodology

Each of the parameters will be adjusted individually to see how the changes affect the performance of ORB-SLAM3. A total of 12 different individual adjustments will be made to the three parameters.

The changes in parameters will be used to construct a naming convention that will be used throughout the rest of the study and will be further referred to as the "Configuration." The convention that will be used to construct the configuration name is: X_Y_Z, where X is the percentage reduction in input image resolution, Y is the maximum features to extract per frame, and Z is the number of pyramid levels. The 12 configurations that will be investigated are listed in Table 30. All of the configurations will be tested on the four EuRoC MAV dataset [11] sequences, Table 31, using the flow diagram of Figure 14.

Figure 14 illustrates that the configurations listed in Table 30 are loaded for execution of ORB-SLAM3. After execution, the appropriate time statistics and estimated trajectory are saved to a file for processing. This process is repeated five times to determine the robustness

of the changes. After the five repetitions, the average time statistics and ATE is calculated. The execution of ORB-SLAM3 is then profiled using ARM Map [56]. The details on what exactly is profiled by ARM Map is discussed in Section 4.2.

Table 30: **Experiment configuration table**

Configuration name	Resolution	Features	Levels
100_1000_8	480 × 752	1000	8
95_1000_8	456 × 714	1000	8
90_1000_8	432 × 676	1000	8
80_1000_8	384 × 601	1000	8
70_1000_8	336 × 526	1000	8
60_1000_8	288 × 451	1000	8
50_1000_8	240 × 376	1000	8
100_500_8	480 × 752	500	8
100_250_8	480 × 752	250	8
100_200_8	480 × 752	200	8
100_1000_6	480 × 752	1000	6
100_1000_4	480 × 752	1000	4

Table 31: **EuRoC MAV dataset sequences that will be used in for verification**

Dataset type	Sequence name	Difficulty
Machine hall	MH01	Easy
Machine hall	MH02	Easy
Vicon room	V101	Easy
Vicon room	V201	Easy

After the results of the three-parameter changes have been investigated, the best performing variation of each parameter change will be combined with each other to see whether there is an increase in performance for multiple parameter changes. The combination of parameter changes will form new configurations that will be tested using the same methodology that is illustrated in Figure 14 from which the best combination of parameters will be chosen

D.4 Method verification and validation

The experiments aim to increase the performance of ORB-SLAM3 on the Nvidia Jetson TX2. As verification, the modified ORB-SLAM3 algorithm needs to increase performance without changing the expected output thereof. In Section 3.3.2.1 Table **??** the performance criteria of ORB-SLAM3 was defined that is needed to be classified as real-time. The modified ORB-SLAM3 algorithm can be verified if the performance achieved for the four sequences listed in

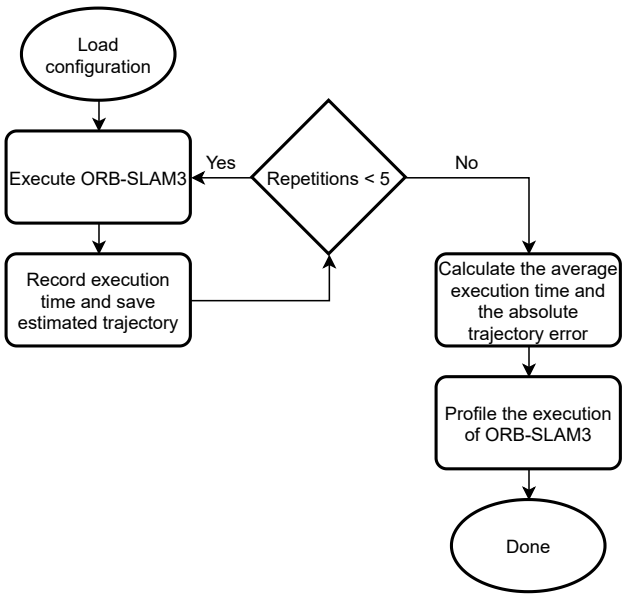


Figure 14: **Flowchart explaining the methodology of the experiment**

Table 31 are within the set criteria, therefore proving that the performance of ORB-SLAM3 has been increased and can be implemented on the Nvidia Jetson TX2.

The modified ORB-SLAM3 algorithm will be validated by testing it on the more challenging sequences of the EuRoC MAV dataset as listed in Table 32. If the performance of the modified ORB-SLAM3 algorithm is within the criteria listed in Section 3.3.2.1 Table **??** the modified ORB-SLAM3 algorithm has been validated. An additional form of validation will also be used by implementing the modified ORB-SLAM3 algorithm on the Denver 2. If the modified ORB-SLAM3 algorithm has a performance increase, it validates an increase in the performance of ORB-SLAM3 and that monocular-inertial SLAM can be implemented on embedded platforms.

Table 32: **EuRoC MAV dataset sequences that will be used for validation**

Dataset type	Sequence name	Difficulty
Machine hall	MH03	Medium
Machine hall	MH04	Difficult
Machine hall	MH05	Difficult
Vicon room	V102	Medium
Vicon room	V103	Difficult
Vicon room	V202	Medium
Vicon room	V203	Difficult

D.5 Conclusion

This appendix described the two experiments that will be used to investigate the performance achieved by ORB-SLAM3 when changing the OpenCV versions and build configurations, along with varying the identified ORB-SLAM3 parameters. In the following appendix the results of the experiments are showed and discussed.

E Sensitivity Analysis - Results

E.1 Introduction

This Appendix will cover the results that were obtained for the experiments described in Appendix D. The Appendix will be divided into a section for each parameter change: input image resolution, number of features, and the pyramid levels. After the individual changes have been discussed, the results obtained concerning the combination of the parameter changes will be addressed.

All the results were obtained by using ORB-SLAM3 v0.4 built with OpenCV 4.1.1 optimized for the Jetson TX2. The investigation into why this configuration of ORB-SLAM3 and OpenCV were chosen can be found in Appendix C. This configuration of ORB-SLAM3 and OpenCV has a speedup of 1.25 for the average total tracking time, primarily due to SIMD instructions used during the FAST algorithm.

E.2 Experiment 1 - Increasing OpenCV perfromance

In this section we will highlight the results that were obtained during Experiment 1 as described in Section D.2. This experiment investigates how the performance changes for different configurations of OpenCV and ORB-SLAM3.

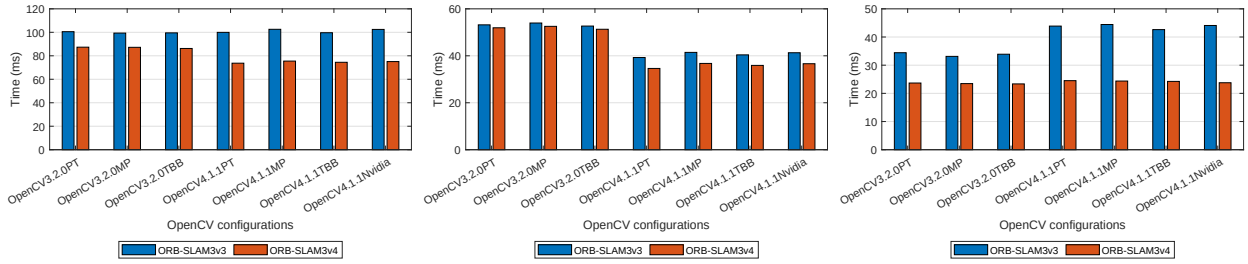
E.2.1 Experiment 1 - Execution statistics

The configuration that provided the fastest results with the highest level of accuracy was used in this study. This section defines the fastest results as the shortest average total tracking time a single sequence achieves over five iterations. The average ORB extraction time and local mapping time will be measured as they are the two portions of the code that make up most of the total tracking time. This time analysis is done by programmatically measuring the time the functions take to execute as provided by the authors of ORB-SLAM3. The results are also provides in Tables 23 and 24 in Appendix C.

Figure 15 shows the time in ms for the total tracking, ORB extraction, and local map time. The y-axis is the time measurement in (ms), the x-axis represents the OpenCV configurations used to build the ORB-SLAM3 versions. The two bins represent ORB-SLAM3v3 and ORB-SLAM3v4, for which the differences were discussed in Section 2.5.1.

From Figures 15a to 15c it is clear that ORB-SLAM3v4 performs faster than ORB-SLAM3v3 for all OpenCV configurations. It is also noted that both ORB-SLAM3 versions do see an increase in speed when extracting ORB features going from OpenCV3.2.0 to OpenCV4.1.1, as shown in Figure 15b. This speed increase is because OpenCV4.1.1 has modified the source

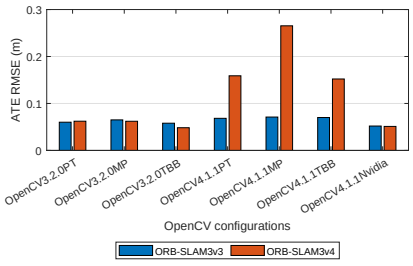
code of the FAST algorithm to include SIMD instructions for the ARM architecture. This, however, harms ORB-SLAM3v3, which sees a decrease in speed for the local mapping portion of ORB-SLAM3v3; this negative impact causes ORB-SLAM3v3 to have no overall speedup changing from OpenCV3.2.0 to OpenCV4.1.1. ORB-SLAM3v4, however, does gain an overall speed increase due to the use of SIMD in OpenCV4.1.1. Besides switching to different versions of OpenCV, the variations in build configurations do not increase any of the execution times.



(a) **Total tracking time** *(b)* **ORB extraction time** *(c)* **Local mapping time**

Figure 15: **Bar graphs comparing the average execution time for different ORB-SLAM3 functions between ORB-SLAM3v3 and ORB-SLAM3v4 build with the different OpenCV configurations**

Focusing on figure 16a it is clear that the ATE of all OpenCV3.2.0 builds configurations along with ORB-SLAM3v3 built with OpenCV4.1.1 stayed closely together, however, ORB-SLAM3v4 experienced larger errors for all of the OpenCV4.1.1 builds, except for the OpenCV-4.1.1Nvidia build. From Figures 15 and 16 it is clear that ORB-SLAM3v4 built with OpenCV-4.1.1Nvidia provides the best results considering the fast execution time and lowest error achieved for this sequence.



(a) **ATE RMSE (m)**

Figure 16: **Bar graph comparing the error achieved during execution between ORB-SLAM3v3 and ORB-SLAM3v4 build with the different OpenCV configurations**

Table 33 shows how ORB-SLAM3v4 built with OpenCV4.1.1Nvidia performs on all of the

EuRoC datasets. From Table 33 it is clear that this configuration between ORB-SLAM3 and OpenCV provides consistent results for all sequences having similar execution times and errors.

Table 33: **Execution results of ORB-SLAM3v4 OpenCV4.1.1Nvidia on all EuRoC MAV sequences**

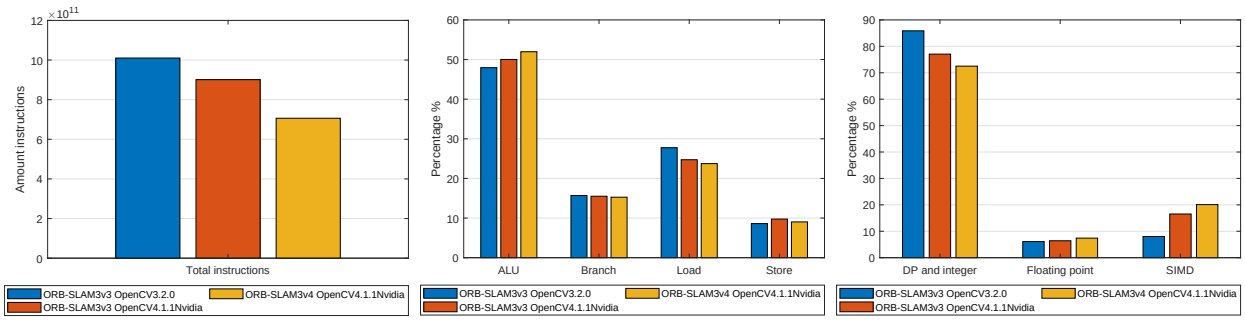
Configuration	Mean Tracking Time [ms]	ATE RMSE [m]	ORB Extraction [ms]	Local Map Track [ms]
MH01	81,49	0,040	41,73	26,63
MH02	80,43	0,036	41,75	24,94
MH03	89,23	0,040	44,29	21,99
MH04	83,34	0,075	40,11	20,86
MH05	83,77	0,063	40,60	21,21
V101	86,34	0,039	38,50	29,56
V102	75,78	0,037	37,11	20,66
V103	72,19	0,065	36,60	19,93
V201	75,56	0,184	36,66	24,07
V202	74,88	0,041	36,28	23,41
V203	71,88	0,055	36,13	18,57

E.2.2 Experiment 1 - Execution profiling

In this section, three configurations of ORB-SLAM3 and OpenCV are profiled. The three configurations are ORB-SLAM3v3 build with OpenCV3.2.0 and OpenCV4.1.1Nvidia, and ORB-SLAM3 build with OpenCV4.1.1Nvidia. The results can be found in Appendix C.

Figure 17 shows the instructions executed, along with the instruction mix during the execution of different configurations. Figure 17a illustrates that the configuration of ORB-SLAM3v4 OpenCV4.1.1Nvidia executes the least instructions of all three configurations. Figure 17b shows that configuration ORB-SLAM3v4-OpenCV4.1.1Nvidia has a higher percentage ALU mix and lower load and store mixes than the other configurations. It is clear that ORB-SLAM3 build with OpenCV4.1.1Nvidia better utilizes the CPU hardware as they have a higher percentage of executed SIMD instructions. This is due to the FAST algorithm now using SIMD instructions.

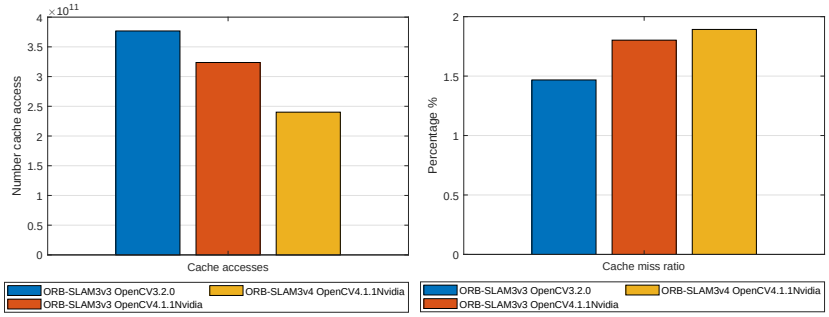
Figure 18 illustrates the cache performance that was achieved by ORB-SLAM3 for different configurations. It is clear that configuration ORB-SLAM3-OpenCV4.1.1Nvidia has the least amount of cache accesses as illustrated by Figure 18a. Configuration ORB-SLAM3-OpenCV4.1.1Nvidia does have the highest cache miss-rate as shown in Figure 18b. However,



(a) **Total instructions executed** *(b)* **Percentage instruction mix** *(c)* **Percentage ALU mix**

Figure 17: **Bar graphs comparing the workload characteristics for different ORB-SLAM3 configurations**

this does not cause concern as it is only 2 % which is almost ideal.



(a) **Total times the cache was accessed** *(b)* **Overall cache miss rate**

Figure 18: **Bar graphs comparing the cache performance for different ORB-SLAM3 configurations**

E.2.3 Closing remark

This section clearly illustrated that by using ORB-SLAM3v4 built with OpenCV4.1.1Nvidia, the performance achieved by the Jetson Tx2 has already been increased. A total speed-up of 1.25 was seen going from the default configuration to the newest release of ORB-SLAM3 along with OpenCV4.1.1 provided by Nvidia for the Jetson Tx2.

E.3 Image Resolution change

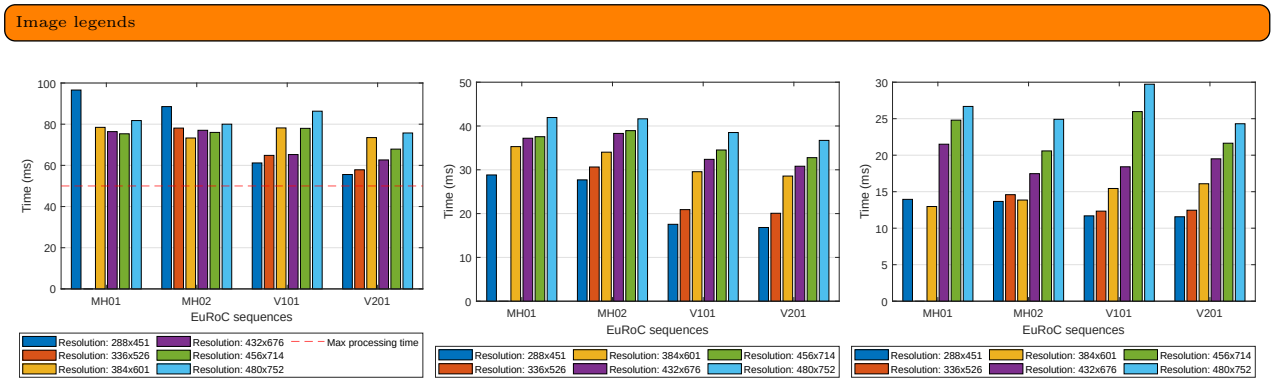
In this section, the input image resolution of the EuRoC sequences has been reduced. The reduction process and all the details regarding this section are discussed in Section 3.3.

E.3.1 Execution statistics

The image resolution is varied for the scales defined in Table 30 on the four EuRoC dataset sequences defined in Table 31. The Tables that were used to create the graphs in this section are provided in Appendix F. This section uses the 480×752 resolution as a baseline since it is the default value for the input sequences.

ORB-SLAM3 failed to execute the input image resolution reduction of $240 \times 376\%$, so it will not form part of this discussion. Figure 19 shows the time in ms for total tracking, ORB extraction, and local mapping. The y-axis is the time measurement in ms; the x-axis represents the EuRoC MAV sequence. The bins represent the resolution of the image used.

Figure 19a shows that non of the tests were able to achieve a total tracking time that is less that the required value. Figures 19b and 19c illustrates that there is an increase in the ORB extraction and local mapping time as the resolution is increased from 288×451 to 480×752.

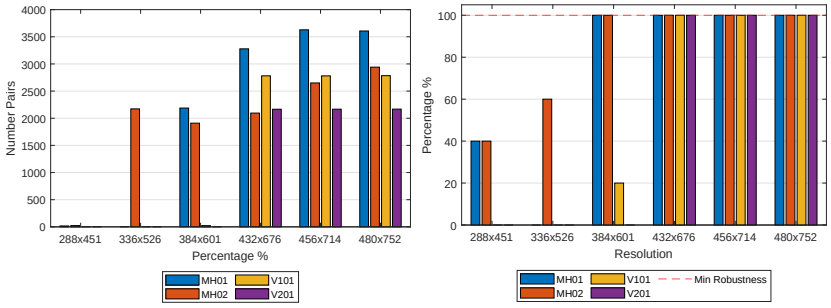


(a) **Total tracking time** *(b)* **ORB extraction time** *(c)* **Local map time**

Figure 19: **Bar graphs comparing the average execution time for different ORB-SLAM3 functions for different image resolutions on four EuRoC MAV sequences**

Figure 20a shows the number of pairs matched between the estimated trajectory provided by ORB-SLAM3 and the ground-truth trajectory provided with the EuRoC MAV sequences. From Figure 20a it is clear that for image resolution of 288×451 to 384×601, there are very few points matched between the two trajectories; thus, the created map will be incomplete. Figure 20b shows that for image resolution of 288×451 to 384×601, ORB-SLAM3 was not

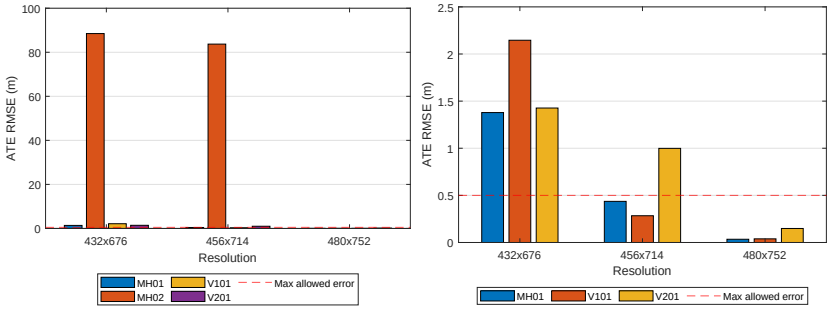
able to create a map on every execution, whereas the other resolutions were able to create a map on every implementation. With the evidence provided in Figure 20a and 20b, it is clear that if an images' resolution is less than 432×676, it will not provide results within the validation criteria. Thus the rest of the discussion will only focus on resolutions of 432×676 and above.



(a) **Pairs matched** *(b)* **Robustness**

Figure 20: **Bar graph comparing the pairs matched and robustness achieved during execution of ORB-SLAM3 for different image resolutions on four EuRoC MAV sequences**

Figure 21a shows that for the image resolutions of 432×676 and 456×714, the ATE of the MH02 sequence is incredibly high; this can be linked back to Figure 20a that illustrates that there are considerably fewer pairs matched for lower resolutions images. Figure 21b shows the ATE results with the MH02 sequence removed to investigate the ATE of the other resolutions. It is noted from Figure 21b that the ATE for the image resolution of 432×676 has an error above 1 m for all sequences. At an image resolution of 456×714 %, the required ATE of less than 0.5 m is achieved for only two of the remaining sequences.



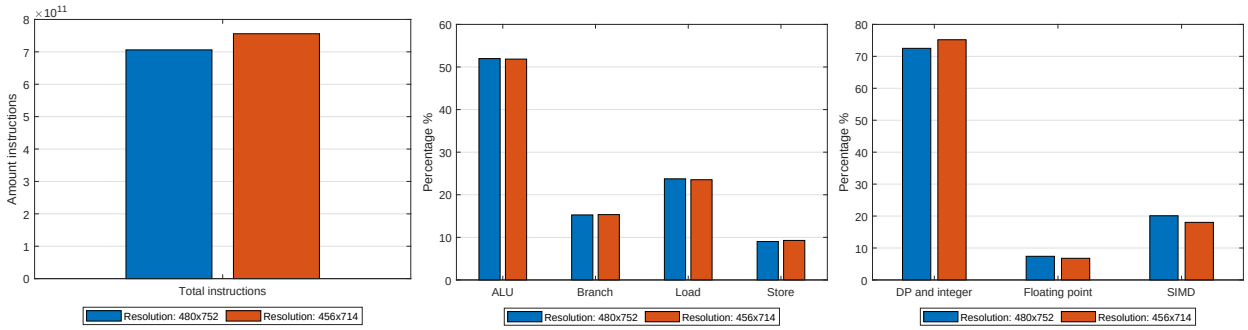
(a) **ATE RMSE 90% to 100%** *(b)* **ATE RMSE 90% to 100% without MH02**

Figure 21: **Bar graph comparing the ATE of ORB-SLAM3 for different image resolutions on four EuRoC MAV sequences**

E.3.2 Execution profiling

This section will look into the profiling of ORB-SLAM3 for image resolutions of 480×752 and 456×714. The results obtained are provided in the tables in Appendix F.2.

Figure 22 shows the workload characteristics of ORB-SLAM3 for different image resolutions. Analyzing Figure 22a shows that more instructions were executed for a lower image resolution which contradicts the expectation that a lower resolution will result in fewer instructions executed. Figure 22b shows that both image resolutions have similar instruction mixes. Figure 22c shows that both image resolutions are dominated by data processing and integer instructions and that the lower resolution has a slightly higher percentage value for this type of instruction.



(a) **Total instructions executed** *(b)* **Percentage instruction mix** *(c)* **Percentage ALU mix**

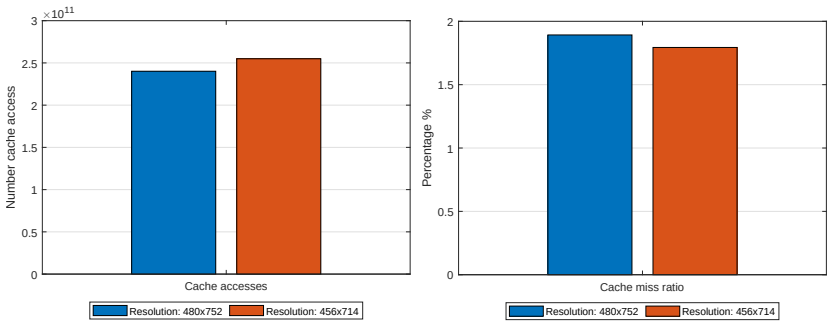
Figure 22: **Bar graphs comparing the workload characteristics for different image resolutions**

Figure 23 shows the cache performance of ORB-SLAM3 while executing different image resolutions. Figure 23a and 23b shows that although the lower resolution had to access the cache more times it had a lower cache miss rate. It is also noted that it was expected for the lower resolution to have fewer cache access since fewer reads and writes are needed to work with a smaller resolution.

CHange title: Discussion of image resolution result

E.3.3 Closing remarks

Considering the validation criteria, an image resolution of 456×714 is the only resolution that partially fulfills the accuracy criteria as two of the four sequences tested had an ATE larger than 0.5 m. An image resolution of 456×714 also does not meet the real-time constraints of having an average tracking time of less than 50 ms; however, there is an increase in speed



(a) **Total times the cache was accessed** ***(b)*** **Overall cache miss rate**

Figure 23: **Bar graphs comparing the cache performance for different image resolutions**

with a factor of 1.08. The image resolution of 456×714 does well in meeting the robustness criteria.

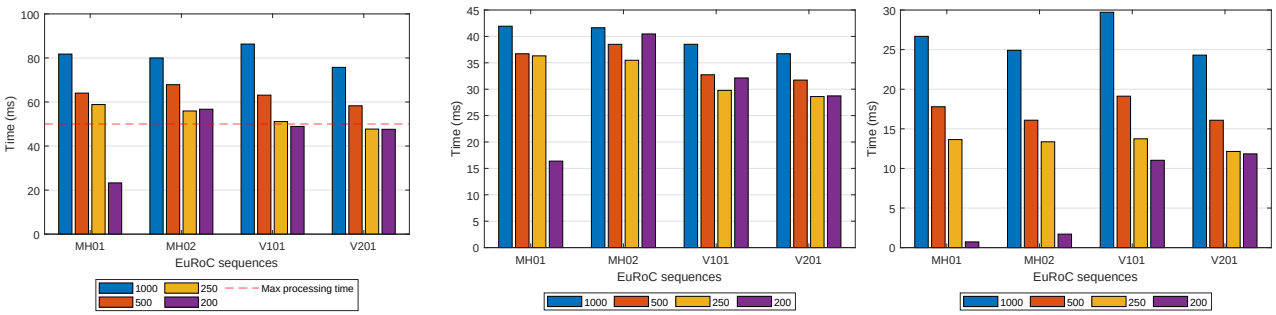
E.4 Number of features

In this section, the amount of features to extract per frame is reduced. The Tables that were used to create the graphs in this section are provided in Appendix G. In this section, a 1000 features serve as a baseline as it is the default amount of extracted features for ORB-SLAM3.

E.4.1 Execution statistics

Figure 24 shows the average total tracking, ORB extraction, and local mapping time, respectively. From Figure 24a to 24c it is clear that there is a downward trend in average time for total tracking, ORB extraction, and local mapping due to the decrease in the maximum allowed features to extract per frame. It should be noted that in Figure 24c the average time for MH01 and MH02 is swift for 200 features. This could be an indicator that the accuracy of the map will be poor as it is significantly faster than that of V101 and V201. From Figure 24a it is also clear that the real-time constraints are met for V101 and V201 for 250 and 200 features.

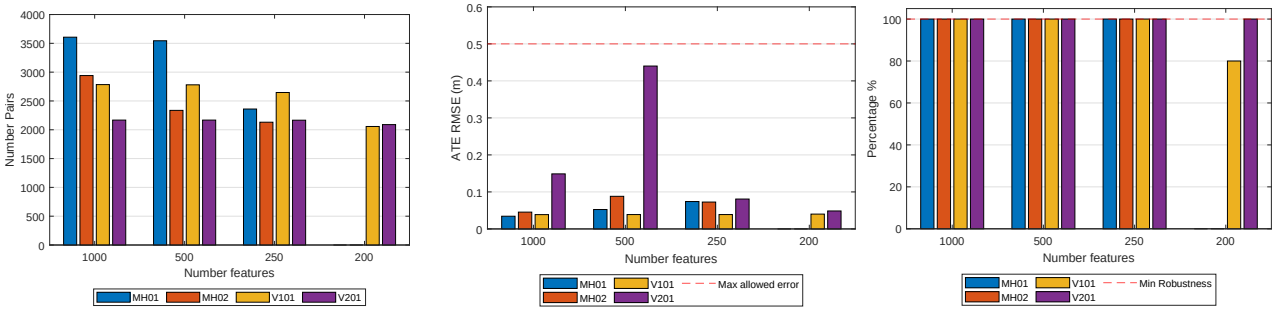
Figure 25 shows the accuracy and robustness of ORB-SLAM3 for a reduced number of features. Figure 25a shows that for 200 features, MH01 and MH02 have no pairs that match; thus, no maps would have been able to be created as evidence in Figure 25c. For 500 and 250 features, V101 and V201 have a similar amount of pairs that are matched compared to that of the baseline. Figure 25b shows that the ATE for all sequences and number of features are below



(a) **Total tracking time** *(b)* **ORB extraction time** *(c)* **Local mapping time**

Figure 24: **Bar graphs comparing the average execution time for different ORB-SLAM3 functions for different image resolutions on four EuRoC MAV sequences**

the criteria of 0.5 m. From Figure 25c it is clear that the robustness criteria are not met for only 200 features.



(a) **Pairs matched** *(b)* **ATE RMSE** *(c)* **Robustness**

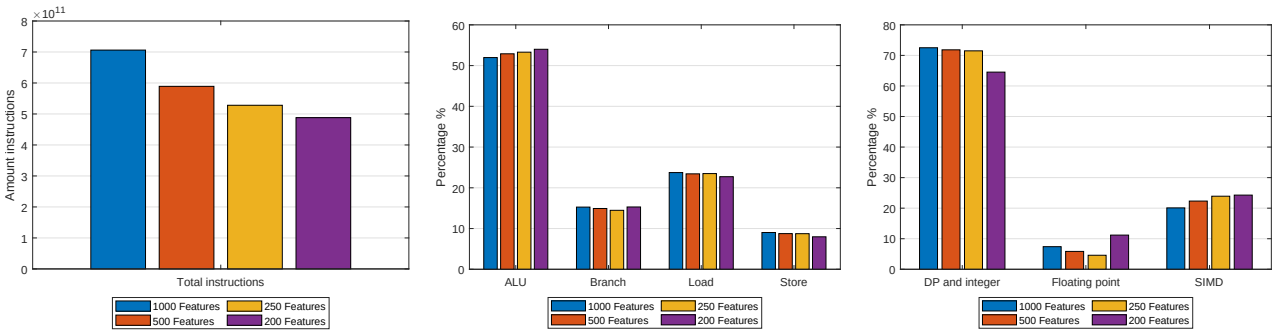
Figure 25: **Bar graph comparing the pairs matched, ATE, and robustness achieved during execution of ORB-SLAM3 for different image resolutions on four EuRoC MAV sequences**

E.4.2 Execution profiling

This section will analyze the results obtained by profiling ORB-SLAM3 for the number of features. The results of this section are provided as Tables in Appendix G.2.

Figure 26 shows the workload characteristics for ORB-SLAM3. Figure 26 indicates that there is a downward trend for the number of instructions executed as the number of features are decreased. Figures 26b and 26c indicate how the instruction mix and the ALU mix change as the features are decreased.

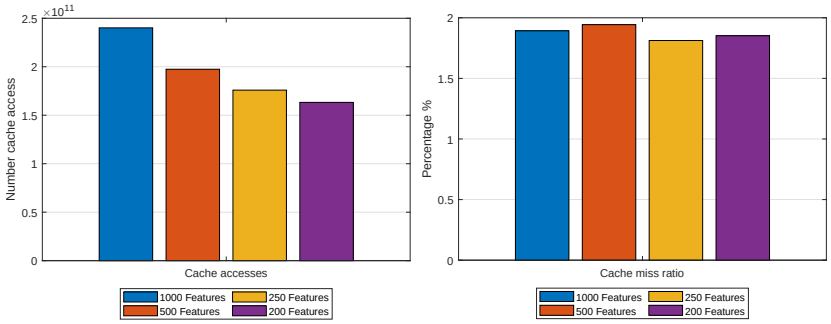
Figure 27 shows the cache performance of ORB-SLAM3 for the changed amount of features



(a) **Total instructions executed** *(b)* **Percentage instruction mix** *(c)* **Percentage ALU mix**

Figure 26: **Bar graphs comparing the workload characteristics for changes in the maximum allowed features to extract**

to extract. As expected, the total times the cache was access also has a downward trend as the number of features are reduced, as seen in Figure 27a. Using 250 features also has the lowest cache miss ratio as illustrated in Figure 27b. It is thus clear that using 250 features does provide the best cache performance as it has the lowest cache access and miss rate.



(a) **Total times the cache was accessed** *(b)* **Overall cache miss rate**

Figure 27: **Bar graphs comparing the cache performance for changes in the maximum allowed features to extract**

E.4.3 Closing remarks

Setting the number of features to be extracted to 250, all the validation criteria are met for sequences V101 and V201. Unfortunately, the average total tracking time of MH01 and MH02 does not meet the real-time constraints of 50 ms. The average of the total tracking time over

the four sequences is 53.4 ms which is close to the real-time constraints. Changing the number of features to 250 has gained a speedup of 1.52 against the baseline.

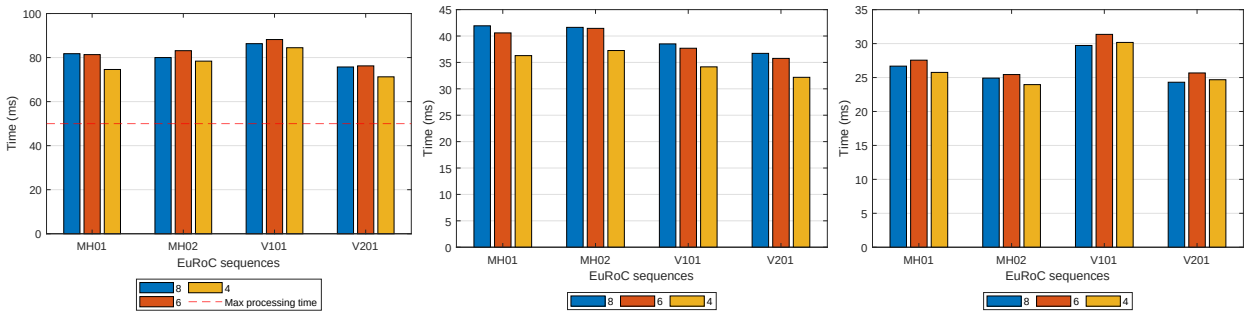
Reducing the number of features does provide a performance increase for ORB-SLAM3. It seems that the more the number of features is decreased, the faster ORB-SLAM3 can execute—this coincides with what was mentioned in Section 3.3. However, there exist a number of features at which ORB-SLAM3 can no longer reliability create maps.

E.5 Number pyramid levels

In this section, the number of pyramid levels are reduced. The tables that were used to create the graphs in this section are provided in Appendix H. In this section, eight levels serve as a baseline as it is the default setting for ORB-SLAM3.

E.5.1 Execution statistics

Figure 28 illustrates the time performance that ORB-SLAM3 achieves for different pyramid levels. Figure 28b shows that there is a decrease in ORB extraction time as the number of pyramid levels are decreased. However for there is only a slight decrease in local mapping time for four levels and an increase for six levels as shown in Figure 28c. Figure 28a illustrates that for four pyramid levels, there was a decrease in total tracking time but an increase for six pyramid levels. This could be caused by the rise in local mapping time for six pyramid levels.

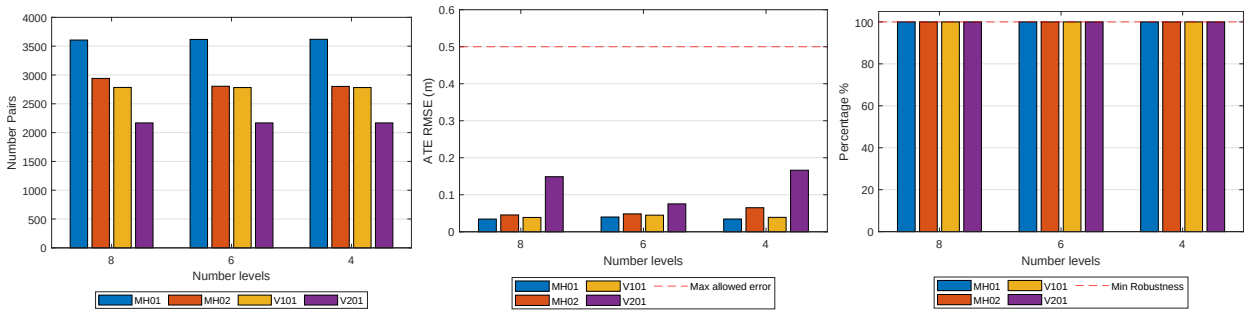


(a) **Total tracking time** *(b)* **ORB extraction time** *(c)* **Local map time**

Figure 28: **Bar graphs comparing the average execution time for different ORB-SLAM3 functions for different pyramid levels on four EuRoC MAV sequences**

Figure 29 shows the accuracy and robustness that ORB-SLAM3 achieves for different pyramid levels. Changing the number of pyramid levels did not influence the number of pairs matched between the estimated trajectory and ground-truth trajectory for any of the levels, as shown

in Figure 29a. The ATE achieved by ORB-SLAM3 for all pyramid levels is below the accuracy criteria as illustrated in Figure 29b. Figure 29c shows that changing the levels did not alter the robustness of ORB-SLAM3.

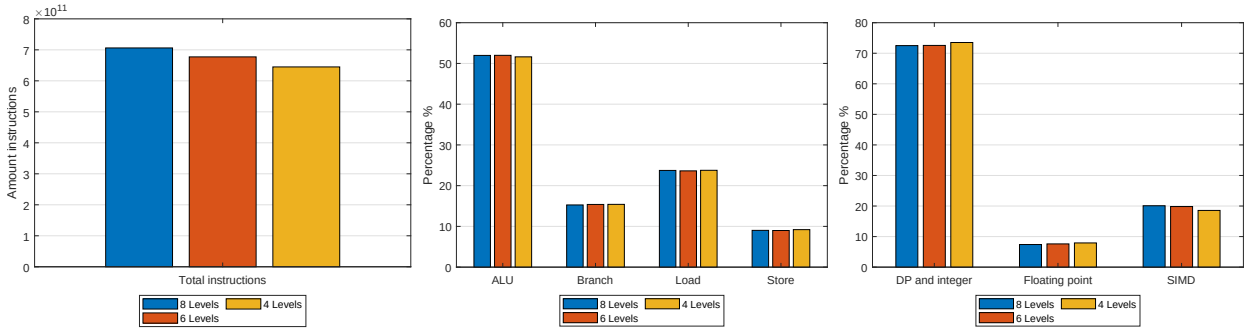


(a) **Pairs matched** *(b)* **ATE RMSE** *(c)* **Robustness**

Figure 29: **Bar graph comparing the pairs matched, ATE, and robustness achieved during execution of ORB-SLAM3 for different pyramid levels on four EuRoC MAV sequences**

E.5.2 Execution profiling

Figure 30 shows the workload characteristics that were measured during the execution of ORB-SLAM3 for certain pyramid levels. Figure 30 shows that the total instructions executed decrease as the number of levels are decreased. The instruction mix and ALU mix stays similar for the changes in pyramid levels as illustrated in Figures 30b and 30c.

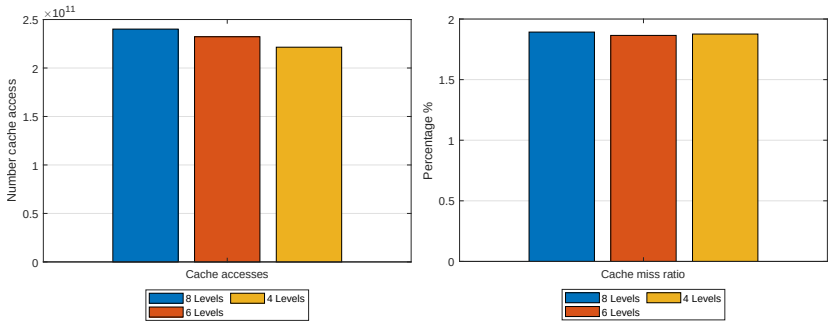


(a) **Total instructions executed** *(b)* **Percentage instruction mix** *(c)* **Percentage ALU mix**

Figure 30: **Bar graphs comparing the workload characteristics for different pyramid levels**

Figure 31 shows the cache performance of ORB-SLAM3 for certain pyramid levels. Figure

31a indicates that there is a decrease in cache accesses, and Figure 31b illustrates that the cache miss rate stays similar for all pyramid levels.



(a) **Total times the cache was accessed** ***(b)*** **Overall cache miss rate**

Figure 31: **Bar graphs comparing the cache performance for different pyramid levels**

E.5.3 Closing remarks

It is clear that decreasing the pyramid levels does provide a decrease in ORB extraction time but only has a speedup of 1.04 for four pyramid levels. This speedup is not substantial to meet the real-time constraints that are needed. The alteration in pyramid levels, however, does not increase the ATE or the robustness of ORB-SLAM3.

E.6 Combined changes

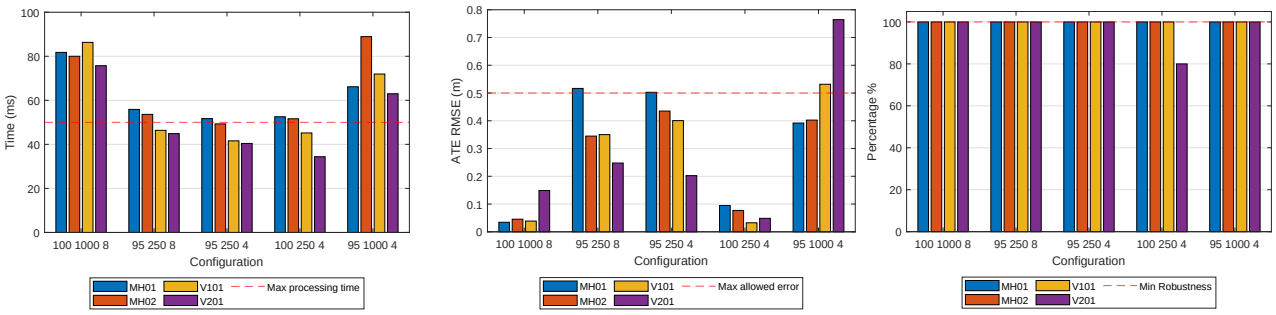
In this section, the best-performing parameters will be tested in combination. The four combinations that will be tested along with the baseline are listed in Table 34. The first configuration will test the original image resolution with the 250 features and four pyramid levels as those two parameters provided the best results for their respective changes. The second configuration tests the resolution and level change on the default number of features of 1000. The third configuration tests the resolution change and the number of features change on the default number of levels of 8. The final configuration tests all three parameter changes in a single test. The best-performing configuration will also be tested on the rest of the EuRoC MAV sequences to verify that it does work for different sequences and scenarios.

Table 34: **Parameter combination configurations**

Configuration Name	Resolution	Features	Levels
100_250_4	480x752	250	4
95_1000_4	456x714	1000	4
95_250_8	456x714	250	8
95_250_4	456x714	250	4

E.6.1 Execution statistics

Figure 32 illustrates the execution performance of ORB-SLAM3 for different ORB-SLAM3 configurations. Figure 32a to 32c shows the execution performance for the different ORB-SLAM3 configurations provided in Table 34 on four EuRoC MAV sequences. From Figure 32a it is clear that configurations 95_250_4 and 100_250_4 both almost satisfy the performance criteria with sequences MH01 and MH02 slightly above the cut-off value of 50 ms. All other configurations have a larger average for the total tracking time that is required by the criteria. Figure 32b shows that most configurations do meet the accuracy requirements except for the MH01 sequence for 95_250_8 configuration and the V101 and V201 sequences for the 95_100_4 configuration. It is apparent from this figure that reducing the image resolution will increase the ATE error. Figure 32c shows that only sequence V201 for the 100_250_4 configuration failed to complete a map for every execution.



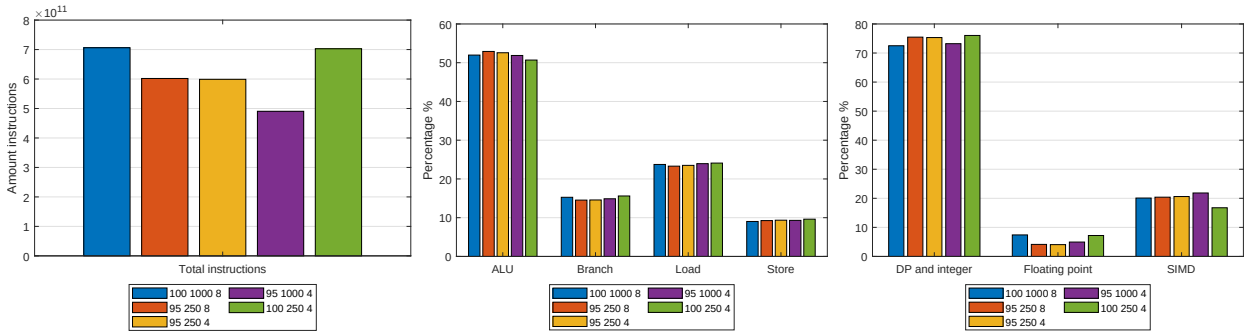
(a) **Total tracking time** *(b)* **ATE** *(c)* **Robustness**

Figure 32: **Bar graphs comparing the execution performance of different ORB-SLAM3 configurations on four EuRoC MAV sequences**

Considering Figures 32a to 32c, configurations 95_250_4 and 100_250_4 provides the results closest the meeting the criteria listed in Table **??**. Configuration 95_250_4 provides fast and robust results at the cost of some accuracy, whereas the 100_250_4 configuration provides fast and accurate results at the cost of robustness.

E.6.2 Execution profiling

In this section, the configurations are profiled to see how they differ in execution. Figure 33 shows the workload characteristic for the different ORB-SLAM3 configurations. Figure 33a shows an interesting results as the amount of instructions executed for configurations 100_1000_8 and 100_250_4, although configuration 100_250_4 executes 1.76 times faster. This suggests that the CPU hardware is better utilized for this configurations. Figures 33b and 33c shows how the the instructions are divided among the total instructions.



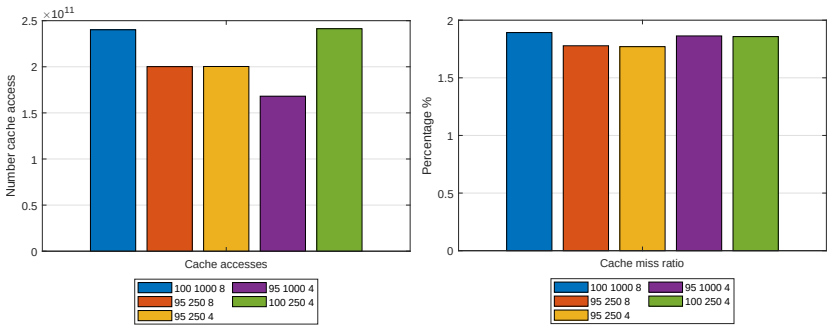
(a) **Total instructions executed** *(b)* **Percentage instruction mix** *(c)* **Percentage ALU mix**

Figure 33: **Bar graphs comparing the workload characteristics for different parameter combinations**

Figure 34shows the cache performance that was achieved by the different combinations of parameter changes. As with Figure 33a it is noted in Figure 34a that configuration 100_250_4 has a similar amount of cache access than that of configuration 100_1000_8, although being faster. All of the configurations have a similar cache miss rate as illustrated in FIgure 34b.

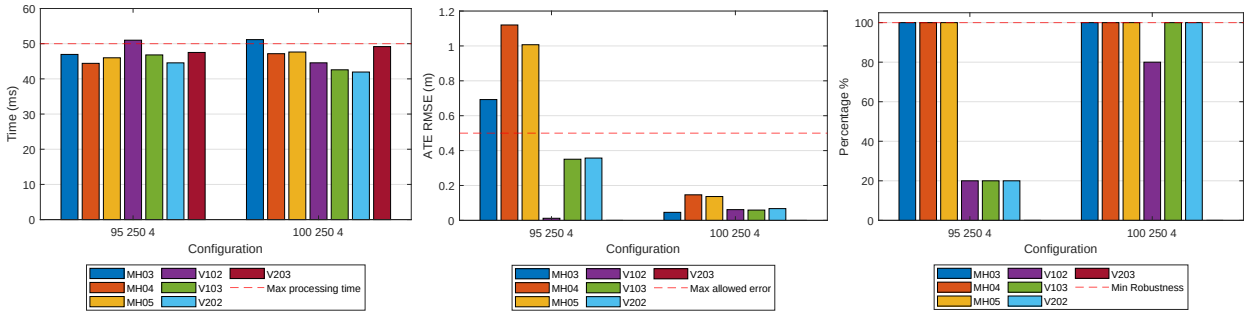
E.6.3 Configuration performance on more sequences

Figure 35 shows the performance that was gained by using the different configurations on the rest of the EuRoC MAV dataset. Figure 35a clearly shows that almost all sequences of both configurations have an average total tracking time lower than the required time. However, Figure 35b illustrates that for the 95_250_4 configuration, three out of the seven sequences have an ATE larger than the required value. Configuration 95_250_4 also has very low robustness as seen in Figure 35c. Figures 35b and 35c indicate that configuration 100_250_4 has a very low ATE and relatively good robustness with failing only two of the seven sequences. It is noted from Figure 35b and 35c that both configurations could not create any map for sequence V203, one of EuRoC MAV's difficult sequences.



(a) Total times the cache was accessed *(b)* Overall cache miss rate

Figure 34: **Bar graphs comparing the cache performance for different parameter combinations**



(a) Total tracking time *(b)* ATE *(c)* Robustness

Figure 35: **Bar graphs comparing the execution performance of different ORB-SLAM3 configurations on the rest of the EuRoC MAV sequences**

E.7 ORB-SLAM3 performance on the Denver 2 CPU

Throughout the study we focused on implementing and testing ORB-SLAM3 on the A57 CPU cluster and not on the Nvidia Denver 2 CPU cluster. In this section we investigate and compare the performance differences between the two CPU clusters for three configurations of ORB-SLAM3 using the V201 input sequence.

E.8 Conclusion

This Appendix looked into the results that were obtained during the experiment mentioned in Section D. It was clear that by reducing the number of features to extract per frame, the most significant decrease in total tracking time was achieved with the slightest change in accuracy.

Changing the image resolution did improve the tracking time, but the accuracy degraded. Decreasing the pyramid levels had a slight decrease in total tracking time while having a similar accuracy.

Combining the individual parameter changes two configurations provided promising results on the test sequences, achieving the required performance, accuracy, and robustness. Upon executing the two ORB-SLAM3 configurations on more complex sequences of the EuRoC MAV dataset, it was evident that the configuration of original image resolution, 250 features, and four pyramid levels provided the most consistent results with the best performance.

F Resolution

This Appendix contains the results that were obtained for the resolution change experiment. The resolution that were investigated are listed in Table 30 in Section D.3.1. The results and the code used to obtain the results are provided online and can be found here.

F.1 ORB-SLAM3 performance

These Tables represents the execution performance of ORB-SLAM3 during execution for the different image resolutions.

Table 35: **Execution performance for image resolution:** 240×376

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	0.000	0.000	0.000	0.000	0.000	0%
MH02	0.000	0.000	0.000	0.000	0.000	0%
V101	0.000	0.000	0.000	0.000	0.000	0%
V201	0.000	0.000	0.000	0.000	0.000	0%

Table 36: **Execution performance for image resolution:** 288×451

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	96.58	0.026	16.50	28.82	13.95	40%
MH02	88.54	0.010	23.00	27.70	13.67	40%
V101	61.17	0.000	0.00	17.56	11.69	0%
V201	55.56	0.000	0.00	16.83	11.57	0%

Table 37: **Execution performance for image resolution:** 336×526

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH02	78.10	153.78	2172.33	30.64	14.5923	60%
V101	64.85	0.000	0.00	20.91	12.3389	0%
V201	57.85	0.000	0.00	20.07	12.4634	0%

Table 38: **Execution performance for image resolution:** 384×601

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	78.46	1236.6	2187.60	35.29	12.9709	100%
MH02	73.29	515.47	1909.40	34.02	13.8586	100%
V101	78.18	0.005	22.00	29.56	15.4409	20%
V201	73.49	0.000	0.00	28.57	16.0888	0%

Table 39: **Execution performance for image resolution:** 432×676

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	76.39	1.379	2378.00	37.21	21.5062	100%
MH02	77.04	88.494	2095.00	38.30	17.4716	100%
V101	65.26	2.146	2781.00	32.39	18.4126	100%
V201	62.64	1.427	2166.00	30.82	19.5021	100%

Table 40: **Execution performance for image resolution:** 456×714

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	75.33	0.436	3628.00	37.56	24.8009	100%
MH02	76.00	83.710	2649.80	38.94	20.5863	100%
V101	77.98	0.284	2780.00	34.52	25.9722	100%
V201	67.89	0.999	2167.00	32.77	21.6426	100%

Table 41: **Execution performance for image resolution:** 480×752

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	81.76	0.034	3606.80	41.93	26.6799	100%
MH02	80.00	0.045	2940.80	41.64	24.9166	100%
V101	86.31	0.039	2784.00	38.51	29.7244	100%
V201	75.73	0.149	2168.00	36.71	24.3081	100%

F.2 ORB-SLAM3 Profiling

The Tables in this section provides the performance metrics that was measured during execution of ORB-SLAM3 with different image resolutions.

Table 42: **Experiment 2 - Resolution change instruction mix**

Configuration	Instructions executed	Percentage Load/Store [%]	Percentage ALU [%]	Percentage branch [%]
480x752	7.06E+11	32.76	51.97	15.27
456x714	7.56E+11	32.81	51.85	15.34

Table 43: **Experiment 2 - Resolution change ALU mix**

Configuration	Percentage data processing/inte [%]	Percentage floating point [%]	Percentage SIMD [%]
480x752	72.50	7.40	20.09
456x714	75.20	6.78	18.02

Table 44: **Experiment 2 - Resolution change cache performance**

Configuration	Cache accesses	Cache miss rate [%]
480x752	2.40E+11	1.89
456x714	2.55E+11	1.79

G Number Features

This Appendix contains the results that were obtained for the number of features change experiment. The results and the code used to obtain the results are provided online and can be found here.

G.1 ORB-SLAM3 performance

These Tables represents the execution performance of ORB-SLAM3 during execution for the number of features.

Table 45: **Number features 1000**

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	81.76	0.034	3606.80	41.93	26.68	100.00%
MH02	80.00	0.045	2940.80	41.64	24.92	100.00%
V101	86.31	0.039	2784.00	38.51	29.72	100.00%
V201	75.73	0.149	2168.00	36.71	24.31	100.00%

Table 46: **Number features 500**

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	64.04	0.052	3544.60	36.71	17.79	100.00%
MH02	67.86	0.088	2336.60	38.50	16.09	100.00%
V101	63.12	0.039	2780.00	32.73	19.13	100.00%
V201	58.28	0.440	2168.00	31.73	16.09	100.00%

G.2 ORB-SLAM3 Profiling

The Tables in this section provides the performance metrics that was measured during execution of ORB-SLAM3 with different number of features.

Table 47: Number features 250

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	58.84	0.074	2360.00	36.31	13.64	100.00%
MH02	55.93	0.073	2131.20	35.48	13.36	100.00%
V101	51.11	0.039	2646.80	29.78	13.74	100.00%
V201	47.71	0.081	2166.00	28.61	12.15	100.00%

Table 48: Number features 200

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	23.27	0.000	0.00	16.38	0.72	0.00%
MH02	56.72	0.000	0.00	40.46	1.71	0.00%
V101	48.92	0.040	2057.00	32.13	11.03	80.00%
V201	47.60	0.049	2089.60	28.73	11.84	100.00%

Table 49: Experiment 2 - Number of features instruction mix

Configuration	Instructions executed	Percentage Load/Store [%]	Percentage ALU [%]	Percentage branch [%]
1000 Features	7.06E+11	32.76	51.97	15.27
500 Features	5.89E+11	32.18	52.89	14.92
250 Features	5.28E+11	32.23	53.29	14.48
200 Features	4.88E+11	30.70	54.00	15.30

Table 50: Experiment 2 - Number of features ALU mix

Configuration	Percentage data processing/inte [%]	Percentage floating point [%]	Percentage SIMD [%]
1000 Features	72.50	7.40	20.09
500 Features	71.82	5.85	22.33
250 Features	71.49	4.59	23.92
200 Features	64.53	11.19	24.28

Table 51: **Experiment 2 - Number of features cache performance**

Configuration	Cache accesses	Cache miss rate [%]
1000 Features	2.40E+11	1.89
500 Features	1.97E+11	1.94
250 Features	1.76E+11	1.81
200 Features	1.63E+11	1.85

H Number Levels

This Appendix contains the results that were obtained for the number of pyramid levels experiment. The results and the code used to obtain the results are provided online and can be found here.

H.1 ORB-SLAM3 performance

These Tables represents the execution performance of ORB-SLAM3 during execution for the number of pyramid levels.

Table 52: **Pyrimid levels - 8**

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	81.76	0.034	3606.80	41.93	26.68	100.00%
MH02	80.00	0.045	2940.80	41.64	24.92	100.00%
V101	86.31	0.039	2784.00	38.51	29.72	100.00%
V201	75.73	0.149	2168.00	36.71	24.31	100.00%

Table 53: **Pyrimid levels - 6**

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	81.34	0.040	3616.80	40.59	27.56	100.00%
MH02	83.13	0.048	2804.80	41.45	25.44	100.00%
V101	88.19	0.045	2782.00	37.69	31.36	100.00%
V201	76.22	0.075	2168.00	35.76	25.67	100.00%

H.2 ORB-SLAM3 Profiling

The Tables in this section provides the performance metrics that was measured during execution of ORB-SLAM3 with different number of pyramid levels.

Table 54: Pyrimid levels - 4

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	74.59	0.034	3619.00	36.29	25.76	100.00%
MH02	78.38	0.065	2801.60	37.25	23.96	100.00%
V101	84.47	0.039	2782.00	34.16	30.17	100.00%
V201	71.26	0.166	2168.00	32.18	24.68	100.00%

Table 55: Experiment 2 - Number of pyramid levels instruction mix

Configuration	Instructions executed	Percentage Load/Store [%]	Percentage ALU [%]	Percentage branch [%]
8 Levels	7.06E+11	32.76	51.97	15.27
6 Levels	6.77E+11	32.62	51.99	15.39
4 Levels	6.45E+11	32.97	51.62	15.41

Table 56: Experiment 2 - Number of pyramid levels ALU mix

Configuration	Percentage data processing/inte [%]	Percentage floating point [%]	Percentage SIMD [%]
8 Levels	72.50	7.40	20.09
6 Levels	72.56	7.60	19.83
4 Levels	73.52	7.92	18.56

Table 57: Experiment 2 - Number of pyramid levels cache performance

Configuration	Cache accesses	Cache miss rate [%]
8 Levels	2.40E+11	1.89
6 Levels	2.32E+11	1.86
4 Levels	2.21E+11	1.88

I Combination

This Appendix contains the results that were obtained for the combination in parameter changes. The results and the code used to obtain the results are provided online and can be found here.

I.1 ORB-SLAM3 performance

These Tables represents the execution performance of ORB-SLAM3 during execution for different ORB-SLAM3 configurations.

Table 58: **Combination of parameters 100_1000_8**

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	81.49	0.040	3626.60	41.73	26.63	100.00%
MH02	80.43	0.036	2935.20	41.75	24.94	100.00%
V101	86.34	0.039	2784.00	38.50	29.56	100.00%
V201	75.56	0.184	2168.00	36.66	24.07	100.00%

Table 59: **Combination of parameters 100_350_6**

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	56.63	0.099	3360.80	33.16	14.47	100.00%
MH02	47.98	0.102	2175.75	27.82	10.73	80.00%
V101	53.57	0.037	2781.00	29.00	15.66	100.00%
V201	49.77	0.301	2167.00	28.10	13.32	100.00%

I.2 ORB-SLAM3 Profiling

The Tables in this section provides the performance metrics that was measured during execution of ORB-SLAM3 during the combination of parameter changes.

Table 60: Combination of parameters 95_250_8

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	55.87	0.517	2500.80	32.50	13.182	100.00%
MH02	53.62	0.345	2130.80	32.29	13.1212	100.00%
V101	46.37	0.350	2184.80	26.56	11.2821	100.00%
V201	44.89	0.248	2163.00	25.19	11.5921	100.00%

Table 61: Combination of parameters 95_250_4

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH01	51.70	0.503	2623.60	26.15	11.80	100.00%
MH02	49.29	0.435	2143.20	25.88	11.43	100.00%
V101	41.59	0.401	2151.60	21.74	10.15	100.00%
V201	40.42	0.202	2124.40	20.74	10.98	100.00%

Table 62: Combination of parameters for all sequences 100_1000_8

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH03	89.23	0.040	2263.40	44.29	21.99	100.00%
MH04	83.34	0.075	1582.80	40.11	20.86	100.00%
MH05	83.77	0.063	1814.20	40.60	21.21	100.00%
V102	75.78	0.037	1602.00	37.11	20.66	100.00%
V103	72.19	0.065	1897.60	36.60	19.93	100.00%
V202	74.88	0.041	2272.00	36.28	23.41	100.00%
V203	71.88	0.055	1670.80	36.13	18.57	100.00%

Table 63: Combination of parameters for all sequences 100_350_6

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH03	58.25	0.059	2219.80	33.32	13.49	100.00%
MH04	53.89	0.123	1569.20	29.86	12.47	100.00%
MH05	52.85	0.078	1788.40	30.11	11.87	100.00%
V102	49.72	0.022	1543.60	27.72	12.69	100.00%
V103	45.67	0.051	1960.00	27.10	11.36	100.00%
V202	48.55	0.030	2250.00	27.50	13.17	100.00%
V203	46.45	0.073	1796.00	26.90	11.22	100.00%

Table 64: Combination of parameters for all sequences 95_250_4

Config	Total Tracking Time [ms]	ATE RMSE [m]	Pairs Matched	ORB Extraction [ms]	Local Map Track [ms]	Created Maps
MH03	46.95	0.693	1993.60	24.78	10.02	100.00%
MH04	44.41	1.120	1557.20	22.02	9.42	100.00%
MH05	45.99	1.007	1659.80	22.67	9.49	100.00%
V102	50.99	0.012	3.00	23.88	4.92	20.00%
V103	46.81	0.351	1583.00	23.11	5.90	20.00%
V202	44.55	0.357	2252.00	22.07	7.11	20.00%
V203	47.51	0.000	0.00	23.15	4.63	0.00%

Table 65: Experiment 2 - Parameter combinations instruction mix

Configuration	Instructions executed	Percentage Load/Store [%]	Percentage ALU [%]	Percentage branch [%]
100-1000-8	7.06E+11	32.76	51.97	15.27
95-250-8	6.02E+11	32.55	52.91	14.54
95-250-4	5.99E+11	32.86	52.57	14.57
100-250-4	4.91E+11	33.25	51.87	14.88
95-1000-4	7.03E+11	33.71	50.69	15.60

Table 66: **Experiment 2 - Parameter combinations ALU mix**

Configuration	Percentage data processing/inte [%]	Percentage floating point [%]	Percentage SIMD [%]
100-1000-8	72.50	7.40	20.09
95-250-8	75.47	4.14	20.39
95-250-4	75.32	4.06	20.61
100-250-4	73.22	4.95	21.84
95-1000-4	76.05	7.20	16.75

Table 67: **Experiment 2 - Parameter combinations cache performance**

Configuration	Cache accesses	Cache miss rate [%]
100-1000-8	2.40E+11	1.89
95-250-8	2.00E+11	1.78
95-250-4	2.00E+11	1.77
100-250-4	1.68E+11	1.86
95-1000-4	2.41E+11	1.86