République Algérienne Démocratique et Populaire
Ministère de l'enseignement supérieur et de la recherche scientifique
Ecole Nationale Polytechnique

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Department of Electronics
Signal & communication Laboratory

مخبر الإشارة و الإتصالات
Signal & Communications Lab.

Doctoral thesis in Electronics

Presented by

Mr. Salah BOUHOUN

Entitled

Prototyping Platform for Embedded SLAM

Defended publicly on Wednesday, November 24th, 2021
in front of the jury composed of:

President:	Mr. Adel BELOUCHRANI	Professor, ENP
Supervisors :	Mr. Rabah SADOUN	Professor, ENP
	Mr. Mourad ADNANE	Professor, ENP
Examiners:	Mr. Cherif LARBES	Professor, ENP
	Mrs. Nouara ACHOUR	Professor, USTHB
	Mr. Abdelkrim NEMRA	MC A, EMP

ENP 2021

Ecole Nationale Polytechnique
10, Avenue Hassan BADI – El-Harrach 16200
Algiers, Algeria

République Algérienne Démocratique et Populaire
Ministère de l'enseignement supérieur et de la recherche scientifique
Ecole Nationale Polytechnique

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Department of Electronics
Signal & communication Laboratory

مخبر الإشارة و الإتصالات
Signal & Communications Lab.

Doctoral thesis in Electronics

Presented by

Mr. Salah BOUHOUN

Entitled

Prototyping Platform for Embedded SLAM

Defended publicly on Wednesday, November 24th, 2021
in front of the jury composed of:

President:	Mr. Adel BELOUCHRANI	Professor, ENP
Supervisors :	Mr. Rabah SADOUN	Professor, ENP
	Mr. Mourad ADNANE	Professor, ENP
Examiners:	Mr. Cherif LARBES	Professor, ENP
	Mrs. Nouara ACHOUR	Professor, USTHB
	Mr. Abdelkrim NEMRA	MC A, EMP

ENP 2021

Ecole Nationale Polytechnique
10, Avenue Hassan BADI – El-Harrach 16200
Algiers, Algeria

République Algérienne Démocratique et Populaire
Ministère de l'enseignement supérieur et de la recherche scientifique
École Nationale Polytechnique

Département d'électronique
Laboratoire Signal et communication

Thèse de doctorat en Électronique

Présenté par:

M. Salah BOUHOUN

Thème

Plateforme de prototypage pour SLAM embraqué

Soutenu publiquement le Mercredi 24 Novembre 2021
devant le jury composé de:

Président:	M. Adel BELOUCHRANI	Professeur, ENP
Directeurs de thèse:	M. Rabah SADOUN	Professeur, ENP
	M. Mourad ADNANE	Professeur, ENP
Examinateurs:	M. Cherif LARBES	Professeur, ENP
	Mme. Nouara ACHOUR	Professeur, USTHB
	M. Abdelkrim NEMRA	MC A, EMP

ENP 2021

École Nationale Polytechnique
10, Avenue Hassan BADI – El-Harrach 16200

الملخص:

التموضع ورسم الخرائط مهمتين أساسيتين للمركبات ذاتية القيادة. ركزت العديد من المشاريع البحثية على تصميم أنظمة مضمنة لتنفيذ خوارزميات التموضع ورسم الخرائط المتزامنين(SLAM) . في هذا العمل ، نقترح منصة للنماذج الأولية العامة مخصصًا لـ SLAM ، مستندين إلى مقاربة "Meet-in-the-middle". تم استخدام نموذجين رئيسيين لهذا الغرض: SoC والمصادر المفتوحة. تتكون الأرضية من شقين: الشق الجهازي والشق البرمجي. تم تصميم الشق الجهازي حول SoC-FPGA ذو تكلفة منخفضة، تدمج وحدة معالجة مركزية CPU و FPGA في نفس الشريحة. تم تصميم دائرة التسريع على FPGA باستخدام لغة عالية المستوى OpenCL. يقلل استخدام هذه اللغة من وقت التطوير ويوفر إمكانية النقل إلى منصات التسريع الجهازي الأخرى. يعتمد الشق البرمجي على نظام Linux والمكونات مفتوحة المصدر. تُستخدم أداة أتمتة البناء لبناء وإنشاء نظام Linux المضمن. تتيح هذه الأداة تقليل وقت التطوير بالإضافة إلى إمكانية إعادة الاستهداف باستخدام المترجمات المتقاطعة. في النهاية ، أجرينا دراسة حالة حول SLAM المرئي. تم توطين SLAM، مصمم مسبقا، على منصتنا. تم استخدام منهجية التصميم المشترك للبرامج / الأجهزة. تتيح هذه المنهجية تحديد وظيفة SLAM التي سيتم تنفيذها على أي شق من المنصة (برنامج أو جهاز)، استنادًا إلى التحليل الزماني والأدائي لـ SLAM. استخدمت تقنيات تحسين نواة OpenCL للزيادة في الأداء. تمكنا من تشغيل SLAM في الوقت الفعلي (57.14Hz) ، مع تسريع بنحو 1.93 ضعف التصميم البرامجي.

الكلمات المفتاحية: SLAM، SoC-FPGA، تصميم مشترك برامج / أجهزة، OpenCL، منصة نمذجة، لينكس مضمن، Buildroot

Résumé

La localisation et la cartographie sont des taches essentielles pour les véhicules autonomes. Plusieurs travaux de recherche avaient comme objectif la conception des systèmes embarqués exécutant les algorithmes de localisation et de cartographie simultanées(SLAM). Dans ce travail, on propose une plateforme de prototypage générique dédiée au SLAM en se basant sur une approche "Meet-in-the-middle". Pour cela deux paradigmes clés ont utilisés : les SoC et l'open source. La plateforme comporte deux partie : partie matérielle et partie logicielle. La partie matérielle est conçue autour d'un SoC-FPGA de faible coût intégrant un CPU et un FPGA dans une même puce. Le circuit accélérateur sur le FPGA est conçu en utilisant un langage haut niveau : OpenCL. L'utilisation de ce langage permet de réduire le temps de développement et offre la possibilité de portage vers d'autres plateformes d'accélération matérielle. La partie Logicielle est basé sur le système Linux est des briques open-source. Un outil d'automatisation de construction est utilisé pour construire et générer le système Linux embarqué. Un tel outil permet une réduction du temps de développement ainsi qu'un reciblage en utilisant des compilateurs croisés. A la fin, une étude de cas sur un SLAM visuel a été réalisée. Une implémentation existante du SLAM a été implémentée sur notre plateforme. Une méthodologie de conception conjointe (co-design) Logicielle/matérielle est utilisée. Cette méthodologie permet de décider quelle fonction du SLAM qui va être exécuté sur quelle partie (Logicielle ou matérielle) de la plateforme en se basant sur un profiling et une analyse du code SLAM. Des techniques d'optimisation du noyau OpenCL ont été utilisées pour augmenter la performance. Le fonctionnement du SLAM étais en temps réel (57.14Hz), avec une accélération de 1.93 fois l'implémentation logicielle.

Mots clés : SLAM, SoC-FPGA, Co-conception logicielle/Matérielle, OpenCL, Plateforme de prototypage, Linux embarqué, Buildroot.

Abstract :

Localization and mapping are essential tasks for autonomous vehicles. Several research projects focused on the design of embedded systems executing simultaneous localization and mapping algorithms (SLAM). In this work, we propose a generic prototyping platform dedicated to SLAM, based on a "Meet-in-the-middle" approach. Two key paradigms have been used for this: SoCs and open-source. The platform has two parts: hardware part and software part. The hardware part is designed around a low cost SoC-FPGA integrating a CPU and an FPGA in the same chip. The accelerator circuit on the FPGA is designed using a high level language: OpenCL. The use of this language reduces development time and offers the possibility of porting to other hardware acceleration platforms. The Software part is based on the Linux system and open-source bricks. A build automation tool is used to build and generate the embedded Linux system. Such a tool allows a reduction in development time as well as retargeting using cross compilers. At the end, a case study on a visual SLAM was carried out. An existing implementation of SLAM has been implemented on our platform. A software / hardware co-design methodology is used. This methodology makes it possible to decide which SLAM function will be executed on which part (software or hardware) of the platform, based on profiling and analysis of the SLAM code. OpenCL kernel optimization techniques were used to increase performance. The operation of the SLAM was in real time (57.14Hz), with an acceleration of 1.93 times the software implementation.

Keywords: SLAM, SoC-FPGA, Co-design Hardware/software, OpenCL, Prototyping platform, Embedded Linux, Buildroot

ACKNOWLEDGEMENT

This thesis has been conducted at the Signal & Communication Laboratory in the Department of Electronics of Ecole Nationale Polytechnique (ENP) of Algiers.

First of all, I thank **ALLAH** for giving me the strength and the ability to learn, understand, and accomplish my thesis.

I would like to take this opportunity to express my deepest gratitude and thanks to my project supervisors, Pr. **Rabah Sadoun** and Pr. **Mourad Adnane** for their constant guidance, assistance and support as well as all the knowledge they shared during this research project.

I also wish to thank President of jury Mr. **Adel Belouchrani**, Professor at ENP, and the members of the jury: Mr. **Cherif Larbes**, Professor at ENP, Ms. **Nouara Achour**, Professor at USTHB and Mr **Abdelkrim Nemra**, MC.A at EMP for accepting to be members of the reading committee and their constructive analysis of the present work.

Last but not least, I would like to express my appreciation and gratitude to my family members, who have encouraged, motivated and supported me during my studies.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

GENERAL INTRODUCTION

The design of autonomous robots has been a growing subject of research for many years. Today, interest in the development of these robots is increasing with the emergence of robotic systems widely used in many applications. Autonomous mobile robots are widely used to explore and interfere in inaccessible area to humans like distant worlds or dangerous situations. These robots must have the capacity to move safely and autonomously in environments that are unknown and which cannot be mapped in advance by humans.

An autonomous mobile robot must have information about its surrounding environment in order to move safely. It must answer two basic questions: Where am I and what does the surrounding environment look like? The entire process that enables a mobile robot to identify the surrounding environment and localize itself in this environment is known by Simultaneous Localization And Mapping (SLAM). Mapping allows the construction of a map representing the spatial structure of the environment from certain information gathered by the robot's sensors. Localization determines the robot's position on the map that corresponds to its position in the real environment.

Motivation

In the context of SLAM, many algorithms have been developed to solve this problem. These algorithms have complex calculations, and for a real-time execution, we need

high-performance and powerful machines. High performance machines have large dimensions and weight. Besides the high power consumption, these disadvantages prevent them from being used on mobile robots. Running SLAM on a fully autonomous mobile robot requires an implementation of SLAM in an embedded system with an efficient software / hardware architecture to maintain real-time processing.

In the literature, we find several research works which aim to implement different SLAM algorithms on embedded systems. These embedded systems (targets) have different characteristics and vary in size, technology, performance...etc in addition to the different methods of implementation of each target. The design of these embedded SLAM systems follows a top-down approach that make these system specific, and without the possibility to reuse it for other systems, and rise the cost of te time of the development. A generic prototyping platform that bases on hardware and software reuse can be used to reduce the time to market, development risks and system cost. These SLAM-dedicated embedded systems prototyping platforms must take into account the constraints associated with autonomous vehicles and SLAM algorithms , such as energy consumption, embeddability, processing power, algorithm complexity...etc. and supports multi-targeting. In the literature these prototyping platform for embedded SLAM are not very common.

A Magister project, by Ben Messaoud [1], proposed a generic prototyping platform, to help design and build embedded systems dedicated to SLAM. The core of the SoPC used in the platform is an FPGA from Altera. The hardware system is based on the soft processor NiosII, in which is ported an embedded operating system μCLinux. The design of the SLAM is based on a co-design methodology using an AAA approach (Algorithm-Architecture Adequacy). The platform is design to be adaptable to different mobile platform and can integrate different sensors.

SLAMBench [2–4]is a publicly available software framework which is starting point for quantitative, comparable, and validatable experimental research to investigate trade-offs in performance, accuracy and energy consumption of SLAM systems. It aims to unify the interface of SLAM algorithms to perform the benchmarking and the evaluation of these algorithms over an extensible list of datasets. Wide variety

of existing SLAM algorithms are supported. It provides implementation in many programming languages from generic to specific ones (C++, OpenMP, OpenCL and CUDA). These algorithms can be tested on a variety of multicore and GPU accelerated platforms. The framework is growing to support more and more algorithms. This framework concerns high-level layers and requires an existing operating system. In addition, the list of supported platforms doesn't include FPGA based platforms. The evaluation on this type of platform can result in better performance over other type of platforms.

Contribution

The objective of this thesis is to propose a generic prototyping platform, to aid in the design and production of embedded systems dedicated to SLAM. This platform minimize the time to market cost and risks, while taking into account constraints related to embedded system sand SLAM. Accordingly, the following are the major contributions of this thesis:

- A prototyping platform for designing SLAM systems is proposed designed. The platform, consisting of hardware part and software part, bases on System on Chip (SoC) and open source bricks. The platform offers the necessary resources to design an embedded SLAM system

- A design methodology is used to implement the SLAM algorithms on the platform, based on already existing implementations. In this methodology a performance analysis is used in order to make the Software/Hardware partitioning decision. We also use OpenCL as a high level language to facilitate the hardware design, reduce design time and allow the possibility of porting to other targets.

- A validation of the proposed platform is done by implementing a SLAM system in DE1-SoC board. This board is a low-cost SoC-FPGA that combines a mobile hard core CPU with an FPGA in the same chip.

Thesis Organization

The thesis is organized as follows:

Chapter 1 presents a retrospective on SLAM algorithms and embedded SLAMs. Some works are cited in this context. In addition, some recent works, on which our study was based, is presented.

Chapter 2 introduces embedded systems and design methodologies of embedded systems, to better choose the implementation methodology and the design and evaluation tools.

In, Chapter 3 we proceed to the presentation of our prototyping platform. The platform contains two parts: the hardware part, and the software part. In the hardware part, we see the different existing architectures, then we make our choice on the SoC-FPGA which will be the basis of our platform. We see the design methodology on this SoC, the high level design tool (OpenCL) and performance analysis tools. In the second part, we present the software designs of our platform. We also see the Buildroot tool which is used to build a Linux system customized to our application and the design methodology and evaluation tools used.

In chapter 4, we present a case study of an implementation of an EKF SLAM on our platform. We analyze the performance obtained and compare with other implementations.

Finally, a General Conclusion summarizes the thesis, and highlights the main contributions of the research. It also contains recommendations for future works.

CHAPTER 1

SLAM AND EMBEDDED SYSTEMS

Contents

1.1 Introduction

Localization and mapping are essential tasks in robotics. Autonomous mobile robot navigating in an environment needs to localize itself and map its surroundings to take the appropriate decisions. In an unknown environment, these two tasks are performed simultaneously: Simultaneous Localisation and Mapping (SLAM). The SLAM is performed using an algorithm that exploits the data collected by the sensors from the environment.

In robotic, SLAM is needed to support other tasks such as path planning, avoiding obstacle, etc., and to provide a graphical visualization to humans. SLAM also corrects the localization errors generated by the proprioceptive sensors like odometer, by revisiting the area already visited, in this case we speak of "loop-closure".

The interest in using SLAM appears in military, rescue, and exploration applications. For such applications, a robot must be able to localize itself and build a cartography of dangerous non-accessible environment for humans. These places may have bad or no positioning system coverage, and whose environment is unknown, such as caves, collapsed buildings,etc.

In the last two decades, researchers reported several algorithms to resolve this problem. However, these algorithms require a lot of computation. Besides, the SLAM's application field is in mobile robots. This means it involves the use of an embedded system to run these algorithms. However, an embedded system is limited on processing power and energy consumption. It requires designing an appropriate architecture to get around these limitations.

In this chapter, we describe the SLAM problem. Then, we give a brief history of the SLAM. We give also the most important works done to develop SLAM algorithms and to implement SLAM on embedded systems.

1.2 Simultaneous localization and mapping

The SLAM problem, in robotics, addresses two questions: Where am I in the environment? and, what does the environment, where I am, look like?. The answer to

the first question gives the position of the robot, while the answer to the second gives the map of the environment. These questions seem intuitive to humans, however, this is not the case for robots. In fact, these two questions are related, and each question needs the answer of the other question. A robot must have the environment map to localize itself in this environment. On the other hand, the environment is unknown, and to build the map of this environment the robot must have information about its position.

Localizing a robot means giving its 3 dimensional position (x, y, z) and orientation (roll, pitch, yaw). These parameters are used to describe the state of the robot. In addition to these parameters, in some cases, other parameters of the robot are used to describe the state of the robot too, like linear and angular velocities, and linear and angular accelerations.

Mapping the environment means giving the positions of the points of interests that describe the environment around the robot.

Mathematically, the problem can be interpreted with the mathematics formula:

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) \tag{1.1}$$

Where:

- $X_{0:k} = \{x_1, x_2, ..., x_k\}$ is the current location and orientation of the vehicle
- $m = \{m_1, m_2, ..., m_n\}$ the map landmarks
- $Z_{0:k} = \{z_1, z_2, ..., z_k\}$ is the set of all landmark observations
- $U_{0:k} = \{u_1, u_2, ..., u_k\}$ is the history of control inputs

The localization question aims to estimate the current location x_k with respect to the landmarks, and it is described in the form of the probability distribution $p(x_k|Z_{0:k}, U_{0:k}, m)$. This assumes that the locations of landmarks m are known as well as the observations $Z_{0:k}$ and the control inputs $U_{0:k}$. Conversely, the mapping question aims to estimate the landmarks location m assuming that the current location x_k is known with the observations $Z_{0:k}$ and the control inputs $U_{0:k}$. It is described in the form of the probability distribution $p(m|X_{0:k}, Z_{0:k}, U_{0:k})$

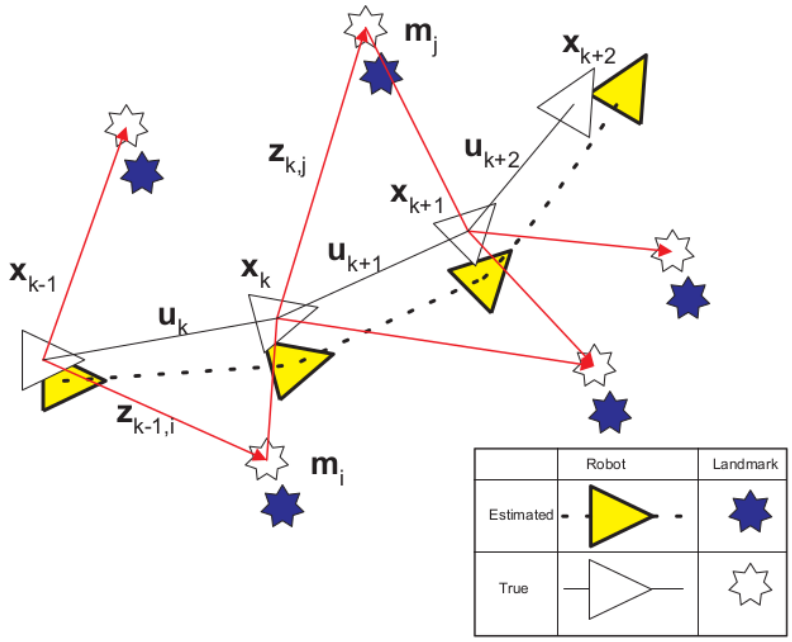


Figure 1.1: The essential SLAM problem [5]

Figure 1.1 shows the relation between the mobile vehicle path (successive locations) and the landmarks. $z_{k,j}$ are the observations made between the true mobile vehicle and landmarks locations, and the u_k are the control inputs that move the mobile vehicle between two true successive locations. The true locations of the mobile vehicle and the landmarks are never known or measured directly, however, they can be estimated from the observations and the control inputs made.

To compute these estimations from the control inputs and the observations, we need mathematical models that describe the state transitions and the observations. Two models are used: motion model and observation or measurement model.

The motion model is the probability density that describes the state transition of the mobile vehicle from state x_{k-1} to x_k knowing the previous state x_{k-1} and the control input u_k. It can be described in the form:

$$P(x_k \mid x_{k-1}, u_k) \tag{1.2}$$

The control input uncertainty (translation and rotation) affects the probability density of the vehicle state. This uncertainty propagates by the motion model with the motion of the vehicle, so that it accumulates as the vehicle moves forward. The control input is

generally measured either with an odometer or a speed sensor. Even the two sensors suffer from slip and drift errors, the odometer is more accurate than the speed sensor. This is because it gives directly the position of the vehicle, while a speed derivation is needed to get the position when using the speed sensor. This derivation increases the impact of the errors on the results. In systems without control inputs, only the uncertainty of the previous state is propagated.

The observation model describes how a landmark is observed (z_k) knowing the landmark position m and the vehicle state x_k. It is described in the form of the probability density:

$$P(z_k \mid x_k, m) \tag{1.3}$$

The sensors used to observe landmarks and measures the distances are called exteroceptive sensors since they are linked to external measurements. Many types of sensors can be used, and each type of sensor has its own observation model. Errors related to the sensors determine the accuracy of the measurements.

These previous descriptions are the basics of the various solutions to the SLAM problem. These solutions are the result of extensive research over the years, making a great story of SLAM.

1.3 Structure of a SLAM system

A SLAM system has three main parts: the data acquisition part, the preprocessing part, and the processing part of SLAM. In the data acquisition part, data are acquired by sensors. There are two types of sensors: proprioceptive sensors and exteroceptive sensors. In the preprocessing part, useful informations are extracted from the data received from the sensors using filters and detectors algorithms. The processing part represents the core of SLAM, we use the data produced by the second part to perform the localization and mapping process. This composition is summarized in Figure 1.2.

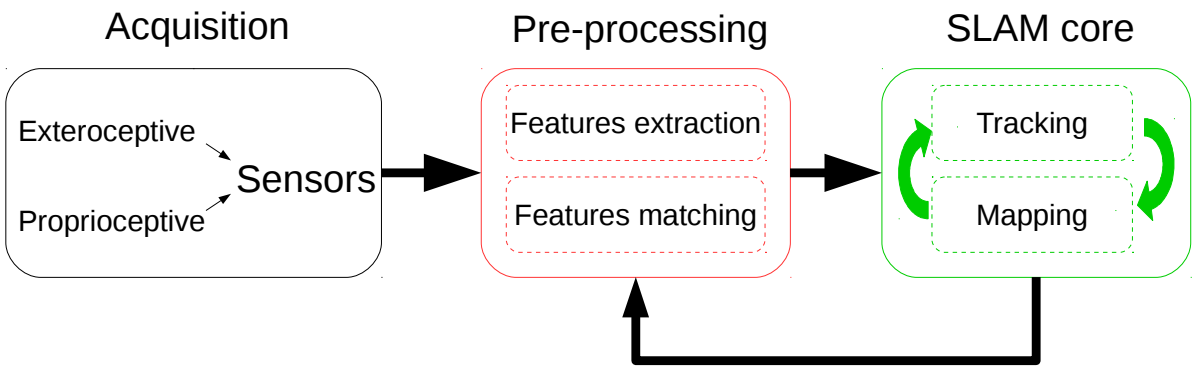


Figure 1.2: SLAM system main components

1.3.1 Data acquisition phase

This phase is considered the input phase of the system. The variables of the state of the mobile vehicle and that of the external environment are measured using sensors. There are two types of sensors, according to the type of measurement: proprioceptive sensors and exteroceptive sensors.

1.3.1.1 Proprioceptive sensors

Proprioceptive sensors measure values internal to the system, in this case the mobile vehicle. In other words, it measures the internal state of the system. Examples of these measurements are: distance traveled, velocity, acceleration...etc. These sensors are not sufficient to implement a SLAM since it requires data from the external environment to create a map. In SLAM two type of proprioceptive sensors are used

Odometer An odometer is an electromechanic device used to measure the distance traveled by a mobile vehicle. It consists of a rotary encoder installed on the wheel to give direction and measure the distance traveled. However, many factors can affect the odometer measurements results, like wheel slip and odometer precision. These will generate cumulative errors with the movement of the vehicle.

Inertial measurement sensor An inertial measurement unit (IMU) measures the acceleration and angular velocity of the vehicle along three perpendicular axes. The

velocity and the distance traveled can be derived from accelerations, using derivation, and the same for the angular deviation. However, The results are affected by cumulative errors of measurement, just like the odometer.

1.3.1.2 Exteroceptive sensors

Exteroceptive sensors measure values external to the system. In other words, it measures the state of the external environment. Examples of these measurements are distances, light intensity, ...etc. Some SLAM systems can do their jobs using exteroceptive sensors only. Some algorithms can extract the robot state using only external data. In SLAM, exteroceptive sensors used can be classified into range sensors and bearing sensors

range sensors Range sensors measure the distance between the sensor and objects in the environment (the range). Laser sensors are the commonly used range sensors in robotics. However, it is hard to use on experimental and low-cost platforms because it is an expensive sensor. Ultra-sonic sensors are also used in robotics as range sensors [6].

bearing sensors Most common bearing sensor is the camera. A lot of computer vision algorithms exists, this makes them widely used in SLAM systems. The camera gives us a 2D representation of the 3D space. The camera must be calibrated in order to get a precision data. The calibration parameters are used in the measurement model to make a precise representation of the environment.

bearing-range sensor The bearing range sensors are sensors that are capable of giving both the bearing and the range of the objects in the environment. An omnidirectional laser sensor is a bearing-range sensor, that gives the direction and the range of objects. An RGB-D is a type of camera with depth sensor that gets each pixel's depth.

1.3.2 Pre-processing phase

There are two methods regarding the use of sensor data by SLAM: direct method and features-based method. The data acquired by the sensors contains a lot of information in raw format. In direct method, all the information of the sensor data is used by the SLAM core, to build a dense map. This method does not require a pre-processing phase but has more computational costs. In features-based method, the data acquired by the sensor are not used directly by the SLAM core. Instead, they are pre-processed to extract distinguishing features that will be used to build a sparse map later. Besides, this simplifies the SLAM process. This phase contains two main process: features-extraction and features matching

1.3.2.1 Features extraction

Features extraction process searches to extract some points of interest from the sensor data. These points of interest must be identifiable and distinctive. In SLAM systems with a vision sensor, these points of interest are features extracted from images, and they are representations of landmarks in the environment. The relation between the position of the landmarks in the environment and the detected feature in the sensor data is described by an observation model. An initialization process can be done for each feature to estimate its position in the environment if it is impossible to estimate it at the first time, as is the case with monocular SLAM.

1.3.2.2 Features matching

The features matching process is responsible for matching the features from the previous frame with their corresponding features on the new frame. The information of the features from the previous frames are retrieved from the SLAM core. These information are the position of the feature, its descriptor, and the uncertainty of its position. The matching process uses the position and its uncertainty to delimit the search region, and the descriptor to find the exact position of the feature in the new image.

1.3.3 SLAM core

The SLAM core is the heart of the SLAM system. It runs the necessary algorithms for the localization and mapping. It contains two main process: tracking and mapping. Running these two processes is insufficient for long trajectories. In fact, returning to the start point after a long distance isn't detected, and the robot appears as it is performing odometry in a long corridor. The loop-closing process is added in some SLAMs in order to detect the loops in trajectory and correct the map after returning to a previous mapped place. This is usually used in outdoor applications, when the traveling long distances results in drifts.

1.3.3.1 Tracking

The role of the tracking process is to localize the sensor in the environment. In visual SLAM, it gives the pose of the sensor with respect to a defined reference. The results of theses poses over time are used to trace the path of the sensor. In key-frame-based SLAM, only frames with the most important poses are retained to trace the path, and are passed to the mapping process.

1.3.3.2 Mapping

The Mapping process, as it is named, builds the surrounding map, and corrects the existing map using the observation model. It retrieves informations from the tracking process and the sensor data. In direct SLAM, as mentioned before, there is no pre-processing of sensor data, and all sensor data information are used. The constructed map is dense, robust, and more detailed. However, this demands more computations. In features-based SLAM the map is constructed with the features retrieved from the pre-processing phase. The constructed map is a set of sparse features that describes the map. This reduce the computation complexity, but gives sparse and less detailed map.

The information from the mapping and tracking process are used by the features matching process in the pre-processing phase.

1.3.3.3 Loop-closing

A vehicle, after moving a certain distance, may return to an already visited area. Without additional process (to the tracking and mapping processes), this fact cannot be noted, and the vehicle seemed like it moves in a infinite corridor. Loop-closing is responsible for finding out whether the vehicle has returned to an area already visited or not. Then, it aligns the new and the previous regions of the constructed map to close the loop by calculating the needed transformations.

1.4 Embedded SLAM, History and state-of-the-art

In the last century, many works done outside robotics studied methods that was to be used by SLAM later, like least squares methods [7], Kalman Filters [8], bundle adjustment [9]...etc.

According to Durrant-Whyte and Bailey [5], The first works on SLAM begins to appear in the mid-eighties, with the works of Smith and Cheesman [10] and Durrant-Whyte [11]. Their works were about representing and describing the geometric uncertainty, using statistical methods. According to the same author, the acronym SLAM has first appeared in the seventh international symposium on robotic researches [12].

The SLAM development has evolved over the previous three decades. Cadena *et al.* [13] divided the SLAM evolution into three periods, Classical age from 1986 to 2004, where the main probabilistic formulations for SLAM are introduced. The second period from 2004 to 2015 is called the algorithmic-analysis age, where the fundamental properties of SLAM are studied, Theses properties includes observability, convergence and consistency. This period also saw the development of the main open-source algorithms. In their paper, Cadena*et al.* described the age after 2015 as the age of robust perception which is characterized by robust performance and high-level understanding of the environment.

1.4.1 Formulating and solving SLAM problem researches

The first researches focused on formulating and solving the SLAM problem. Probabilistic methods and filters, notably Extended Kalman Filter, particle filters were used. The graphical representations based SLAM comes later to find solutions to some limitations of the filter based approaches.

1.4.1.1 Kalman Filters:

Extended kalman filter was used in the earliest works [14], the goal is to minimize map uncertainty and estimate the robot and map state using uncertain observations. It uses a single state vector containing the locations estimation of the vehicle and the landmarks. The state vector is associated with a covariance matrix that represent the uncertainty of the state vector estimations, and the correlation between them. This approach assumes that the environment is represented using features. These features are used as landmarks to in the state vector.

As the robot moves, it takes measurement from the environment. At each move, it predicts its position and the observation of the landmarks according to the previous state and the control inputs. Then it updates the state vector using the new measurements.

As new features is detected, they are added to the state vector, and the covariance matrix size grows quadratically. The Extended Kalman Filter calculates the entire state vector and covariance matrix. The advantage of calculating the entire state vector is the propagation of the correction information to all previously observed landmarks. However, the computation complexity increases quadratically with the addition of the features to the state vector. This is considered as the main disadvantage of this approach, especially in outdoor uses when the number of landmarks is large.

1.4.1.2 Particle Filters:

Another approach was adopted to deal with some limitation of the EKF approach, especially with the large environment, is the particle filter [15]. In this approach, we use a set of particles. Each particle contains a sample path and 2D guassian

representation for each landmark. These particles represent the state of the system (the vehicle and landmarks). It has two main steps: prediction and correction. The prediction uses the motion model to predict the new location of the robot and predict the measurements. The particles are then sampled according to this prediction with the addition of system noise. In the correction step, the observations from the sensors are used to compute the weights of each particle. The weight measures how well the particle represents the real state of the system. In the end, a resampling step is done to redistribute the particles according to these weights, the good particles are likely to survive and the bad particles are likely to die. And the process repeats the steps again. The correlation between the landmarks are not calculated which reduces the computation complexity comparing with EKF. The particles are distributed to represent the probability of the state of the system FastSLAM [16] is one of the first works in SLAM that uses particle filter. They used a type of particle filters known as Rao-Blackwellised.

1.4.1.3 Graph based SLAM:

A third approach is used in SLAM which is graphical SLAM, it is used in Graph-SLAM [17] and is based on graphical representations. In this approach, sparse non-linear optimization methods are used to solve the SLAM problem. The system is represented with a graph with nodes and edges (see Figure 1.3) . The series of the consecutive vehicle locations and landmarks are represented by nodes. These nodes are related with edges. The edges represent the constraints between the nodes. The control inputs are represented with edges between consecutive vehicle locations nodes, and measurements are represented with edges between the landmark nodes and the corresponding vehicle location node where they are observed. The number of constraints increases linearly with the number of the nodes since each node is only connected to a small number of other nodes. The Constraints are computed as the uncertainties of the observations. To find the configuration of the nodes (robot state and landmarks), these uncertainties are minimized using graph optimization algorithms.

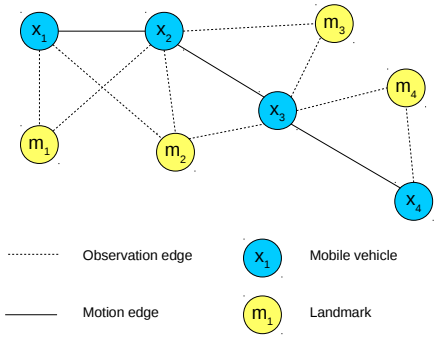


Figure 1.3: Graph-based SLAM illustration

1.4.2 Embedded SLAM implementation researches

In the second decade, the implementation of SLAM algorithms becomes interesting. Many works have sought to implement different algorithms on different computer systems. The goal was to design a system running SLAM in real time with more precision. One of the remarkable works is the one of Davison [18], where he implemented an EKF-SLAM running in real-time. The SLAM community developed many SLAM implementations, and they made many of these implementations open-source. This helps the community to contribute in these open-source project and provide a complete SLAM solution [3, 19–22].

Over the past two decades, researchers were able to implement the different SLAM algorithms in different platforms from powerful computing machines to less powerful embedded systems. These works aim to get SLAM implementation with more certainty, precision, and operation in real time, in order to integrate them on mobile robots.

As the SLAM is needed to be run on a mobile robot, researches are conducted to implement the SLAM on embedded targets. Implementing SLAM on an embedded system presents a challenge by taking into account the limited performance, memory, and power consumption of the embedded systems. Table 1.1 shows some recent examples of SLAM implementations in embedded targets. We observe that the works can be divided into Four, depending on the acceleration method used: pure software optimization, acceleration using parallel hard co-processors, acceleration using FPGA,

and the use of high level programming language.

Table 1.1: SLAM implementations in embedded systems

Reference	Algorithm	Implementation	Platform	Acceleration hardware	Year
Gonzalez *et al.* [23]	C-SLAM	C	PPC440/ARM9/VIA EPIA	PPC440, ARM9, VIA EPIA	2011
Vincke *et al.* [24]	EKF-SLAM	C++/NEON/DSP	OMAP3530	ARMv7-A/NEON/DSP	2012
Vincke *et al.* [25]	EKF-SLAM	C++/NEON/OpenMP	OMAP4430	ARMv7-A/NEON	2014
Abouzahir *et al.* [26]	FastSLAM	C++/OpenGL	Tegra K1	ARMv7-A/GPGPU	2016
Bonato *et al.* [27]	EKF-SLAM	Handel C	Altera Stratix II	FPGA	2009
Tertei *et al.* [28]	EKF-SLAM	C++/HDL	Xillinx Virtex 5	PPC440 + FPGA	2014
Gu *et al.* [29]	VO-SLAM	*Not Mentioned*	Altera Stratix III	FPGA + NIOSII	2015
Idris *et al.* [30]	EKF-SLAM	HDL	Xillinx Spartan-3A DSP	FPGA	2012
Botero *et al.* [31]	C-SLAM, RT-SLAM	C/C++/HDL	Xillinx Virtex 5 and 6	PPC + FPGA	2012
Fang *et al.* [32]	ORB-SLAM	*Not Mentioned*	Altera Stratix V	FPGA + ARM	2017
Liu *et al.* [33]	ORB-SLAM	*Not Mentioned*	Xilinx Zynq 7045 SoC	FPGA + ARM	2019
Abouzahir *et al.* [34]	FastSLAM	C++/OpenCL	Intel Arria 10 SoC	Host PC + FPGA	2018
Boikos *et al.* [35,36]	LSD-SLAM	C++/HLS	Xillinx Zynq-7020	ARM/FPGA	2017
Bodin *et al.* [3]	Multiple algorithms	C++, OpenCL, CUDA, OpenMP, PThread	Multiple platform	Multi-cores-CPU, GPU	2018

1.4.2.1 Software optimization

In this type of optimization, only the software is optimized. The developer does not have to know about the hardware, but a good knowledge about the algorithm and the code is required. Gonzalez *et al.* [23] designed a new architecture of a C-coded program for visual SLAM intended to be implemented on different embedded boards. In that work, only the software is optimized to face the limited resource of the embedded targets. The software must respect some strong constraints like DAL standard (Design Assurance Level), where the external libraries are forbidden in addition to the use of recursive loops and dynamic memory allocations.

1.4.2.2 Hard co-processors

Other researchers explored the hardware domain, to benefit from the performance offered by the existing hardware. The hard co-processors have parallel architectures and run at a high frequency. They programmed using languages specific to these hardware or using generic languages and specific compilers. Several heterogeneous architectures are proposed using multi-core CPUs, DSPs, and GPUs to accelerate complex computations and fit with the embedded system requirements. Vincke *et al.* [24]

implemented an EKF SLAM on a multi-processor system, the authors used a heterogeneous architecture, a low-cost system composed from a single-core ARM processor equipped with an SIMD co-processor in addition to a DSP in the same chip. The authors analysed the SLAM program and picked up the main tasks and their processing time, then the program is partitioned into functional blocks, each block is implemented on CPU with NEON or in the DSP, depending on its processing nature. The processing time is reduced by a factor of 4.7 from 80.85ms on an ARM to 17.5ms in a non-optimized implementation.
In a later work [25], a Dual Core ARM (with a dual NEON) is used in order to optimize some non-optimized blocks with the help of OpenMP library. This reduced the processing time by a factor of 2.75 from the non-optimized implementation.

Abouzhir *et al.* [26] proposed an implementation of FAST-SLAM2.0 on Tegra K1 SoC equipped with a Quad-core ARM with NVIDIA GPU, the algorithm was partitioned on functional blocks. Each functional block is then implemented on the CPU or the GPU according to its nature. The sequential tasks run on the CPU, while the tasks with intensive computations are processed by the GPU. The embedded GPU is programmed with OpenGL among the existing choices, which gives the possibility to be implemented on multitude GPU hardware. The processing time is reduced to 120ms by a factor of 37 from a non-optimized implementation on a single-core CPU.

1.4.2.3 FPGA and SoC-FPGA

In addition to the works that use platforms with hard processors, other works took advantage of the high customizability of the FPGA and low power consumption to design custom accelerators and used them as co-processors.

Many works discussed the acceleration of matrix operations with FPGA like [27–29], they analyzed the algorithm, the complexity of the algorithm used in each case, and the required memory bandwidth, then proposed architectures to speed-up some functions of the algorithm. Bonato *et al.* [27] presented one of the first implementations of EKF-SLAM in an FPGA. They proposed an FPGA-based architecture with parallel access to the memory banks, to accelerate matrix multiplication and increase

memory bandwidth. In the same way, Tertei *et al.* [37] proposed a hardware architecture to accelerate the matrix multiplications using systolic arrays. The multiplier is used as a coprocessor with a PPC 440. Gu *et al.* [29] implemented a VO-SLAM on DE3 board (Altera Stratix III). A hardware architecture is proposed to accelerate Matrix operations. A NIOSII soft-core is used as the master processor.

Some other works have chosen to accelerate other functions like [30–32]. Idris *et al.* [30] proposed pipelined and parallel architecture to accelerate a cross-correlation function on an FPGA. The decision is taken after profiling the entire algorithm. Botero *et al.* [31] proposed an FPGA circuit to accelerate the Harris point extractor from pictures coming from the camera acting as pre-processor. Fang *et al.* [32] proposed an implementation of an ORB feature extraction on an FPGA for visual SLAM. The used hardware is a Stratix V from Altera controlled by an ARM multi processors.

Nowadays, SoC-FPGA is getting popular as a powerful embedded system containing an embedded CPU with an FPGA in a single SoC. The FPGA is used to design a highly parallelized circuit and use it as an accelerator. The advantage of using such SoC, is to have the full system with the FPGA accelerator implemented in one SoC, reducing the size, the energy consumption and the hardware-software communication cost. Liu *et al.* [33] used a Zynq 7045 SoC, a SoC-FPGA from Xilinx, to implement ORB-SLAM. The feature extraction and matching are implemented on the FPGA, while the remaining functions are executed on the ARM Processor. The feature extraction and matching functions are modified to be hardware friendly.

In a recent work by Boikos *et al.* [38], implemented a semi-dense SLAM on SoC-FPGA. In the first work [35] the tracking block is implemented in the FPGA while the other blocks are running on the ARM side. In a second work [36] the mapping block was chosen to be implemented in the FPGA. In their works, they used a High Level Synthesis (HLS) to design and implement the accelerated blocks in the FPGA.

1.4.2.4 High level and Parallel computing languages

Parallel computing frameworks bring to the SLAM community powerful tools to implement SLAM algorithms on parallel and heterogeneous architectures. One of the

interesting works used those frameworks is SLAMBench Project. In the 3 papers of the projects [2–4] They developed a framework to implement a set of SLAMs on different hardware platforms. The SLAM algorithms are implemented using C++, OpenMP, CUDA, and OpenCL. They aim to compare the computational performances and power consumption between these implementations using different targets. The result shows that the parallel computing implementations (CUDA and OpenCL) have high performances against OpenMP and the single-core C++. The OpenCL/GPU also shows less power consumption compared to other configurations. We observe that platforms with FPGA are not included in their tested targets.

Besides using it as a parallel computing language to program parallel computing units, OpenCL is a cross-platform language that is used to program heterogeneous architectures. In the FPGA domain, OpenCL is considered as high-level synthesis tool that offers parallel computing capabilities. Abouzahir *et al.* [34] used OpenCL to implement Fast-SLAM on the different targets. The used four targets were: a high-end desktop machine two targets containing multi-core ARM CPU, with GPU, and an Arria 10, a mid-range SoC-FPGA from IntelFPGA. The experiments show an improvement of the FPGA implementation against embedded GPU and high-end GPU implementations.

Another tool used as a high level language for FPGAs is HLS (High Level Synthesis), Boikos *et al.* [35], have used it in their work mentioned above. HLS is a C-like language supported by both Xilinx and IntelFPGA, it facilitates the design of FPGA circuits.

1.5 Conclusion

A lot of works in the last decades studied the algorithms of simultaneous localization and mapping. In this chapter, we introduced the simultaneous localization and mapping (SLAM) problem. We described the main mathematical models in which the SLAM problem is based, the motion model that describes the motion of the mobile vehicle in the environment and the observation model that describes how the mobile

vehicle sensor see the environment.

We presented a structure of a SLAM system, from the acquisition part to the SLAM core passing by the pre-processing part. We saw the two types of sensors used in the SLAM: proprioceptive sensors depend on the internal parameters, and give an inconsistent estimation of the mobile vehicle state, and exteroceptive sensors depend on external parameters and improve the state estimation. The nature of the SLAM influence on this structure, in addition to tracking and mapping in the SLAM core, loop-closing are used in some SLAM algorithms to correct deviations in the SLAM map. In the feature based SLAM, extracting and matching features constitutes an essential tasks. These tasks are done in a pre-processing phase.

Finally, we made a literature review about the SLAM evolution history and the state-of-the-art of SLAM implementations in embedded systems. We saw the main approaches used to solve the SLAM problem, which are Kalman filters, Particle filters and graph based SLAM. We classified the main works about SLAM implementation according to the main techniques used to implement SLAM on embedded systems.

CHAPTER 2

EMBEDDED SYSTEMS DESIGN

Contents

2.1 Introduction

The evolution of SLAM algorithms over the years aims to increase precision and robustness. These algorithms require efficient implementation in order to allow real-time execution. Indeed, faced with the growing complexity of these algorithms implemented in embedded systems, and faced with the need to reduce time to market, methods and associated tools are necessary to automate the process of implementing these algorithms on embedded platforms. During the last decades, the architecture of processors and computers as well as the tools of development and design have evolved considerably. This facilitated the implementation of algorithms and the development of complex applications. Therefore, the realization of these systems required a software/hardware co-design methodology that takes into account the constraints of the embedded SLAM system.

In a hardware-software co-design methodology, studying both algorithmic and architectural aspects simultaneously is necessary to achieve better implementation. For this, we first give an introduction to embedded systems, as well as the main characteristics of embedded SLAM systems, this allows us to choose the implementation methodology and the design and evaluation tools. Next, we see the approaches used to design embedded systems, the design and evaluation methodology, as well as the tools used for the implementation.

2.2 Embedded Systems

An embedded system can be defined as an autonomous electronic system that does not have standard inputs/outputs such as a keyboard or computer screen. It is special-purpose system integrated within a larger system to provide a dedicated service to that. In embedded systems, the software is closely linked to the hardware system, and they are not easily discernible. Usually, this software is a function-specific application provided by the manufacturer, and the end-user has a limited access to the application parameters. [1,39]

Embedded systems are widely used nowadays, because many functions, formerly

performed by mechanical or analog systems, are replaced by software components. They are found in consumer electronic products, transport control systems, industrial process control, Telecommunication...etc.

2.2.1 Characteristics of an embedded systems

Embedded systems are differentiated from other digital systems by some common characteristics. Vahid And Givargis [40] describe these characteristics:

- **Single-functioned:** Unlike, PC-like machines that are polyvalent and can executes variety of programs, an embedded system usually executes a specific function repeatedly. For example, the function of a network switch is to receive and redirect data to destination using packet switch, no other function can be executed on it.

- **Reactive and real-time:** An embedded system must be reactive and react to the change on its input continually. Embedded systems can be classified in two categories according to the type of the reaction:

  Hard Real-time: No data sample can be dropped, a delayed response is not permitted. If a data sample is dropped or a delay has been generated, it could results in a serious impact or even failure in the embedded system's mission, for example, a car's brake system must be responsive to the brake signal and provide the right control signal to the brake in real-time, a delayed response may result in a fatal accident.

  Soft Real-time: A delayed response is permitted and data sample can be dropped to maintain the real-time execution of the function. For example a digital video decoder can differ the video sequences or even drop some images, without causing serious damages.

- **Tightly constrained:** An embedded system have tight constraints on design metrics. These metrics are:

  Cost: An embedded must have a low cost.

Energy consumption: As the majority of embedded systems run with battery, it must be designed to consume minimum power to extend battery life. This will also reduce heat dissipation and therefore no need for a cooling system.

Size and weight: From their names embedded systems are usually embedded on larger mechanical or electrical systems. This requires a tiny size and light weight to be placed inside these systems, and can be easily transported, especially for portable systems.

Environment constraints: constraints dictated by the environment such as temperature, humidity, vibrations, shocks, power supply variations, RF interference, corrosion, water, fire, radiation ... etc.

reliability: It must ensure continuous running for years without errors, and in some cases, it must ensure an auto-recover if an error occurred.

2.2.2 General architecture of an embedded system

An embedded system is an electronic system that interacts with the environment.It receives the measurement information from the external environment, processes them and generates the output signals to the actuators. Figure 2.1 shows a general architecture of an embedded system. We can see that it consists :

1. *CPU:* It is the heart of the system, it runs the main program stored on memory, process data received from inputs and generates outputs, and it controls all the other peripherals. The CPU used in embedded systems are characterized by low power consumption, low heat dissipation and low clock frequency. They vary according to their instruction size, address range, etc. They constitutes the majority of the CPU in the market.

2. *Memory:* It stores the main program that is executed by the CPU, and the data used by the program. While stored on non-volatile memory, the data used by the program are stored in volatile memory (RAM) since they are used in run-time.

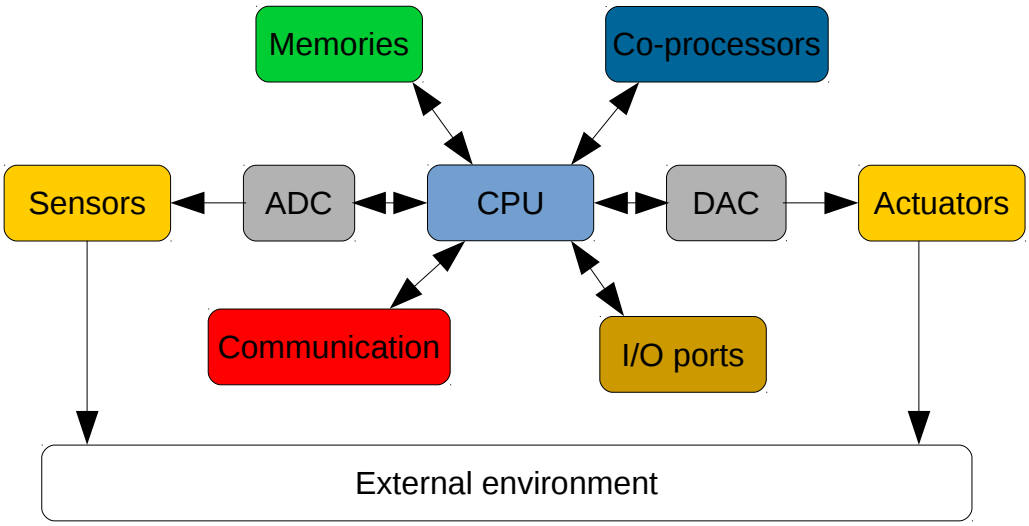


Figure 2.1: General architecture of an embedded system

Many types of memories can be used on an embedded system. These memories differ according to their bandwidth, read/write latency, and size. In general, the size and the bandwidth of memories are inversely proportional. The very used data are stored in memory with high throughput and low latency, while the less used are stored on low throughput.

3. *Co-processors:* Co-processors are electronic integrated circuit used to complement the functions of the main processor (CPU). It is specialized in rapid execution of a particular type of processing. Intensive tasks are offloaded from the main CPU to accelerate them on the co-processor, and increase the system performance.

4. *I/O ports:*I/O ports are used to communicate with other peripherals and to send and receive signals. These ports can be GPIO, USB, I2C, SPI...etc

5. *Communication:* Embedded systems can support communication with other computers to transmit its status, receive commands or for debugging.

6. *Sensors and actuators:* Sensors and actuators are the interface of the embedded system with the external environment. Sensors are the inputs and actuators are the outputs. Sensors transform a physical quantity into an electrical signal,

while actuators do the inverse by inverting the electrical signals from the CPU to an action.

The majority of the embedded systems are implemented using heterogeneous systems [39]. This means that the system is composed of different types of processing units. The system can contain in addition to the CPU, a GPU, programmable circuit (ASIC, FPGA), DSP, microcontrollers...

The heterogeneous systems are implemented either in single board or in single chip. In this latter case, we talk about System-on-Chip (SoC). [39]

Comparing the two types of implementations, the SoC provides better performance and lower power consumption. The distance between the components of the SoC are very small, which allows to have a high data transfer rate between these components, in addition to minimizing the heat dissipation caused by the joule effect. However, in single board systems, the components may be replaced, customized or even modified.

In many SoCs, the system is engraved in the chip at the factory, which make the system uncustomizable and impossible to replace its components. Another type of SoCs are the configurable circuit like FPGA and ASIC. These circuits give the designer possibility to develop his own system at his level.

2.3 Design Methodology

The continuous decrease in geometry size and increase in chip density has greatly increased the complexity of digital systems. This makes it possible to integrate a complete complex system on a single chip. As mentioned before, in a System-on-Chip, many modules of a complex system are integrated in just one chip. This saves greatly space, energy and size. However, the design of such systems becomes more complex and take more time for development. An automated design flow with efficient tools is necessary. One solution to dealing with these complexities is to move to higher levels of abstraction, at the system level [41].

In embedded systems we have to deal with the design of both the hardware and

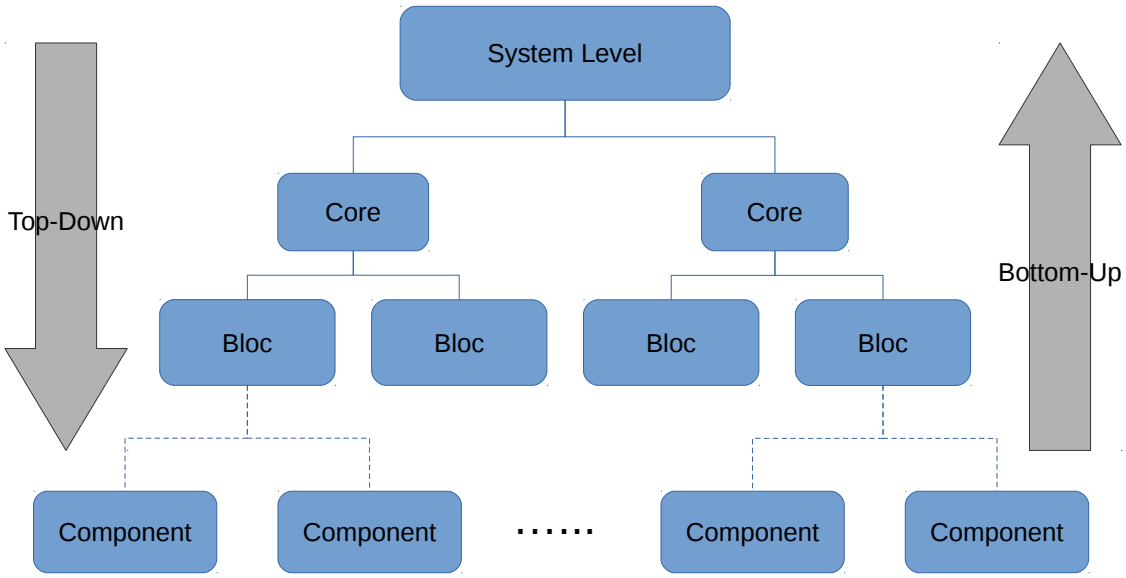


Figure 2.2: Design approaches of an embedded system

the software. In order to achieve the objectives stated at the system level, we must opt for a cooperative design of hardware and software, which is called Hardware/software co-design.

2.3.1 Embedded Design approaches

In literature, there are several works related to developing suitable methodologies to design an embedded system. These methodologies are classified into 3 main approaches, according to the direction of the design flow showed on the Figure 2.2:

- *Top-Down approach:* it starts with system description and generates the architecture from the system behavior. It is performed by gradually adding implementation details to the design. The sequence of design decisions drives the designer toward a solution that minimizes the cost of the micro-architecture. However it is difficult to reuse the blocks on the lower levels. This because optimization is preferred over the standardization. It is called sometimes Hardware/Software

co-design approach [39] because the hardware and the software are designed together in this approach.

- *Bottom-Up approach:* It starts with designing the low-level components, and assemble these components to produce the final systems. In general, in a bottom-up approach, the low-level components are designed to support a set of different applications that are often vaguely defined, this supposes that they have a standard interface to be easily used to produce the final system. This approach maximizes the number of applications that can be used by the designed components. It is called component-based approach. [39]

- *Meet-in-the-middle design:* It is a combination of the previous approaches. It is also called platform approach [39]. Rather than generating the architecture from the system behavior downing to the elementary components as in top-level approach, in the meet-in-the-middle approach the described system is designed using a set of standard predefined low-level components which are called IP cores, to optimize cost, energy consumption, efficiency, and flexibility [42], and gain in design time. In this approach we benefit from the advantage of both top-down and bottom-up approaches.

2.3.2 System Level Design

System Level Design is an electronic design methodology, focused on higher abstraction level concerns. In order to manage the complexities of the embedded systems, the design process begins at the highest level of abstraction. At the system-level, the specification of the system is described and modeled using a high-level language such as C/C++ or by using graphical design tools (Model-based design). Descending to lower design levels, automated design methodologies are used to enable step-by-step refinement.

There are 3 approaches to improve the design process labeled: synthesis, IP libraries, and verification [40]. Figure 2.3 shows these approaches and the relation with the design abstraction levels.

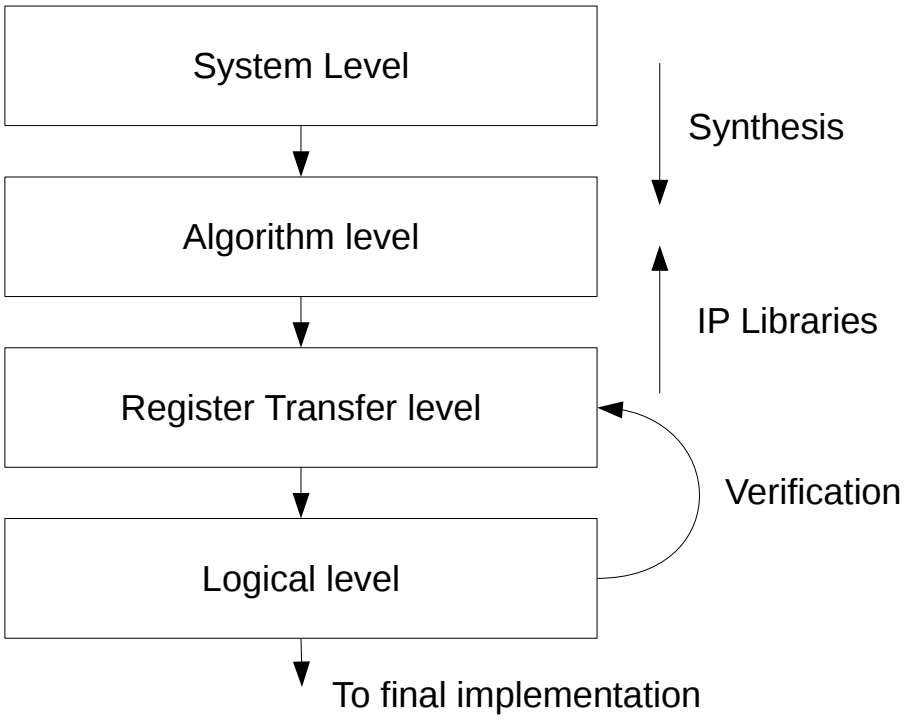


Figure 2.3: System-level design flow

- *Synthesis:* It is the process of exploring and generating the lower-level implementation details from the specification introduced in the higher level.

- *IP Libraries:* During the design process Intellectual Property (IP) components can be used when it is feasible. The pre-designed implementation from lower level are incorporated into higher level. This facilitates the design process and shortens the development time by using pre-designed implementations.

- *Verification:* At each level, we must ensure the correct functionality of the design at this level. This prevents wasting time on low-level debugging. There are many methods of verification, Simulation is one of the most common used methods.

With the evolution of the design tools, the design is automated and computer-assisted. The designer only has to deal with the system-level, and describe the system with a high language, while the automation tools automate the synthesis process and choose the right IPs to use in the system. The verification is also automated, and the

design can use the system level verification tools to verify the design at high abstract level.

2.3.3 Hardware/Software Co-design Methodology

As we have seen earlier in this chapter (see 2.2.2) An embedded system contains, among other components, programmable component which is the CPU, and accelerator circuits (co-processor). A co-processor can be a specialized processor like GPU, DSP, NPU... etc. or a programmable circuit like ASIC and FPGA. An embedded application can be implemented as a software running on a generic CPU and it can be implemented on hardware running on a co-processor. While software implementation is more cost effective and flexible, hardware implementation provide high performance and dedicated circuits. In order to maximize the performance of the embedded system, it is essential to share the computational load of the application between software and hardware resources.

The hardware/software co-design is a cooperative design of software and hardware. The software design engineers and the hardware design engineers are working together to design the embedded system instead of working separately. They focus on presenting a unified view of hardware and software, and the development of synthesis tools and simulators. It is used especially with heterogeneous systems that contain different types of processing units. [39]

Figure 2.4 shows a typical hardware/software co-design flow, as described by Shaout *et al.* [39].

Generally, Hardware/Software co-design follows a Top-down approach or a meet-in-the-middle approach. It begins with the specifications of the system, then the synthesis of the software and the hardware and finally the prototyping. The design can be done in several iterations.

The design flow starts with the system specifications. In this step, the behavior of the system is described without specifying the implementations. This description is written using a high level language like C/C++, etc. It is the first application model that will be used to evaluate the final prototype. In the costs estimation we estimate

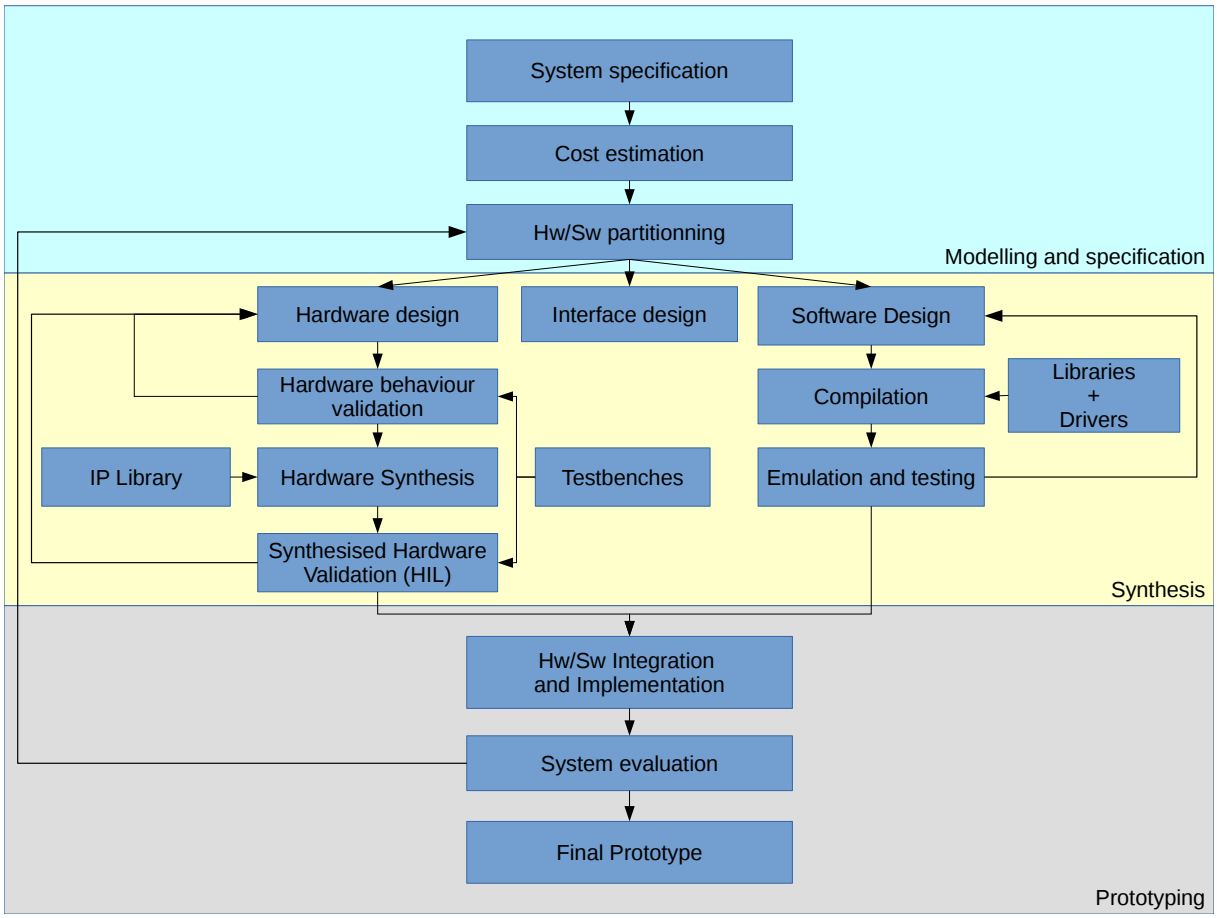


Figure 2.4: Hardware/Software Co-design workflow

the system implementation cost, like resource usage, memory usage, performance, power consumption, execution time, etc. Cost estimation helps us in design decisions and reduce design iterations. In the first iteration, only the software costs are evaluated. This step assists in making partitioning decision.

In the Hardware/Software partitioning step, we specify the parts of the application that will be implemented in software and those will be implemented in hardware based on the estimation of the previous step. The choice of which part of the application will be executed in which part of the hardware (hardware/software partitioning) depends on many factors like performance, cost, flexibility and development time or time-to-market. Processors provide a much easier implementation of complex algorithms, while configurable circuits are better by far in real-time processing and power consumption.

In the synthesis stage, the hardware and the software are designed and tested. The hardware is designed using description languages like VHDL, Verliog, or with High-level languages like C, OpenCL, etc. The behavior of the hardware is simulated and validated before passing to its synthesis. This decrease the development time. The hardware is synthesized using IP cores when it is possible, and then, the synthesized hardware is tested using a testbench with Hardware In the Loop (HIL) method. The software on the other side is designed, compiled, and emulated. Libraries and drivers are used when it is needed.

After validating the software and the hardware, They are integrated to to be implemented on the target platform. The entire system is evaluated in term of many parameters (execution time, memory usage, precision,etc.) to decide if an other iteration is required. The evaluation is done using a known dataset as input, and the output is compared with other results. If performances are not met, the hw/sw partitioning must be reviewed, according to the last evaluation of the system.

2.4 Rapid Prototyping and Platform Design

When designing an embedded system, many factors must be taken into account. Costs and time to market are factors that need to be reduced. Another factor, that need to be reduced too, is development risks, especially in complex systems. Before having the final system, it is necessary to go through prototyping. Prototyping is to make an incomplete and non-definitive copy of the final product. In a prototype, the design can be refined and validated to release the right one that meet the initial constraints. while prototyping reduces development risks, it can be expensive and take a relatively long time to get the final product.

Rapid prototyping is a method of using a set of tools that make it possible to achieve a prototype in a very short time, at a lower cost and with the minimum of tools and intermediate steps in the production process while guaranteeing the performance of the final product. it makes it possible to reduce the two remaining factors mentioned above: time-to-market and cost. [43]

2.4.1 Prototyping tools

The goal of the prototyping is study, then try and test a number of different solutions. For this, a good development tool must be flexible and modular. Development tools used in prototyping, usually, contains a library of functions, each allowing to perform a precise operation. This library (bank) allows the designer to choose the necessary functions (off-the-shelf). Some tools offer a graphical environment, the model is designed using functional blocks connected with wires that show the data flow. The development tool must be extensible by permitting the designers to add their custom functions to the library. These functions can be programmed using standard programming languages.

The components used by the prototyping tools must be standardized. This will reduce the development time and cost comparing to non-standard components, and it helps in simulating and verifying the system's functioning. In addition it and facilitate the reuse of these components. In this case, we talk here about Platform-based-design.

In case of a non-standard component a warp layer is developed to give the component a standard interface. [43]

2.4.2 Platform-based-design

In embedded systems, Hardware/Software co-design is used to find the right combination of hardware and software resulting in the most efficient product meeting the specification. However, the design of an embedded system cannot be done by a synthesis process using only the behavioural specification and without taking available components into account. Components reuse is unavoidable to cope with the increasing complexity of embedded systems and their time-to-market requirements. This led to the platform-based design methodology. [44]

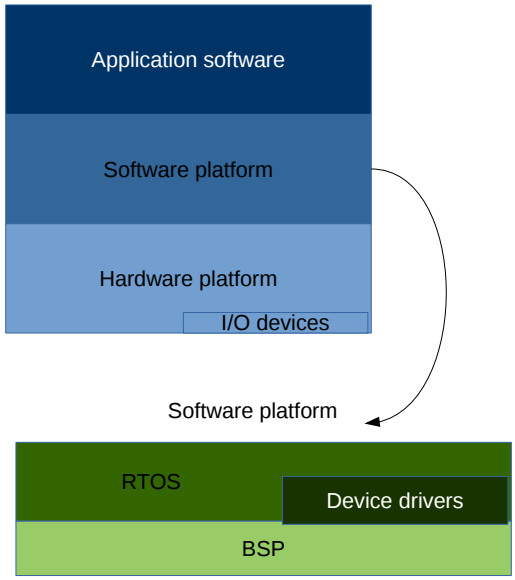


Figure 2.5: The layered structure of a system platform

Platform-based design is a design methodology following a meet-in-the-middle approach. This methodology emphasizes systematic reuse, to develop complex products based on platforms. In particular, as system designers will increasingly use software to implement their products, design methodologies that allow reuse of software are highly needed. This implies that the hardware architecture must remain fixed with a certain degree of parametrization. The hardware architecture consists of different cores that can be found on an embedded systems like programmable cores, programmable circuits, memories, inputs/outputs (I/O)...etc.

A family of hardware components that allow a substantial reuse of the software is called Hardware Platform. In the top of this hardware platform, an Application Program Interface (API) abstracts the hardware to a high-level where it can be seen

by the software. This interface is called the software platform (see Figure 2.5). The software platform is a set of software programs that are used to wrap the different parts of the hardware platform. We found the operating system, the device drivers, libraries...etc. The combination of the Hardware and software platforms is called the system platform. [42].

2.4.2.1 Hardware Platform

The hardware Platform contains a family of hardware components that allow the substantial reuse of the software. The decision of the hardware platform to use (The components that make up our platform), is defined by two sets of constraints.

The first are the constraints seen from the application domain (from the top). These constraints are given in terms of performance and memory size. The applications require a minimum processing speed and a memory with at least a given number of bytes. A more complex application requires stronger architectural constraints.

The second set of constraints are seen from the hardware domain (from the bottom). These constraints are given in terms of cost and power consumption. These constraints reduce the number of choices.

The hardware platform then is defined by the intersection of the two sets of constraints. In some cases, the hardware platform can be over-designed. It means that the constraints of the hardware domain are relaxed in favor of the application domain, to extend the application space and deliver new software. [42]

2.4.2.2 Software Platform

For application to take advantage of the hardware platform, it must be able to see a high-level interface of the hardware. This interface is the abstraction of the hardware platform. A software layer is used to form the hardware interface. This layer is called the software platform. This layer functions as a wrapper of the components of the hardware platform. This layer must be standard and have a unique representation of the hardware platform components. This means that a software application can be reused in different hardware platform or target. We speak here about platform

retargeting. [42]

Figure 2.6 shows the platform abstraction and the design flow of system platform. The application is mapped into the lower level abstract representation. This abstraction representation is chosen from a restricted library of available components (platform), to meet the constraints of the system and optimize cost, efficiency, energy consumption and flexibility.

The selected set of components from the platform defines a platform instance. In a single platform we can obtain multiple instances. Refinement of the application specifications leads to selecting one instance from the set of platform instances. The selection of the instance is guided by the parameters characterizing the platform components, this is called platform design space export [45].

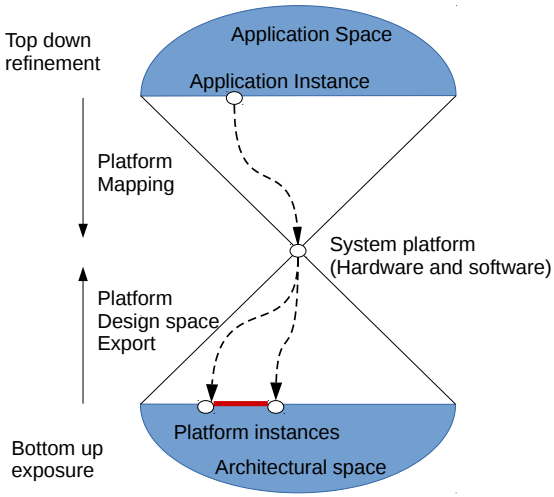


Figure 2.6: Design flow in platform-based-design

2.5 Conclusion

Current advances in semiconductor technology allow the development of complex digital systems on embedded systems, devices that can contain millions of transistors. Many types of computing components are developed, from the generic CPU to the specialized units like GPUs, DSP, etc... to meet the needs of the embedded applications. The current trend is to assemble several possibly heterogeneous components in a single chip to meet the requirements of complex embedded applications. These complexities at the hardware level make it difficult to design the system. A methodology that facilitates the design is essential to reduce the design cost and time.

In this chapter, we've seen an overview of embedded systems architecture. Then we had an overview of design approaches and methodologies. We've also seen a platform-based design methodology. This design methodology aims to ease reusing

and retargeting.

In the next chapter, we will present the proposed prototyping platform based on the methodology seen in this chapter, and we will describe its components and the design flow.

CHAPTER 3

TOWARDS A RAPID PROTOTYPING PLATFORM FOR EMBEDDED SLAM

Contents

3.1 Introduction

Implementing a SLAM algorithm in an embedded system requires a design methodology that takes into account a certain number of constraints such as processing power, memory size, power consumption, hardware weight and size, cost, etc. It must also consider constraints related to the SLAM, like large maps computations, large image processing, etc. As a first step, a prototype must be produced to test and evaluate the performance of the designed SLAM. In this step, the development time is important. The choice of the hardware and the tools to use depends on the type of the SLAM and the aforementioned constraints.

In this chapter, we propose a rapid prototyping platform for embedded SLAM. This platform is a combination of hardware and software platforms that permits to fully prototype a SLAM system and evaluate the performances. It aims to facilitate the design of the SLAM system and to minimize the time of its development while taking into account the design constraints. In such a platform, it must adapt to the different mobile platforms (southbound) on one hand and the development, design, and evaluation tools (northbound) on the other hand.

For this, we will describe both the hardware and the software parts of the platform. Finally, we give our proposed design flow.

3.2 Hardware Design

In order to choose the Hardware platform to be used on our work, many factors must be considered. First, this platform chosen must meet the specifications of the system. In our case, the objective is to design a generic prototyping platform, to help design and build embedded systems dedicated to SLAM. The system specifications lead us to define the constraints of the system to be taken into consideration, in particular, the processing power, the energy consumption, the physical properties (size and weight), cost... etc. The necessary tools to help design and program the system are also an important factor to choose the right platform. These tools facilitate and automate the design process. The more these tools offer the possibility to program and design at a

higher level the more the design process is easy and quick. another advantage of the tools that uses the standardized languages is that they allow multi-targeting, ie. that different type of platforms can be targeted with the same code.

In this section, we see the different possible choices of hardware platforms in context of SoC. For that, we see the different architectures that can be used and the difference between them. Then we see the tools used to design and evaluate the hardware of the SLAM system.

3.2.1 SoC architectures and technologies

As seen before in section 2.2.2, an embedded system contains a CPU as the central processing unit, acceleration circuits, memories and Inputs/outputs. In an SoC, all these components are embedded in a single chip and linked to each other using an on-chip interconnects, such buses and networks (Network on Chip or NoC). The CPU and the acceleration circuit are the processing components of the system that runs the application program.

3.2.1.1 System architecture

Many architectures can be used to implement a SLAM system on the SoC. Figure 3.1 shows a homogeneous multiprocessor system. In this system many processor cores are presented, and the software program is distributed between these processor cores. [46] presented an implementation of a SLAM application using a homogeneous multiprocessing system. It affects to each processor core a SLAM task (Feature extraction, propagation, update, mapping). These processor cores communicates with each other using a shared memory.

Most of the used SoC contain an acceleration circuit. The acceleration circuits are a specialized processor such as DSP, GPU, NPU, etc, or configurable circuit such as FPGA, ASIC, etc. Figure 3.2 shows another architecture where accelerator circuits are used. They are usually connected to the system and receive data from the main processor to be processed. Some acceleration circuits are used as to perform data preprocessing received from the input [31] as shown on Figure 3.3

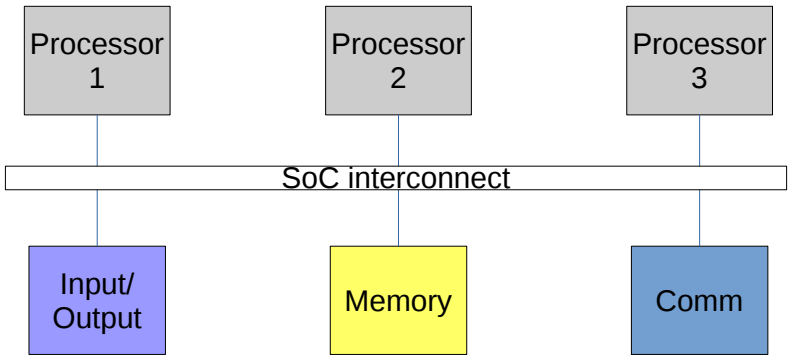


Figure 3.1: Architecture of a multiprocessor SoC

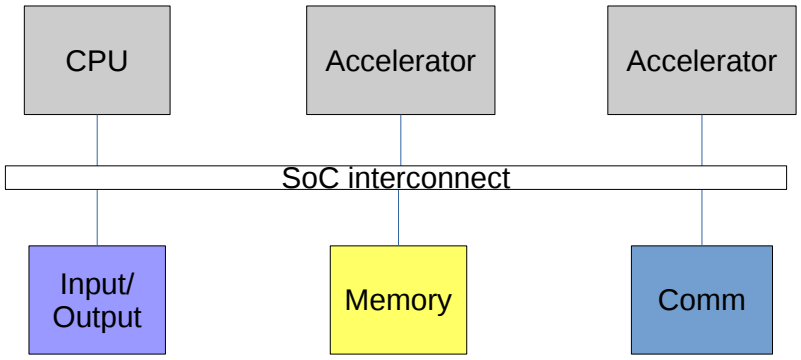


Figure 3.2: Architecture of a heterogeneous SoC

The choice of the types of the processing components to use in this system and the architecture is important and depends on the application and the constraints mentioned above.

3.2.1.2 Central Processor Unit

The processor is the core of any computer system. It executes the instructions of the main software program. In multiprocessor SoC we find more than one processor core. Two types of processors can be implemented in an SoC.

Hard-core processor It is a processor that is implemented on the final design in the silicon chip using a physical layout. It is optimized and validated by the designer but it can not be modified. It can achieve much faster processing speeds comparing

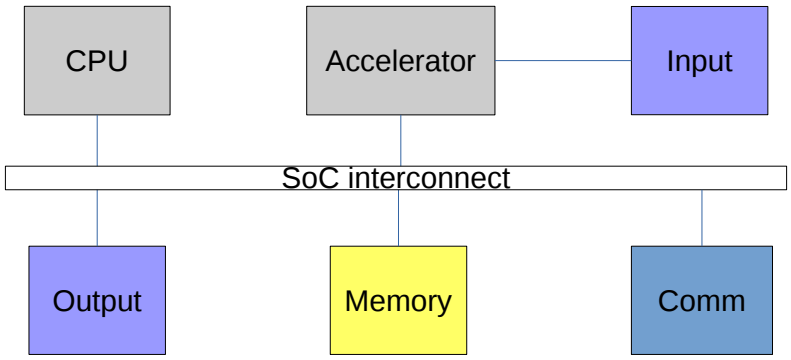


Figure 3.3: Architecture of a heterogeneous SoC with preprocessor circuit

to the soft-core processors since they are optimized for this. Examples of hard-core processors are Intel processors, AMD processors, ARM cortex-A processors, etc.

Soft-core processor It is a processor that is implemented on a programmable circuit(FPGA, ASIC, CPLD...etc), with the logic elements of that circuit. It is delivered as a synthesizable design using hardware description language like VHDL, Verilog,etc. [47]. The soft-core can be easily modified and customized by adding custom instructions and more features. However, the processing speed is limited when comparing with hard-core processors. Besides, it reduces the remaining resources on the programmable circuit for the implementation of other circuits. Examples of soft-core processors are: MicroBlaze from Xillinx, and NIOS-II from IntelFPGA.

3.2.1.3 Accelerator circuits

The acceleration circuit is a specialized circuit intended to accelerate some functions that take a lot of time on the CPU. These are capable of executing a large number of tasks in parallel, to speed up processing flows. Accelerator circuits can be classified in three categories [48]:

Programmable Processors: They are processors that can be software programmed to perform a task. This category mainly includes GPU (Graphics Processing Unit) and DSP (Digital Signal Processor). GPUs are widely used in SLAM, especially visual SLAM that uses computer vision algorithm. The parallel architecture of the GPU

and its high frequency of processing attracted the researchers to use them, not only to perform computer vision tasks but also as GPGPU to process all the parallelized tasks [26].

Dedicated circuits: A dedicated circuit is a circuit customized to perform a specific task in a hardwired manner. ASIC (Application-Specific Integrated Circuit) is a common example used to design customized circuits.

Reconfigurable circuits: They are circuits composed of logical elements that can be freely reconfigured to implement any logical function. They combine the customization features found in dedicated circuits and the flexibility of programmable processors. In this category we find FPGA and CPLD. In the SLAM domain, FPGAs are widely used. It has a low power consumption comparing to other types of circuits, and despite it has a low frequency limited by the FPGA fabric, the high customizability of the FPGA makes it possible to design a very specialized acceleration circuit. It is used usually to implement computer vision functions and matrix multiplication functions [37, 49]. It is used also to implement the whole SLAM system [31]

3.2.1.4 On-chip interconnect

An SoC is made up of several execution units. These units exchange data and instructions back and forth. This requires having an on-chip communication subsystem. This communication subsystem must take into account integrability in the SoC and support the transfer of data between multiple units. Two types of on-chip interconnects are mainly used

Bus-based communication This type of communication is based on a shared medium connected to the different components. It is the commonly used communication backbone in SoCs. There are two types of components communicating over shared bus: master and slave. A shared bus allows only one communication at a time. Only the master can initiate a communication over the shared bus. Usually, the only master is the CPU. If there are more than one master on the shared bus, an arbitrating system is

used to manage the communication and prevent bus contention. This architecture has advantages of simple topology and low area cost. However, its bandwidth is shared by all the cores on the chip which limits its scalability and the number of cores attached. [50] Some common buses for SoC communication are AMBA from ARM and Avalon from IntelFPGA.

Network-on-Chip Network-on-chip communication is based on data routing. The communication subsystem is a network of routers connected to each other. Each core on the chip is connected to a router via a point-to-point connection (Figure 3.4). The router is responsible for routing data to its destination. This type of interconnection is scalable and supports more connected cores than the buses. The bandwidth in this type of communication is not affected by the number of cores connected, since they do not share the same communication medium. [50].

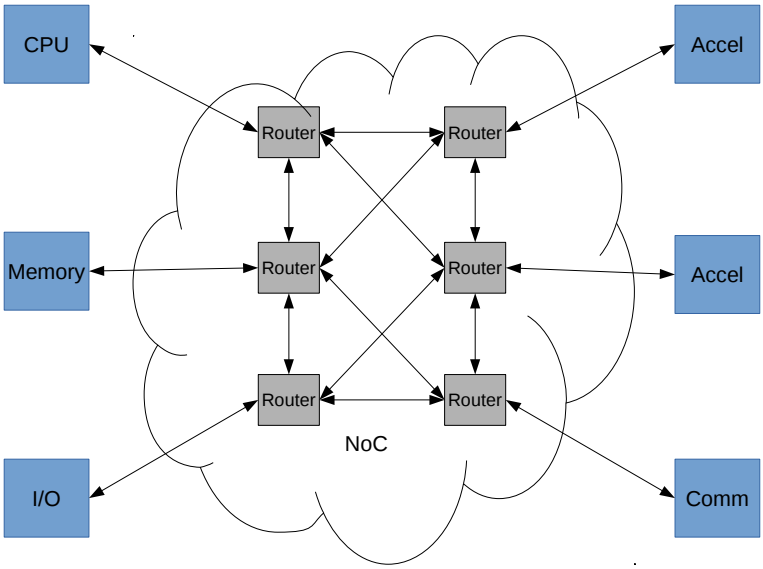


Figure 3.4: Network-on-chip

3.2.1.5 Technology selection

Reconfigurable circuits present a prominent solution for accelerating algorithms. FPGA are a reconfigurable circuits having an advantages over other parallel coprocessors.

The blocks of logic elements of the FPGA are used in parallel or pipelined to achieve a final architecture which exploits its full capacity. Besides, FPGA can be fully configured by users and offers better performance and reliability. However, FPGA may be considered an unappealing option when using the traditional logic design techniques to fully use the FPGA capacity. This techniques are long and complicated and require the usage of a hardware description language (VHDL, Verilog...). High Level Synthesis is a revolutionary solution that enhances the use of FPGAs. It is a technique that transforms high-level languages into logical elements aimed at exploiting the reconfigurable architecture efficiently to accelerate compute operations while conserving the use of resources.

In this thesis, we are interested in using FPGAs. Its main use is to design highly parallel circuits to replace functions with heavy execution on the processor. In addition to the circuit designed on the FPGA, a processor is required to run the main SLAM program. FPGA manufacturers, including IntelFPGA, Xilinix and Microsemi [51–53], have designed an SoC based on FPGA. This SoC-FPGA integrates a hard processor with an FPGA fabric in the same chip.

3.2.2 FPGA-based SoC

FPGA is a constantly evolving technology, especially in terms of logic density and speed. Modern FPGAs have over a billion logic gates. An entire system, with a soft-processor and accelerator circuits, can be implemented on the FPGA. Although clock frequencies have been improved, FPGAs operate at very low frequencies compared to hard-processors. Implementing a soft-processor on FPGA doesn't give the same performances as a hard-processor. FPGA manufacturers, such as Altera, Xilinx and Microsemi, have integrated hard-processors with an FPGA fabric in a single chip called SoC-FPGA. Xilinx embeds ARM processors (Cortex-A9) in the Zynq-700. IntelFPGA uses ARM processors (Cortex-A9) in the Cyclone V, Stratix 10, Arria V, Arria 10 and Agilex SoC circuits. Microsemi uses RISC-V processors in PolarFire SoC, and ARM Cortex-M3 microcontroller core in the SmartFusion circuits.

3.2.2.1 Why SoC-FGPA?

The FPGA offers great flexibility in the design of embedded systems, with low power consumption compared to programmable processors. Its flexibility can be seen in its customizability and reconfigurability. Customization makes it possible to design a tailor-made circuit with a customized, parallelized and pipelined architecture. The reconfigurability allows us to test and retest our system several times before the final product.

On the other hand, a hard-processor operates at a higher frequency than that of the FPGA which allows the software program to be executed with a higher performance than in FPGA with a soft-processor. In addition, a hard-processor saves the resources of the FPGA by avoiding its implementation using the FPGA fabric.

The integration of a hard-processor and an FPGA fabric in the same chip does not just make it possible to take advantage of these two components, but also makes it possible to reduce energy consumption and gain in terms of throughput between the CPU and the FPGA.

3.2.2.2 Cyclone V SoC architecture

Among the SoC-FPGAs that exist, we have chosen to use a hardware platform based on IntelFPGA's Cyclone V SoC. The Arria 10 is one of the low-cost 28nm technology SoCs produced by IntelFPGA company. Figure 3.5 shows a block digram of the Cyclone V SoC. It contains two distinct parts: Hard Processor System (HPS) and FPGA. The HPS is based on the ARM Cortex-A9 dual-core processor operating at 800MHz. The HPS integrates, in addition to the CPU, Memory controllers, interface peripherals, and PLL. The FPGA is clocked with a 50MHz clock. It can reach up to a frequency of 200MHz using PLLs.

The HPS and the FPGA communicate with each other using bridges. These bridges are 128-bit, 64-bit, and 32-bit AXI interfaces. The FPGA can access the slave buses of the HPS through the FPGA-to-HPS bridge. The HPS can access the FPGA slave buses through two types of bridges: a high-speed HPS-to-FPGA interface, used primarly to send data, and a 32-bits lightweight HPS-FPGA bridge used to send control signals.

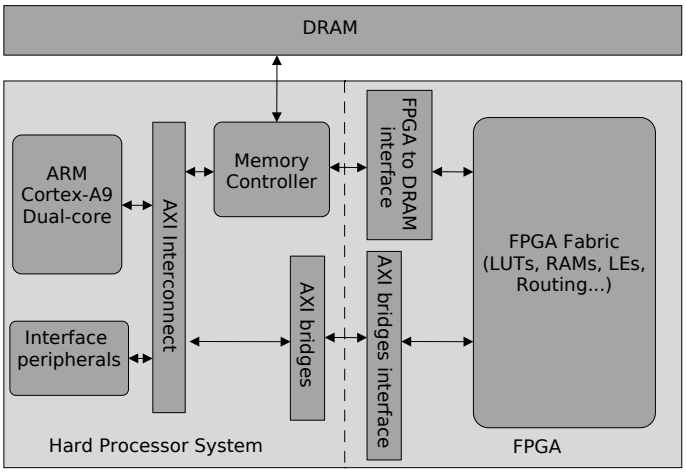


Figure 3.5: Cyclone V SoC Block diagram

The bandwidth of the of the interconnect bridges can reach 6400MB/s.

The SoC doesn't contain memory and it must be associated with external memory. The HPS contains a memory controller to interface the external memory. FPGA can access this memory using one of the six (6) FPGA-DRAM interfaces.

In this thesis work, we used a DE1-SoC development board as an evaluation platform. This board is built around the IntelFPGA Cyclone V SoC. The DE1-SoC board has two external memories to be used with SoC: a 64MB SDRAM to be used with the FPGA and 1GB DDR3 connected with the HPS. The board contains also a set of I/O interfaces like USB, I2C and GPIO. These interfaces can be used to connect a visual sensor (Camera, RGB-D...) or odometers. It contains also a G-sensor, that can be used as a proprioceptive sensor [54].

Table 3.1 resumes the specifications of the DE1-SoC, while Table 3.2 lists the available resources on the FPGA of the Cyclone V 5CSEMA5F31C6 chip used in DE1-SoC

3.2.3 Designing tools

FPGAs have advantages in terms of parallelism and power consumption. But the description of FPGA-based architectures remains relatively complex compared to

Table 3.1: DE1-SoC board specification

	Specifications
FPGA Device	Cyclone V 5CSEMA5F31C6
FPGA SDRAM	64MB SDRAM
Processor	ARM-Cortex-A9 Dual-Core
HPS Frequency	800MHz
HPS SDRAM	1GB DDR3

Table 3.2: Available resources of the Cyclone V 5CSEMA5 chip

Resources		Cyclone V 5CSMEA5
Logic Elements (LE) (K)		85
ALM		32075
Registers		128300
Memory(Kb)	M10K	3970
	MLAB	480
Variable-precision DSP Blocks		87
18 x 18 Multipliers		174

programmable processors. Unlike programmable processors where a program is written in a high-level programming language and then compiled for execution, an FPGA-based architecture is described using low-level hardware description languages and then synthesized for implementation in an FPGA. Since the appearance of programmable circuits, design tools have undergone an evolution which aims to facilitates the hardware description and to move towards high level languages.

3.2.3.1 Low-level hardware description languages

A Hardware Description Language (HDL) is used to describe the structure and behavior of electronic circuits. From this description, the electronic circuits can be analyzed, simulated and synthesized to be implemented on the configurable circuit. Unlike pro-

gramming languages, HDLs make it possible to describe competitive behavior at the instructional level, and explicitly include the notion of time.

The most known and used HDLs are VHDL and Verilog, both introduced in the 80s. VHDL is strongly typed while Verilog is weakly typed and influenced by the the C programming language.

The use of description languages is a complex task and often takes a relatively long time to develop and verify. The complexity and development time increase with the complexity of the system. This requires design help tools and going to high level languages.

The development environments of the FPGA manufacturers have a graphical environments that help design a system from the IP cores that exist in the library. Examples of such graphical environments are IntelFPGA's Qsys System Integration Tool and Vivado IP Integrator and Xilinx. These tools are used to generate the HDL code from the graphic description of the system. However, adding a new IP core requires writing it with HDL.

3.2.3.2 High-level synthesis

High-Level Synthesis (HLS) consists of the automated generation of HDL descriptions from an algorithmic description performed using a high-level language such as C/C ++, Matlab. High level synthesis saves development time by describing our circuit in high level language. Simulation and verification are done at the high level. It also allows developers not experienced in HDL to design their circuits without knowing the details of the target.

Regarding existing high-level synthesis tools, Vitis HLS from Xilinx [55], Intel HLS compiler from IntelFPGA [56]. MATLAB and Simulink have also their High-level synthesis tools called HDL coder [57].

In addition to high-level synthesis tools, the OpenCL standard is also used as a high-level design language. The goal of adopting OpenCL was to standardize design on heterogeneous architectures. Next, we will see OpenCL framework that we will use it on our work.

3.2.4 OpenCL approach

OpenCL is an open standard for parallel programming of heterogeneous systems using a C-based language [58]. It uses a unified programming language to program these devices and offers an API that controls the execution of the program on the acceleration devices. The heterogeneous system contains one or more different computing devices such as CPUs, GPUs, DSPs, FPGA...etc. attached to a host CPU. OpenCL is portable which means that the same kernel code can be used to run on multiple device types.

An OpenCL program is divided into two parts: the host code, and OpenCL kernel code. The host code is the code that is running as software on the CPU, it is written with C/C++. The kernel code, written in OpenCL, is the part of the code to be implemented and executed on the compute device. OpenCL offers a C/C++ API to use in the host code. This API is used to control the execution of the kernel and the data transfer from and to the processing device.

Figure 3.6 shows a model of a generic OpenCL system. An OpenCL compute device contains multiple compute units, each compute unit comprises multiple processing elements.

From the kernel point of view, the workload is divided into work-groups, each work-group is in turn divided into work-items. The work-item is described by the kernel program. Work-items are running on processing elements, and work-groups are running on compute units.

Two types of OpenCL kernels are existing:

Single task kernel: the kernel is executed by only one work-item. This is common on FPGA where the parallelism is achieved using pipelining.

NDRange kernel: the kernel is executed on multiple work-items, and the parallelism is achieved using data-parallelism.

From memory point of view of the system, OpenCL describe 5 different types of memory that can be used with OpenCL:

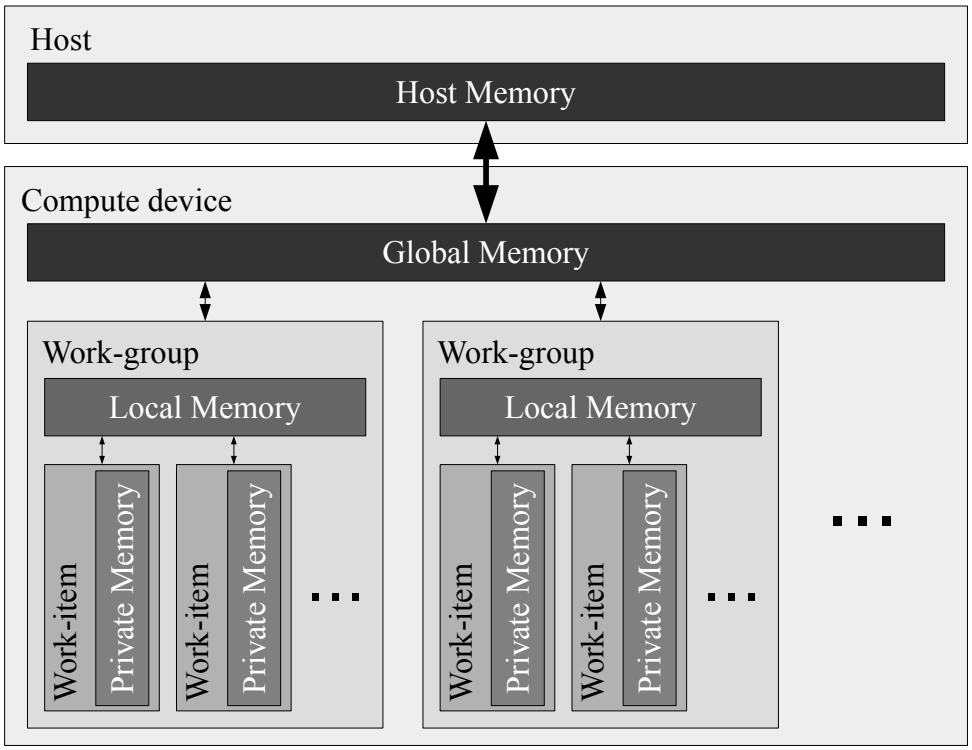


Figure 3.6: Model of generic OpenCL system

Host memory: It is used by the host program (The CPU program) and it is not accessible by the kernel. This memory is cacheable and managed by the operating system

Global memory It can be accessed by both the host program and the OpenCL program, it is accessed by all the work-groups

Constant memory: It is a type of global memory with read-only access from the kernel.

Local memory: it is a memory shared between the work-items of a work-group

Private memory: It is a memory that can be accessed only by the corresponding work-item.

In single task kernels, the local memory and the private memory are the same since there is one work-item.

3.2.4.1 OpenCL for SoC-FPGAs

In the FPGA domain, Altera was the first constructor to provide a complete SDK for OpenCL [59]. Xilinx, also, released its SDK for OpenCL integrated with Vivado environment called SDAccel [60].

On programmable processors, the architecture of the device is known and the OpenCL kernel is compiled at run-time. However, in the FPGA the architecture is configured according to the kernel before run-time. This requires an offline compilation of the FPGA circuit.

The architecture of the OpenCL circuit on the FPGA can take two forms according to the type of kernel used. In the NDRange kernels, the OpenCL circuit has a parallel architecture providing data parallelism. In a single task kernel, the OpenCL circuit has a pipelined architecture.

The memories architecture for an OpenCL system in the SoC-FPGA is shown on Figure 3.7.

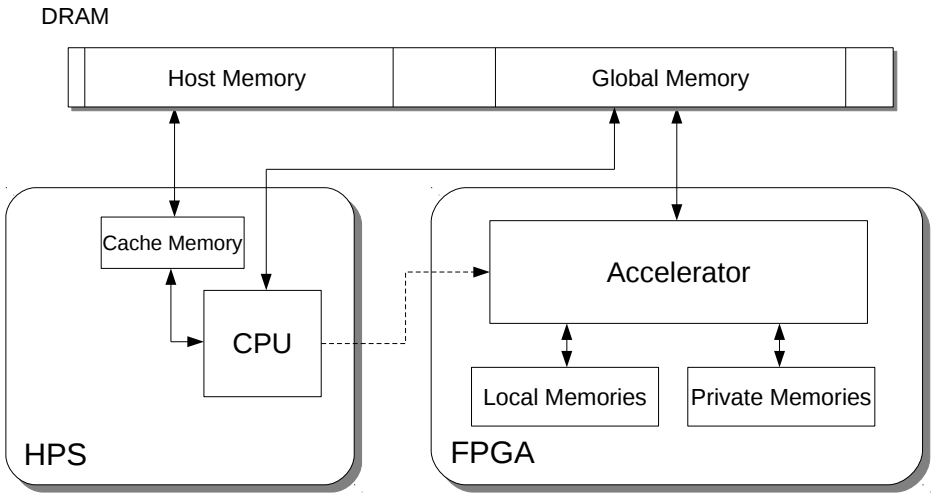


Figure 3.7: Memories architecture for OpenCL system on SoC-FPGA

The host memory is a cacheable memory and it is the region in the DRAM used by the HPS. The global memory is the memory region dedicated to the FPGA device on the DRAM, it is a non-cacheable memory accessed by the kernel. It is characterized

by a large size compared with other OpenCL device memory types, however, it has low bandwidth and causes an undefined stall behavior. Local and private memories are implemented with RAM blocks and registers on the FPGA chip. They are characterized by smaller size and zero delay access comparing with the global memory.

3.2.4.2 IntelFPGA® SDK for OpenCL™

In this work as it is mentioned earlier, we use a DE1-SoC board, containing a Cyclone V SoC from IntelFPGA, as target platform (Figure 3.8). IntelFPGA has released its SDK for OpenCL development on FPGAs. The FPGA programming flow of the IntelFPGA SDK for OpnCL is shown in Figure 3.9.

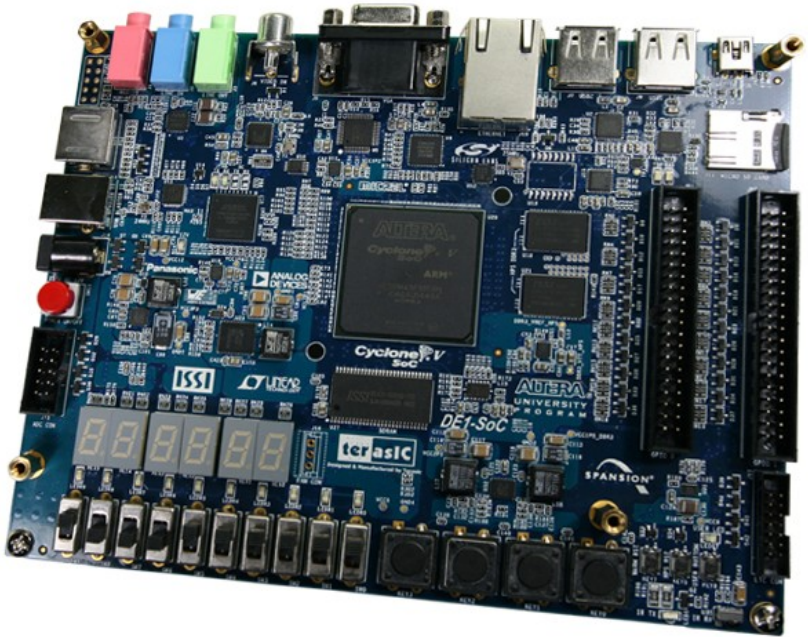


Figure 3.8: DE1-SoC DEveloppement board

The programming flow contains two parts: the host part and the FPGA part. In the host part, IntelFPGA SDK for OpenCL contains the libraries of functions necessary to control and communicate with the kernel on the target. These functions are used with the host code. The host code is then cross-compiled according to the architecture of the target. In our case, it is compiled to run on the ARM processor on the HPS.

On the FPGA part, IntelFPGA SDK for OpenCL contains a tool called AOCL (Altera offline compiler). AOCL creates an executable file (.aocx) containing the hardware

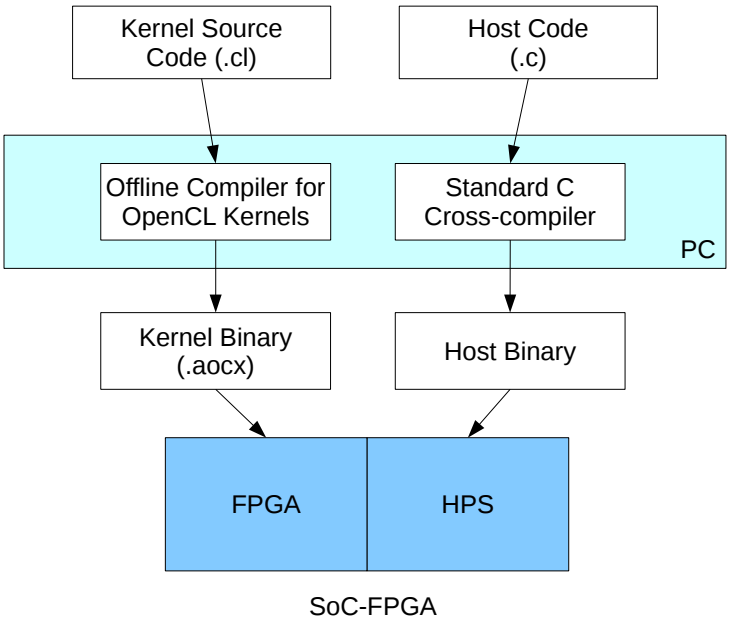


Figure 3.9: IntelFPGA SDK for OpenCL FPGA Programming Flow

configuration and the information necessary at runtime. This file is used to configure the FPGA and provides the necessary information to the host application to create program objects used at runtime. [61].

The process of creating the FPGA hardware configuration file can be done on a single step or several steps, depending on the complexity of the circuit.

In our case, we need to optimize and improve the performance of the OpenCL application. A multi-step design compilation is used. The multi-step design flow is showed in Figure 3.10, it is based on an iterative process, and has four main steps:

Emulation: This is the first step in the programming flow. The aim of this step is to verify the functional correctness of the kernel. To emulate the kernel, AOCL compiles the OpenCL code to be run on x86 host system (PC-Like machine). The compilation is done on the high-level to emulate the behavior of the kernel, and it doesn't depend on the target. The host program code must be also compiled to be run on the same system. This allows us to verify and debug the kernel before implementing it on the FPGA.

Intermediate Compilation: In this task, AOCL checks for syntactic errors and gen-

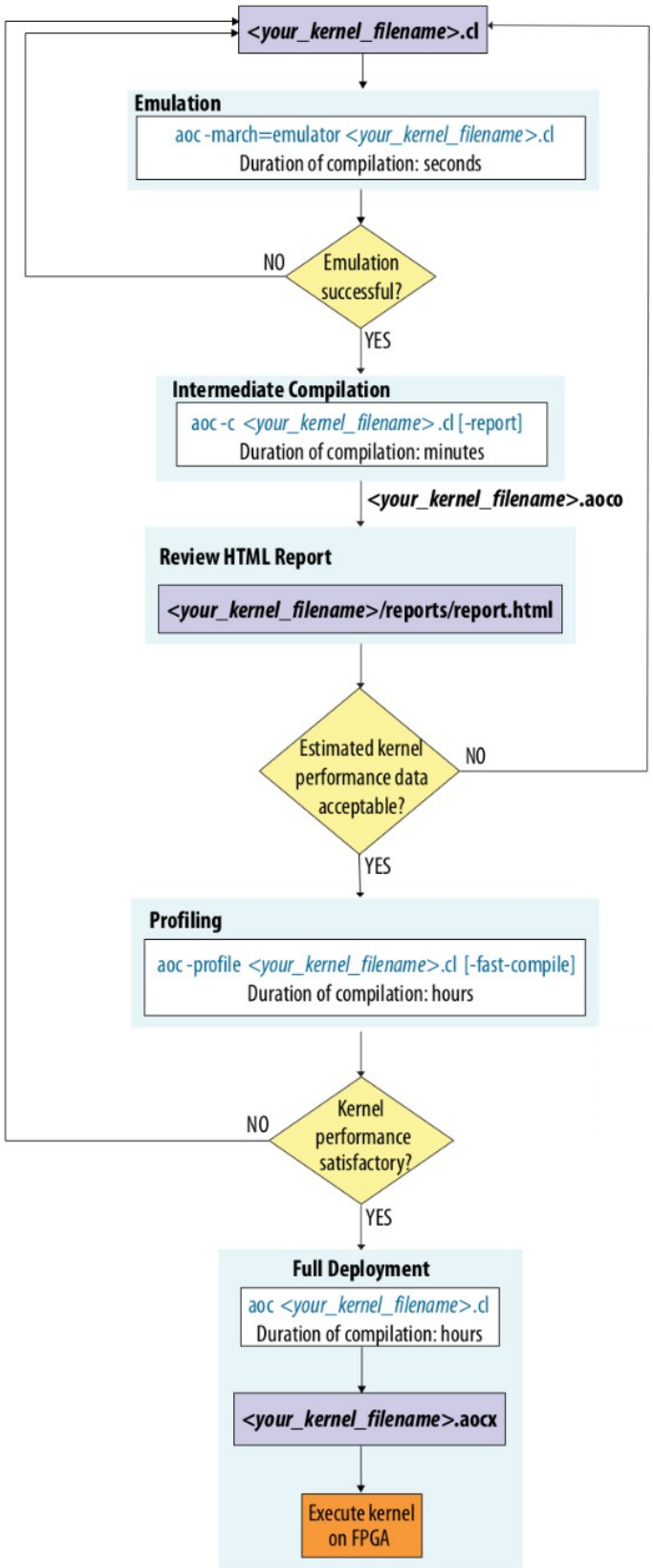


Figure 3.10: FPGA design-flow using AOCL

erates an object file containing information about the kernel. The hardware configuration file is not generated in this step. At the end of the compilation an optimization report is generated in HTML form. This report is specific to the implementation target, it estimates the resource usage of the FPGA, loops analysis, latency estimation, bottleneck, and stalls summary. Based on this report the kernel may be restructured again for more optimization. Using the optimization report saves time since the hardware is not built.

evaluation: Before generating the final kernel, it is necessary to evaluate the real performance on the hardware. AOCL can add performance counters to the HDL code of the harwadre configuration file (*.aocx*). These counters collect performance information at runtime and generate a *.mon* file. The OpenCL Dynamic Profiler measures the performance using the collected data from the performance counters during the execution. The OpenCL Dynamic Profiler GUI reports the total time of the kernel execution and information about the bandwidth, stalls, and efficiency of memory accesses. A kernel restructuring may be necessary depending on the results of the Dynamic Profiler

synthesis: After this step, AOCL performs a full compilation and generates an executable file (*.aocx*) without performance counters. The hardware configuration file is used by the host program to configure the FPGA, communicate and control the FPGA kernel. It contains the necessary informations to be used at runtime.

3.2.5 Conclusion

In this section, we have chosen, among the existing technologies, SoCs based on FPGAs or SoC-FPGAs. This choice is justified by the advantages offered by the combination of a hard processor and an FPGA in the same chip. The CPU executes the software program with a high execution frequency, while the FPGA allows the configuration of an acceleration circuit for heavy functions on the processor. We chose an implementation method based on OpenCL to minimize the design time and facilitate

porting to other targets. The methodology of designing an OpenCL application is described. In the following section, we see the software design for our rapid prototyping platform.

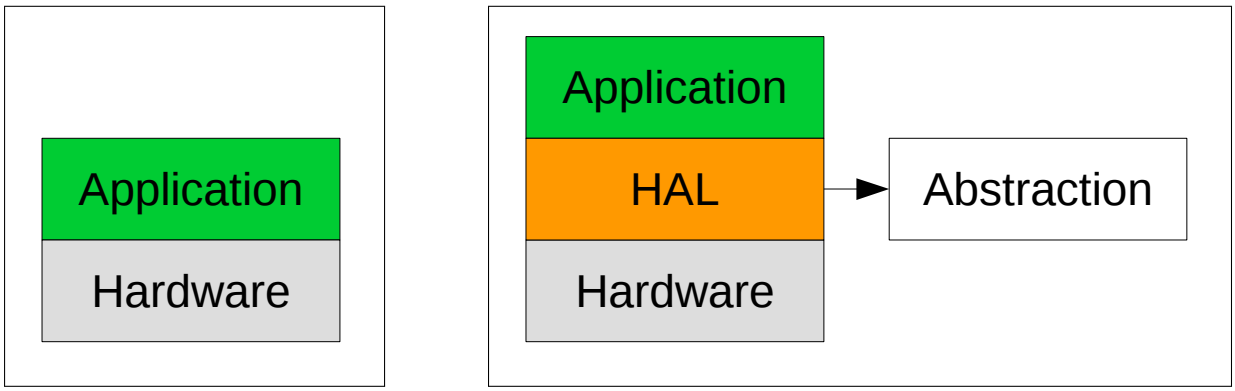
3.3 Software Design

An embedded system is made up of two essential elements: software and hardware. A design platform for an embedded system must support the design of the software and the hardware. In the previous section, we dealt with the hardware part of the platform. In this section, we will deal with the software part and end up choosing our software platform. Indeed, the software design process consists of developing a program in a machine language or in a high-level language. This program will be executed on the CPU. Software design should take into consideration design constraints including software size, development time, retargeting, evaluation, etc. The use of tools to automate the system building allows us to reduce the build time and to customize our software system, which also implies a reduction in its size.

3.3.1 Implementation approaches

In software design, there are three possible approaches to implementing an embedded application. The first approach is bare-metal implementation (Figure 3.11a). In this approach the application is implemented directly on the hardware without the use of an intermediate layer. In this approach, the embedded application runs faster and uses less memory space. In addition, access to peripherals and I/O is direct without any redirection to subroutines. But in this type of implementation, the management of peripherals, memory, interrupts, and errors is not assured if it is not implemented in the application. These applications are generally written with low-level languages like C/C++ and assembler, they are generally used in microcontroller programming. The bare-metal applications are very dependent on the development platform which means that these applications are very rarely portable.

The second approach is to implement the embedded application on a Hardware



(a) Bare-metal application (b) Application on HAL

Figure 3.11: Software Implementation approaches

Abstraction Layer (HAL) (Figure 3.11b). This layer is considered an abstraction of the hardware, it contains the functions necessary to facilitate access to peripherals and I/O.

The third approach is to implement on an embedded OS (Figure 3.12). An operating system provides a variety of solutions such as memory management, scheduling, hardware abstraction, tools, libraries, services, real-time, etc. In addition, it facilitates the portation to other hardware platforms supported by this OS. There are several OSs that are used in the embedded field. Among the most famous OS, we can mention: Linux, Windows Embedded, FreeRTOS, VxWorks. Linux is an operating system that is widely used in this field, considering the advantages it presents. In the next section, we talk about Linux for the embedded.

3.3.2 Embedded operating system

Embedded systems are becoming more and more complex and are evolving towards architectures containing powerful computers in addition to several peripherals. This evolution allows the implementation of complex software applications on these systems. In this case a system that manages the scheduling of tasks and access to hardware resources for this application is necessary. Hence the interest of using an operating system.

The role of an operating system in an embedded system is to abstract hardware

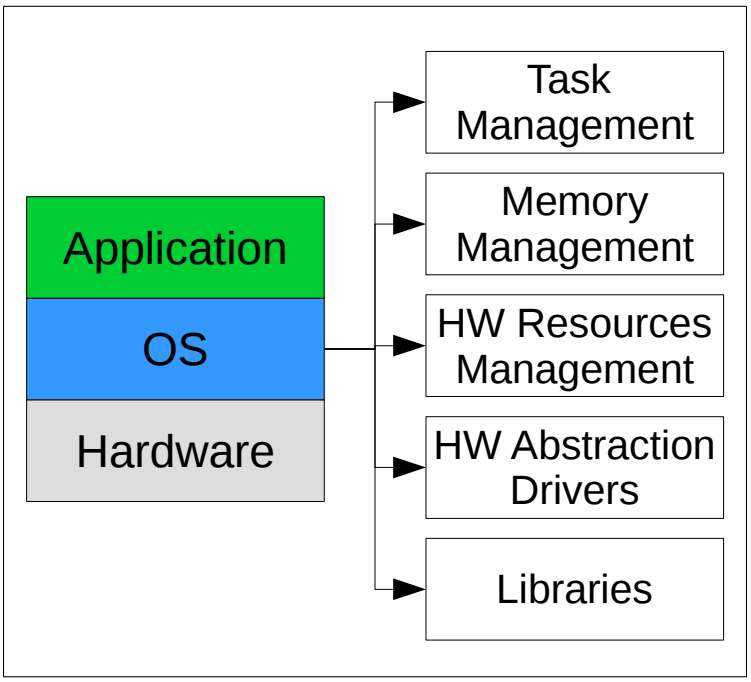


Figure 3.12: System with OS

resources so that the application has a simplified view. This abstraction has two different ways. The first concerns the processor, the role of the OS is to share the computing power of the processor. This sharing is carried out thanks to a "scheduler" which selects a given task at a given moment for immediate execution. The second concerns access to peripherals. The OS must share access to devices and provide a high-level interface to applications. The drivers provide access to peripherals and perform the necessary configuration and data exchange [62].

Embedded OSs are very constrained by the hardware platform which has limited resources, such as computing performance, memory size, energy consumption, storage memory, etc. At the same time, They must ensure the execution of the embedded application in real-time and with the required performances.

One of the most used OS in embedded system is Linux, the famous open-source OS developed initially by Linus Torvalds.

3.3.2.1 Embedded Linux

Linux is a Unix-like multitasking operating system based on the Linux kernel. It was originally developed for personal computers based on Intel x86 type processors, but has since been ported to more platforms than any other operating system. Among the supported platforms, we found many embedded platforms like PowerPC, ARM, NIOSII, MicroBlaze... Linux conforms to the POSIX (Portable Operating System Interfaces) standard, which means that sources developed under Linux can be easily compiled on other POSIX compatible operating systems. [63]

Linux is free software which means that the source code is available for free, with the right to modify and redistribute and use without restriction while respecting the GPL license (General Purpose License).

In the early 2000s, Linux was already widely used in the server world. Thanks to its reliability, code availability, and free redistribution, the Linux community has been able to develop it for use in industrial and embedded solutions. Today several embedded systems are based on Linux such as set-top boxes, smartphones, smart-TV (Android)

3.3.2.2 Why Choosing Linux?

In addition to its Reliability and performances, Linux has some characteristics that make it a better choice for use in embedded systems.

The First characteristic cited earlier is being **Free** and **Open-source** software. This makes it possible for a developer to copy the source code and modify it, then redistribute it without any loyalty. Such characteristic allows a development basing on previous works and not from zero. The users of the systems can read the code source and be sure that there is no risk to its personal information security. Having no royalty is also a problem since there is no person assumes responsibility for damages resulting from its use.

The second characteristic is having a **large community**. This means that Linux is the result of collective efforts. The bugs are fixed fast, and the updates are released constantly. The large community also offers help via forums.

The Linux community leads to the development of the system to adapt it to other architectures. Linux today is ported to a very large number of processors and hardware architectures, including low power processors. **Portability** and **adaptability** are important characteristics of Linux. [64]

Since Linux is Free and Open-source, modification of the source code is permitted. This makes it **customizable**. The Linux customizability is one of the strong points to use for embedded systems. Only the needed modules and functions are included in Linux. This reduces the usage of memory and CPU. A Linux system of few megabytes can be built.

3.3.2.3 Structure of Linux system

Linux is a Unix-Like system and they share a similar architecture. It is composed essentially of a kernel, executables, libraries, and bootloader to boot the system.

Bootloader: It has two essential functions: hardware initialization and kernel and filesystem loading. First, the bootloader initializes the hardware devices essential for booting such as memory and disks. Then it launches the kernel that exists on the hard disk.

Kernel: The kernel is a stand-alone executable file responsible for performing essential system functions such as managing the memory and the scheduling of tasks as well as interfacing between hardware and applications, using device drivers. It is the unique interface between the system and the hardware. The kernel is extensible, modules can be added if needed. These modules can be device drivers or be linked to higher level generic support such as SCSI support. The use of modules optimizes system memory at a given time because an unused driver can be unloaded, freeing its memory. Likewise, the use of modules will allow a dynamic addition of peripherals without restarting the system.

Libraries: Libraries contain functions needed by the applications. There are main libraries that contains essential functions of Linux. Other libraries can be added to

the system when needed. There are two types of libraries Static libraries and dynamic libraries.

Applications: They are in the upper layer. They can be commands delivered with the system or developed for specific needs.

Figure 3.13 shows the layered structure of the Linux system. The kernel is the lowest layer and applications are the highest layer.

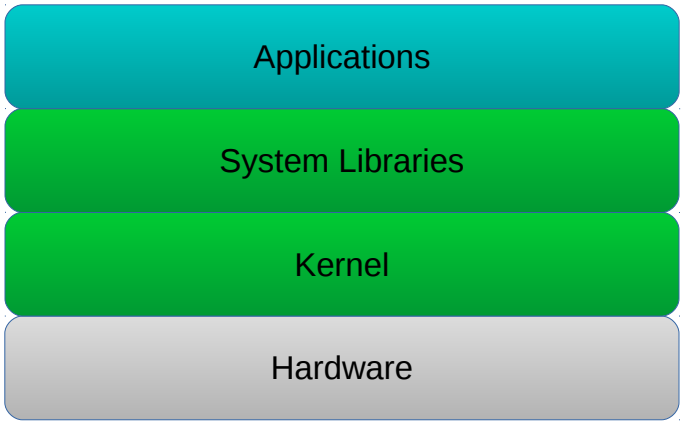


Figure 3.13: Layered structure of Linux system

3.3.3 Software libraries

A library is a collection of pre-compiled functions. These functions are called by the applications to perform a specific task. Libraries improve reusability and modularity. They save time by providing reusable functions, data structures, classes, etc. The programmer doesn't need to rewrite all these functions, fix bugs, and update them, These tasks are done by other developers or by the community. Instead, he has only to call the needed functions from the library.

A library is collection of archived object files with an index (table of content) for fast symbol access. At compile-time, the linker search for symbols and functions in the library. If they're found, the program is linked to to it. There are two types of libraries, static libraries and shared libraries:

Static libraries: A static library is an archive file containing a collection of raw object files with a table of content. When this library is included during a program linking, the linker adds all the code and data, corresponding to symbols used in the source code, to the final program. The advantage of using static libraries is that they are easy to create and use and run faster(no need to search and load the needed functions). The program created with static libraries does not require to install the needed functions in the target system. However, the use of a static library in a program present a number of software maintenance and resource utilization problems. If the library is updated or fixed, the program linked to it need to be rebuilt too, to include these changes. The program file includes all the needed functions, this means that its size will be large, and take a considerable amount of disk space. Executing many programs with shared functions is a waste of disk space and memory [65].

Shared libraries: To resolve maintenance and resource problems encountered with static libraries, shared libraries are used. In this case, at compile-time, the compiler search for the symbols in the library without copying its content to the program file. At run-time, a dynamic linker searches for needed symbols in the library to load them into memory and perform the run-time linking task. Run-time linking allows easier library maintenance. Updating the shared library doesn't require relinking or rebuilding the applications, the symbols are kept unchanged so that can be found the next time by the applications. In addition the size of the program file will be reduced since the shared symbols are not copied. Shared libraries, from their names, are shared between applications in run-time. If many applications need to use a single library, it is loaded once in physical memory and used by all the applications, which saves memory by not loading many copies of the same library. However, the loading libraries at run-time and dynamic linking will influence the execution speed. The application requires that the shared library to be implemented in the target system, since it is not copied with the program file [65].

In the SLAM domain, many known libraries are used, for computer vision, mathematical calculations, linear algebra, graph optimization...etc. These libraries must be

offered by our platform to be implemented when needed by the SLAM application. One of the works that evaluates SLAM libraries is that of Hertzberg *et al.*. [19] They evaluated some open-source libraries used in visual SLAM for three modules: feature detection, data association, source components. The work shows that open-source libraries are mature enough to build a visual SLAM system. Next, we see some of the most used open-source libraries by the SLAM applications.

3.3.3.1 OpenCV

OpenCV (Open Source Computer Vision) is an optimized and open source computer vision library specializing in real-time image processing. It was originally developed by Intel to advance CPU intensive applications in computer vision. It is a cross-platform library written in C / C ++ and python. The OpenCV library provides many very diverse functionalities allowing to create programs from raw data to go up to the creation of basic graphical interfaces. It provides an easy-to-use computer vision environment for quickly building sophisticated applications. OpenCV provides a set of image acquisition and processing functions. These functions include camera calibration, image filtering, landmarks extraction and detection, motion estimation and target tracking, object detection and recognition, and machine learning. We also find projects that aim to develop accelerated implementations with CUDA and OpenCL. [1, 66]

OpenCV is used by many SLAM applications to calibrate and manipulate images and features, We cite for example MonoSLAM [22, 67], ORB-SLAM2 [68, 69], LSD-SLAM [70, 71]...etc.

3.3.3.2 MRPT

MRPT is a cross-platform and open source C ++ library that aims to help researchers design and implement algorithms related to the field of robotics. These algorithms are in the area of simultaneous location and mapping (SLAM), computer vision and motion planning (obstacle avoidance). It was developed for first time in 2004 at MAPIR lab at the University of Málaga [72]. The libraries include classes for easily manag-

ing 3D geometry, probability density functions (pdfs) over many predefined variables (points and poses, landmarks, maps), Bayesian inference (Kalman filters, particle filters), image processing, path planning, and obstacle avoidance, 3D visualization of all kind of maps (points, occupancy grids, landmarks,...), and a sort of drivers for a variety of robotic sensors. A list of SLAM works that used MRPT can be found here [72]. Among these, we can cite Blanco *et al.* [73] in RO-SLAM, Moreno *et al.* [74] in particle filter-based SLAM, Blanco *et al.* [75] ...etc

3.3.3.3 Other Libraries

Many other libraries are used in SLAM programs. Next is a non exhaustive list of these libraries:

Eigen3: It is a C++ open-source library of template headers for linear algebra. It is used in matrix and vectors operations, geometrical transformations, numerical solvers and related algorithms.

SuiteSparse: It is a set of open-source C/C++ libraries developed by Timothy Davis [76]. It implements a number of sparse matrix algorithms.

g2o [77]: It is an open source C++ framework for optimizing graph-based non-linear errors by solving non-linear least-square problems.

Boost: It is a set of C++ libraries that provides support tasks and structures such as linear algebra, image processing...etc.

3.3.4 System builders

Our platform is based on a Linux system, it should contain all the components needed to build an entire system running an embedded SLAM. To build this system, We have multiple choices [78]:

1. Use a **pre-build binary distribution** such as Debian, Ubuntu, or Fedora. They are easy to set up and come usually with a package manager to facilitate installation and updating packages. However, they are not very flexible on package

configuration and support only a few architectures. The system is pre-built and rebuilding it is not easy especially for developing a device driver.

2. Build all system components **manually**. Despite the high flexibility this method offers on system and package configuration, it is a painful task to deal with complex cross-compilation issues and track all the intra-package dependencies. In addition, it cannot be reproducible because it lacks build automation tools. This makes it an inefficient method.

3. Use an **automated build system**, that builds the entire system from source. This method is situated in the middle between the two previous methods. It is flexible and can be easily configured. It allows building a custom system based on Linux. The build automation tool handles most of the cross-compilation issues and track all the intra-package dependencies automatically. The system is easily reproducible and supports a lot of architectures. We can cite as an example: Buildroot, Yocto, OpenEmbedded...

Among these solutions, the last solution is by far the most effective. The automatic build and rebuilding of the system as well as the possibility of configuration and customization all support the goal of the rapid prototyping platform. This solution the one we will use in our platform, Such a tool allows us to produce the components that we can install on the target:

- The image of a bootloader like U-Boot;

- The static kernel image often named zImage or uImage as well as the associated "device tree" files (DTB).

- One or more images of the root filesystem.

The two main Linux system builders for embedded systems are Buildroot and Yocto.

3.3.4.1 Yocto

Yocto or (Yocto project) is a collaborative open-source project that aims to produce tools and processes allowing the creation of Linux distributions intended for embedded systems. It supports many hardware architectures. It is based on OpenEmbedded build system and BitBake a make-like build tool. OpenEmbedded is a set of metadata made up of configuration files, classes, and recipes describing the tasks to be carried out to build the packages and binary images as well as their dependencies.

Yocto uses the notion of layers. A layer can add support for a given hardware (BSP: Board Support Package), but also software components. The layers are represented by metadata grouped in directories each corresponding to a layer. Yocto allows you to create binary packages as in a classic distribution. The formats taken into account are RPM, DEB, and IPK

- **Advantages**
  - Large Community (Support, training, experienced engineers...)
  - It has backing from many influential companies since it is widely used in the industry.
  - It is expandable through layers
  - It supports a wide range of hardware
  - It is highly flexible and customizable.
  - Updating, adding, and removing packages from a running system is possible without a need to rebuild the whole system.
- **Disadvantages**
  - It is hard to learn and master
  - Produces a large image system
  - The configuration is done using files, and there is no interface like *makeconfig*.

3.3.4.2 Buildroot

Buildroot is a set of Makefiles and patches that simplifies and automates the process of building a complete and bootable Linux environment for an embedded system. It builds the required cross-compilation toolchain, generates the bootloader, compiles the Linux kernel image and creates the root file-system for the targeted embedded system. It supports many hardware architectures. It focuses on simplicity and minimalism. Extensions and features can be added to Buildroot using simple Kconfig and Makefile files. It does not support package management and prefer the use of static libraries. Generally, the produced system image is as small as possible.

- **Advantages**
  - It focuses on simplicity, simple to learn, to understand and to contribute
  - It produces smaller system images by disabling all optional build-time settings.
  - It can be configured using a graphical interface like *menuconfig*.
  - It supports many hardware architectures and contains many configuration examples of know boards.
- **Disadvantages**
  - No package management. This makes it difficult to update, install, or remove a package in a running system
  - A change on components of the system requires rebuilding all the system, no partial rebuild is possible.

In our work, we used Buildroot to build our Linux-based system.

3.3.5 Buildroot

Buildroot is an embedded Linux build system. Its purpose is to simplify and automate the process of building an embedded Linux system. It is based on simple and

well-known tools developed by the Linux community: *Kconfig* for the configuration interface and language, and *Makefiles* for The build logic. These technologies are familiar to all embedded Linux developers which makes it simple to use and easy to understand, learn, and develop. The build process is fast by doing only the necessary tasks. By default, it doesn't integrate runtime package management system like dpkg and rpm, to be suitable for small and medium embedded systems.

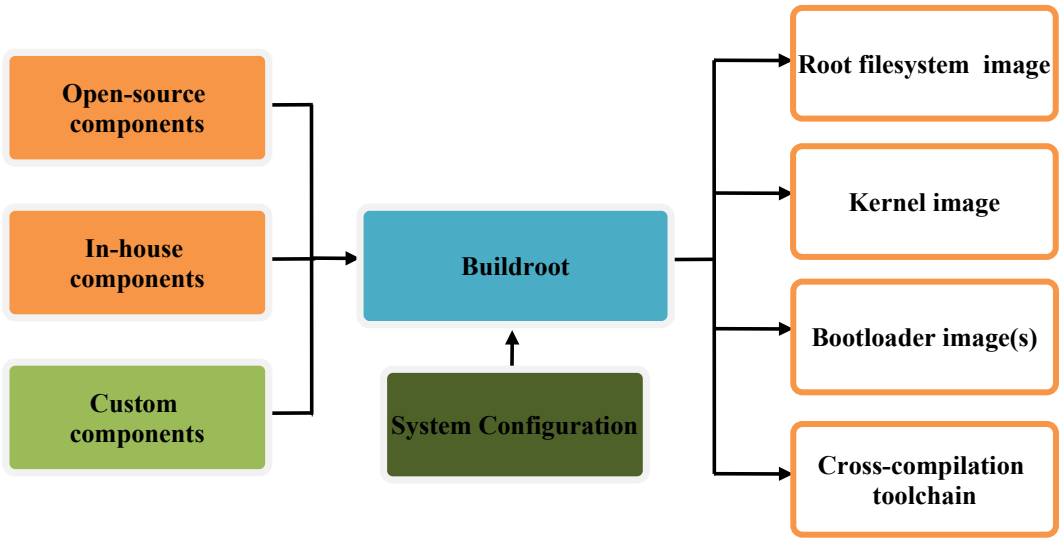


Figure 3.14: Schematic representation of Buildroot

Figure 3.14 shows the schematic representation of Buildroot tool. Buildroot has a list of open source components that can be downloaded from the Internet. This list contains the recipes for downloading, configuring, and building these components. In addition to these open source components, proprietary and custom components can also be added to this list. These components are mainly: cross compilers, Linux kernel, host packages, and target packages. Host packages are the packages needed by the host to build the target system, while target packages are the packages that will be executed on the embedded target.

A Buildroot based project is structured in directories. These directories are:

- `toolchain/`: this directory contains information and recipes for all software related to the cross-compilation toolchain

- `arch/`: It contains the definitions for the supported processor architectures.
- `package/`: contains configurations and recipes to download and build user-space tools and libraries. These informations are inside directories.
- `linux/`: This directory is contains files for Linux kernel.
- `boot/`: It contains files for bootloaders supported by Buildroot.
- `system/`: It contains support for system integration, e.g. the target filesystem skeleton and the selection of an init system.
- `fs/`: contains informations and recipes for software related to the generation of the target root filesystem image.

Buildroot must be configured in order to be able to build the target system. The configuration contains information on the properties of the target's architecture, compilation options, information on the kernel to compile, the packages to install, the type of final system images, etc. Buildroot can be configured in graphical mode with *menuconfig*. The configuration can be stored in a file to be used later.

Once the configuration is done. Buildroot creates the output directories (`staging/`, `target/`, `build/`) inside `output/` directory. Before building the system image, Buildroot downloads the cross-compilation toolchain that will be used to compile the needed package for the selected target. After building the system components, the root filesystem is created as well as the kernel image and the bootloader optionally if it is required.

3.3.6 Debugging and profiling

In the development phase of an application, the appearance of errors is almost inevitable. Many errors can occur during the integration or the execution of the application on the embedded target. To resolve these errors, a debugging tool is essential. this tool allows you to put breakpoints in the program and run it step by step to analyze and identify runtime errors. After resolving all runtime errors, a profiling

tool is used to identify bottlenecks. The bottlenecks present functions in the program that take a long execution time. These functions can be accelerated to improve the program performance.

To debug and profile an application running on Linux there are many tools that can be used.

3.3.6.1 Debuggers

Local debugging: gdb The GNU Debugger (GDB) is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Go, and partially others GDB offers extensive facilities for tracing and altering the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.

Remote debugging: gdb-server GDB offers a possibility of remote debugging of embedded systems. On remote debugging GDB runs on the host machine while the program being debugged runs on the target machine. GDB can communicate to the remote target that uses GDB protocol through a serial device or TCP/IP. gdbserver is used on the target to remotely debug the program without needing to change it in any way.

Manual debugging In this type of debugging, we use the print function to print some intermediate variables on the screen. This allows us to locate the error and identify the instruction or variable that caused it. Even if this method appears to be tedious, but sometimes remains the last solution to debug the program.

3.3.6.2 Profilers

Program profiling is a form of dynamic program analysis that measures some program metrics, for example, the memory usage or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. The

profiling information used to assist program optimization, and performance engineering. Profiling is done by instrumenting either the program source code or its binary executable form using a tool called a profiler. Profilers may use a number of different techniques, including event-based, statistical, and instrumented methods.

Event-based profilers: They are also called tracing profilers. The event-based profilers collect data on well-defined events. These events can be entering/leaving function, thrown exceptions,etc. The profiler consists of an event tracker and a performance monitor. The event tracker detects and manages events, while the performance monitor collects information about the program state (run-time, memory usage, etc.) based on events that occurred during the program execution. The result is information accumulated on an event basis [79]. We can cite in this type of profilers **perf** the performance counter of Linux.

Statistical profilers: Also called sampling profilers. These type of profilers polls the target application on regular intervals. To poll the application ,the profiler use OS interrupts, and at each interruption, it determines the function that is being executed and rises the sample count of that function. At the end, it reports the number of collected samples for each function. This report provides an estimation of the time spent by the application inside each function. The function that has more sample counts is the function that takes much time. This may be the result of two possible reasons: the function is called too frequently while the application is running, or the function is running slowly. This requires the use of another tool or other method to define the reason. In the two cases, optimization is required [80]. We found in this type of profilers **OProfile** and **perf** (Linux).

Instrumented profiling It is a technique that adds instructions to the source code or to the executable of the target program to collect the required information. This technique allows to choose which information to collect but may cause performance change and lead to inaccurate results. This may be done manually by adding instructions to the source code or compiler assisted by adding profiling symbols to the

executable program. We found in this type of profiling **GProf** from GNU project.

3.4 Design Flow

Now we have the hardware and the software platforms, the design flow shown in Figure 3.15 is followed to implement the SLAM on the embedded target and improve its performance.

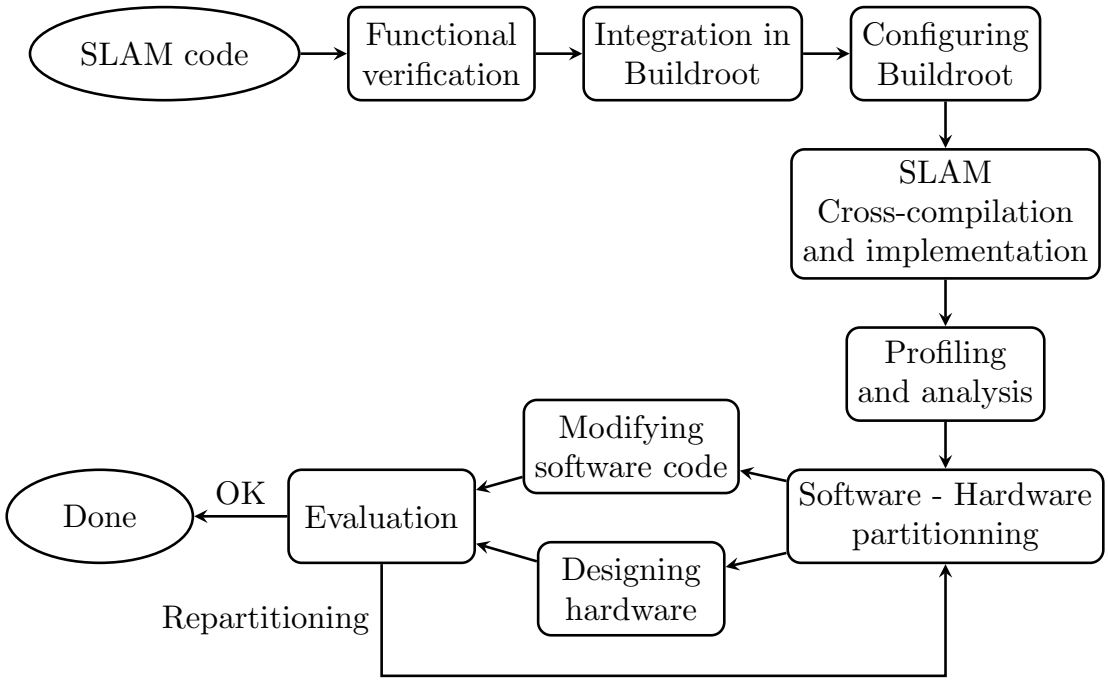


Figure 3.15: SLAM design flow on the SoC-FPGA

The first step is to take an existing SLAM code and verify its functionality in a PC-Like machine. The code is compiled to be run on an x86 architecture. This step ensures the well functioning of the SLAM. The results obtained can be used to compare with those obtained from the embedded target.

After verifying the correct functioning of the SLAM, we proceed with its integration into Buildroot. The SLAM application is integrated in Buildroot as a package. If one of its dependencies is not included in the list of the supported packages, then it must be integrated too. The integration is done by adding two files: `Config.in` and

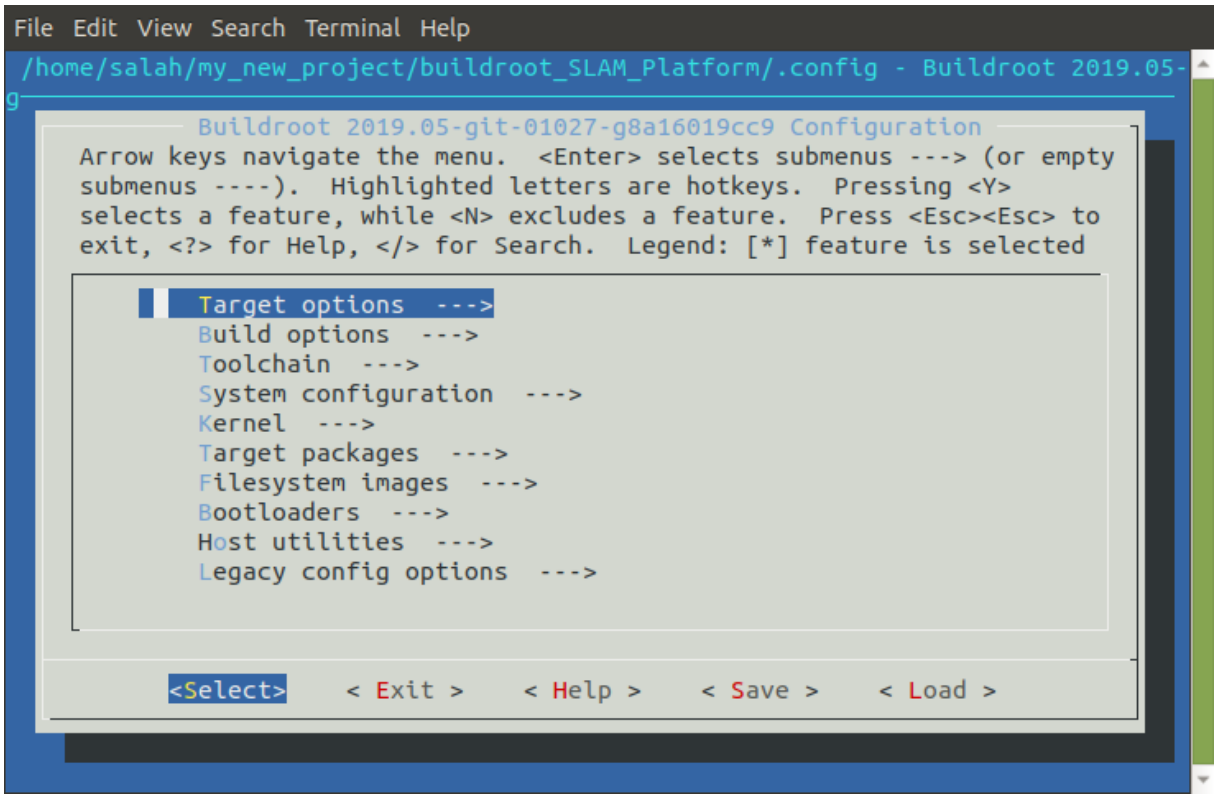


Figure 3.16: Buildroot configuration graphical interface *menuconfig*

`package.mk` with `package` is the name of the package. The `Config.in` file contains the package configurations like: name, dependencies.etc. While `package.mk` contains a recipe of how to download, configure, build and install the package. These files are included in a folder inside `packages/` folder with the name of the package. The main `Config.in` file must be modified to include the new package. These two first steps are done once at the first time. Next time, we don't need to reintegrate the package in Buildroot.

After integrating the needed packages in Buildroot, we configure buildroot to build our system. The configuration is done using `menuconfig` (see Figure 3.16), a graphical interface that helps to select the options and packages to be included in the system.

From this menu we can configure the target options, the toolchain to use, the Linux kernel options, the target packages, and the output filesystem image. In the target package submenu, we found the newly integrated SLAM application and de-

pendencies. From the same menu, we choose a profiler and debugger application to be included in the system. This allows us to debug and profile the application if it is needed. To debug the application we must add the debugging option in the build option submenu. This means that the compiler will add debugging symbols to the executable files.

After finishing the configuration, we proceed to the cross-compilation. In this step Buidroot will use the cross-compilation toolchain to compile the packages of the system and produce the filesystem image.

After building the system and implement it on the target, it is ready now to be executed. The SLAM application is executed and profiled. The profiling allows us to identify bottlenecks in the program and helps us in the software/hardware partitioning. The bottlenecks are analyzed to make a decision of which part of the code to be implemented in the hardware and how will be implemented. The hardware/software partitioning requires a modification in the software source code by replacing the part of code to be accelerated with a calling function to the accelerating hardware. The hardware is designed with OpenCL as described in section 3.2.4.2. The system is then rebuilt and evaluated to decide if a new partition is needed or no. If no repartitioning is needed, the system is rebuilt without debugging symbols, to have the final version of the SLAM system.

3.5 Conclusion

In this chapter, we have described the two parts of our platform.

In the hardware part, we have seen the existing architectures, and we opted for the SoC-FPGA because of the benefits that it offers. The combination of a mobile CPU and an FPGA in the same chip, allows us to take advantage of the advantages of a hard-processor to execute the software and of the advantages of the high parallelism of the FPGA to accelerate the heavy functions, in addition to the high bandwidth data transfer between the two parts. We used the development board DE1-SoC that contains a low-cost SoC-FPGA from IntelFPGA Cyclone V SoC.

In the software part, we opted for a system based on embedded Linux. In addition to the advantage of being open-source, Linux is modular and customizable. It allows to be customized according to our needs. To build the SLAM system based on Linux, we used the automatic build system Buildroot. Buildroot allows to shorten the development time and automate the build of the system.

In the end, we described the design flow of the SLAM system on the embedded target. The design is based on profiling the program and software-hardware partitioning.

In the next chapter , we see a case study of implementing a visual SLAM based on EKF on the platform.

CHAPTER 4

CASE STUDY: IMPLEMENTATION OF EKF-SLAM ON DE1-SOC

This chapter is based on a journal article co-authored by Rabah Sadoun and Mourad Adnane [Bouhoun et al., JSA, 2020 [81]]

Contents

4.1 Introduction

In the chapter we see a case study of implementing of an existing SLAM on our embedded platform. MonoSLAM an EKF based SLAM developed and implemented by Davison and reimplemented by Hanme Kim, is used in our application. In the next section, we describe the EKF-SLAM and the MonoSLAM program. Then, we will configure our platform to implement the chosen SLAM.

After preparing the platform, we implement the SLAM in the platform following the design flow described in the past chapter.

4.2 Monocular EKF-SLAM Algorithm

Monocular EKF-SLAM is a SLAM based on Extended Kalman Filter with a single camera. It has two main parts, the prediction step and the estimation step. In the prediction step, we calculate the predicted (a priori) state of the robot using previous state information and proprioceptive sensors if they exist. Measurement with an exteroceptive sensor is taken, and a matching process finds the association between measured and predicted features. In the estimation (correction) step, we estimate the new state of the robot and the features using the information from the exteroceptive sensors. An initialization step is required whenever the robot explores a new area to add new landmarks to the map (state vector). In this paper, a hand-held camera is used as an exteroceptive sensor for the correction step, while a prediction method is used to replace the lack of the proprioceptive sensor.

4.2.1 State vector and covariance matrix

The state vector $\hat{x}$ and the covariance matrix P describe the environment map, with N landmarks. They are defined as:

$$\hat{x} = \begin{pmatrix} \hat{x}_v \\ \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{pmatrix} \tag{4.1}$$

$$P = \begin{pmatrix} P_{xx} & P_{xy_1} & P_{xy_2} & \cdots & P_{xy_N} \\ P_{y_1x} & P_{y_1y_1} & P_{y_1y_2} & \cdots & P_{y_1y_N} \\ P_{y_2x} & P_{y_2y_1} & P_{y_2y_2} & \cdots & P_{y_2y_N} \\ P_{y_3x} & P_{y_3y_1} & P_{y_3y_2} & \cdots & P_{y_3y_N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ P_{y_Nx} & P_{y_Ny_1} & P_{y_Ny_2} & \cdots & P_{y_Ny_N} \end{pmatrix}. \tag{4.2}$$

In this case study we use 3D landmarks, and a camera with 6 degrees of freedom. The camera state vector $\hat{x}_v$ will contain the position vector r^W, the orientation vector represented with a quaternion q^{WR}, the velocity vector v^W and the angular velocity vector ω^R. W and R are defined as coordinate frames fixed, respectively, in the world or with respect to the camera:

$$\hat{x}_v = \begin{pmatrix} r^W \\ q^{WR} \\ v^W \\ \omega^W \end{pmatrix} = \tag{4.3}$$

$$\begin{pmatrix} x & y & z & q_1 & q_2 & q_3 & q_4 & v_x & v_y & v_z & \omega_x & \omega_y & \omega_z \end{pmatrix}^T \tag{4.4}$$

The landmarks state component $\hat{y}_i$ are written in Cartesian coordinates.

The covariance matrix P represents the uncertainty of the state vector. The diagonal values are variance of the state vector elements, while the off-diagonal values are

the correlation between them. They shows how the measurement of an element of the state vector affects the estimation of the other elements [18].

While this matrix is useful to recognize known areas after loop-closing, it presents a major constraint in implementing EKF-SLAM. This is because increasing the size of the state vector, by exploring more landmarks, will cause it to grow exponentially. This requires the same exponential increase in computation and memory allocation on the hardware target.

The EKF-SLAM steps are shown in Figure 4.1. The main steps are *Prediction*, *Matching*, *Updating* and *Initialization*.

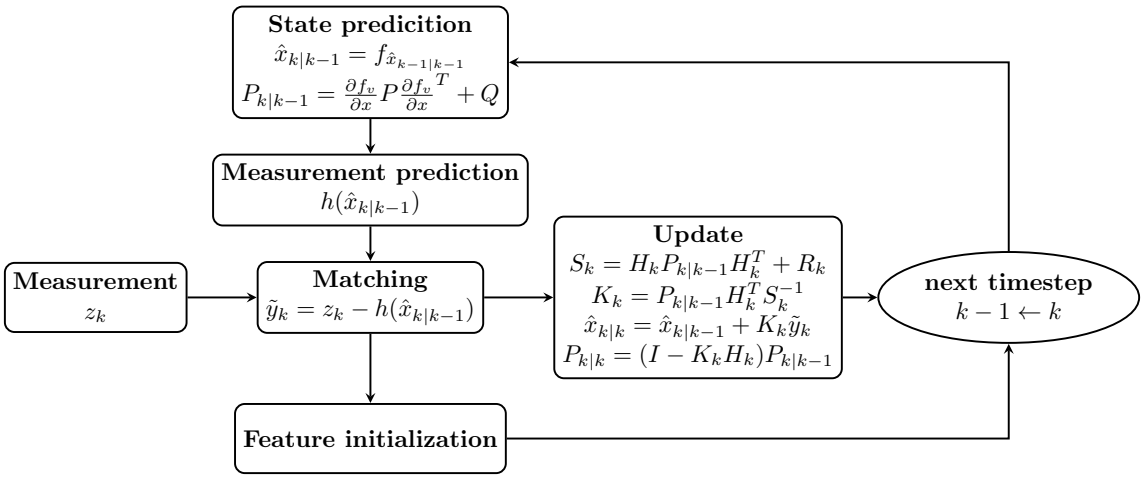


Figure 4.1: Flowchart of EKF based SLAM

4.2.2 Prediction step

In the prediction step, the current state n of the robot is predicted according to its previous state $n - 1$.

$$\hat{x}_{k|k-1} = f_{\hat{x}_{k-1|k-1}, u_{k-1}} = f_v \tag{4.5}$$

In the case of a hand held camera, the control vector does not exist. The movement of the camera causes an unknown linear acceleration $\mathbf{a}^W$ and angular acceleration α^W, that affect on the linear velocity and the angular velocity. These accelerations are

supposed to be a process having a Gaussian distribution with a zero mean. They are added to the transition vector as Gaussian noise vector:

$$\mathbf{n} = \begin{pmatrix} \mathbf{a}^W \Delta t \\ \alpha^W \Delta t \end{pmatrix} = \begin{pmatrix} \mathbf{V}^W \\ \mathbf{\Omega}^W \end{pmatrix} \tag{4.6}$$

The state update equation will be:

$$\mathbf{f}_v = \begin{pmatrix} \mathbf{r}^W_{new} \\ \mathbf{q}^{WR}_{new} \\ \mathbf{v}^W_{new} \\ \omega^W_{new} \end{pmatrix} = \begin{pmatrix} \mathbf{r}^W + (\mathbf{v}^W + \mathbf{V}^W)\Delta t \\ \mathbf{q}^{WR} \times \mathbf{q}((\omega^W + \mathbf{\Omega}^W)\Delta t) \\ \mathbf{v}^W + \mathbf{V}^W \\ \omega^W + \mathbf{\Omega}^W \end{pmatrix} \tag{4.7}$$

The predicted covariance matrix is calculated:

$$P_{k|k-1} = \frac{\partial f_v}{\partial x} P \frac{\partial f_v}{\partial x}^T + Q \tag{4.8}$$

After predicting the positions of the landmarks, the prediction of their new projections on the camera frame (u, v) is calculated using the measurement model. From the camera state and the positions of the landmarks, we can predict the measurement of the landmark relative to the camera.

$$h^R_L = R^{RW}(y^W_i - r^W) \tag{4.9}$$

Using the pinhole camera model, the predicted position of the feature in the image is calculated:

$$h_i = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u_0 - fk_u \frac{h^R_{Lx}}{h^R_{Lz}} \\ v_0 - fk_v \frac{h^R_{Ly}}{h^R_{Lz}} \end{pmatrix} \tag{4.10}$$

4.2.3 Matching step

In this step, a matching process searches for a new image patch in the region near each feature, similar to the initial image patch saved from the previous step. First, a search region is calculated for each visible feature in the new frame, it is determined from the feature position y_i and the covariance matrix P. The reference image patch $f(u, v)$, from the previous frame, scans the region and searches for the position of the

feature $t(u,v)$ in the new frame, using the value of the Normalized Sum of Squared Differences (NSSD) (4.11).

$$NSSD = \frac{1}{n}\sum_{u,v}\left(\frac{(f(u,v)-\bar{f})}{\sigma_f} - \frac{(t(u,v)-\bar{t})}{\sigma_t}\right)^2 \tag{4.11}$$

where $n = length \times width$ of the image patch.

After matching the measured features, measurement error is calculated:

$$\tilde{y}_k = z_k - h(\hat{x}_{k|k-1}) \tag{4.12}$$

4.2.4 Correction and update step

In the correction step, the innovation covariance matrix S_k and the Kalman gain K_k are calculated, and the state vector and the covariance matrix are updated:

$$S_k = H_k P_{k|k-1} H_k^T + R_k \tag{4.13}$$

$$K_k = P_{k|k-1} H_k^T S_k^{-1} \tag{4.14}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k \tag{4.15}$$

$$P_{k|k} = (I - K_k H_k) P_{k|k-1} \tag{4.16}$$

4.2.5 Feature initialization step

In this step, an initialization of new features is done, if there are not enough visible features. In order to do that, we must extract some features from a sensible area in the camera image.

Davison and Murray [82] show that image patches with large dimensions (9×9 to 15×15) can serve as long-term landmark features. In this work, the features are detected using the Shi and Tomasi operator, with patches of 11×11 pixels. The image patch is saved and associated with the corresponding point.

These features are the 2D projections on the camera (u, v), and it cannot be converted directly to give the 3D positions of the landmarks due to the lack of the depth information, we must have more than one image to extract it. The method of Davison [18] is followed, it consist of a first representation in form of 3D semi-infinite line starting at the estimated camera position and heading to the feature direction. The landmark is predicted to be in this line, the probability of its position is equally distributed along this line in the first measurement. The probability distribution is approximated as Gaussian distribution, and after many measurements the probability of the landmark position converge. The standard deviation getting lower so that its representation in the vector state can be replaced with 3D coordinates.

4.3 Integrating MonoSLAM in Buildroot

The MonoSLAM is based on SceneLib2 library, developed by Davison [22] and reimplemented by Kim [67]. The SceneLib, itself, depends on some libraries. All those libraries, except one, are included in the packages list of Buildroot (OpenCV, Boost, ffmpeg, FreeGLUT, etc.) These package are selected in the *menuconfig*. Pangolin is a library that isn't included in the package list of Buildroot.

To add the SceneLib package to Buildroot we add `Config.in` and `SceneLib.mk` files in `package/SceneLib2/` folder, and `Config.in` and `Pangolin.mk` files in `package/Pangolin/` folder. These files are included in the appendix of this thesis (see Appendix A.1 and Appendix A.2).

For the MonoSLAM application, we created the main program executing SLAM locally on the host machine. This application uses SceneLib2 library. Like other packages, the application is also added to Buildroot list of packages by adding `Config.in` and `SceneLib.mk` files in `package/SLAM/` folder (see Appendix A.3).

Newly added packages need to be listed in the packages configuration menu. We add those packages to the configuration menu by adding entries to `packages/Config.in` file (see Appendix A.4).

After integrating the packages and add entries to the configuration menu Kconfig

file, we configure our system. First we use a predefined configuration type of our platform that exists in Buildroot by calling `make altera_socdk_defconfig`. This will set the target, kernel, and toolchain options of the Buildroot configuration to the parameters of the used platform (DE1-SoC). We select then the dependencies packages in target packages menu and our packages. (see figures 4.2)

After configuring and selecting the system packages, we launch the building process. The source code of each package is first downloaded then patched if any patches exists. After applying the patches, It is configured and built. The package is then installed in the output folder. The resulted files are copied to the output folders according to their destination, if it is a host package they are copied to the host folder to be used by other packages, and if it is a target package, they are copied to the target folder. After, building all the packages, the target filesystem is created using the target output folder.

The build process need a large time especially in the first build beacause of downloading packages. Figure 4.3 shows the build time of the SLAM system by steps excluding download step. Package building and configuring take the majority of the build time. Figure 4.4 shows a snippet of the package build durations histogram.

4.4 MonoSLAM profiling and analysis

The SLAM program was tested and profiled on dual-core ARM Cortex-A9. Table 4.1 shows a snippet of the profiling result of the SLAM program. The profiling tools use a statistical approach, and the shown results are the self-time of each function. We notice that *correlate2_warning* function occupies 55.90% of the global time of the program. To get more information, an analysis of the function is required, in addition, we examine the caller tree until the main function, to determine the best level of the tree to be accelerated.

The *correlate2_warning* calculates the correlation between two image patches, and return the sum of the squared differences Eq. (4.11). This function is used to search for a feature in a region.

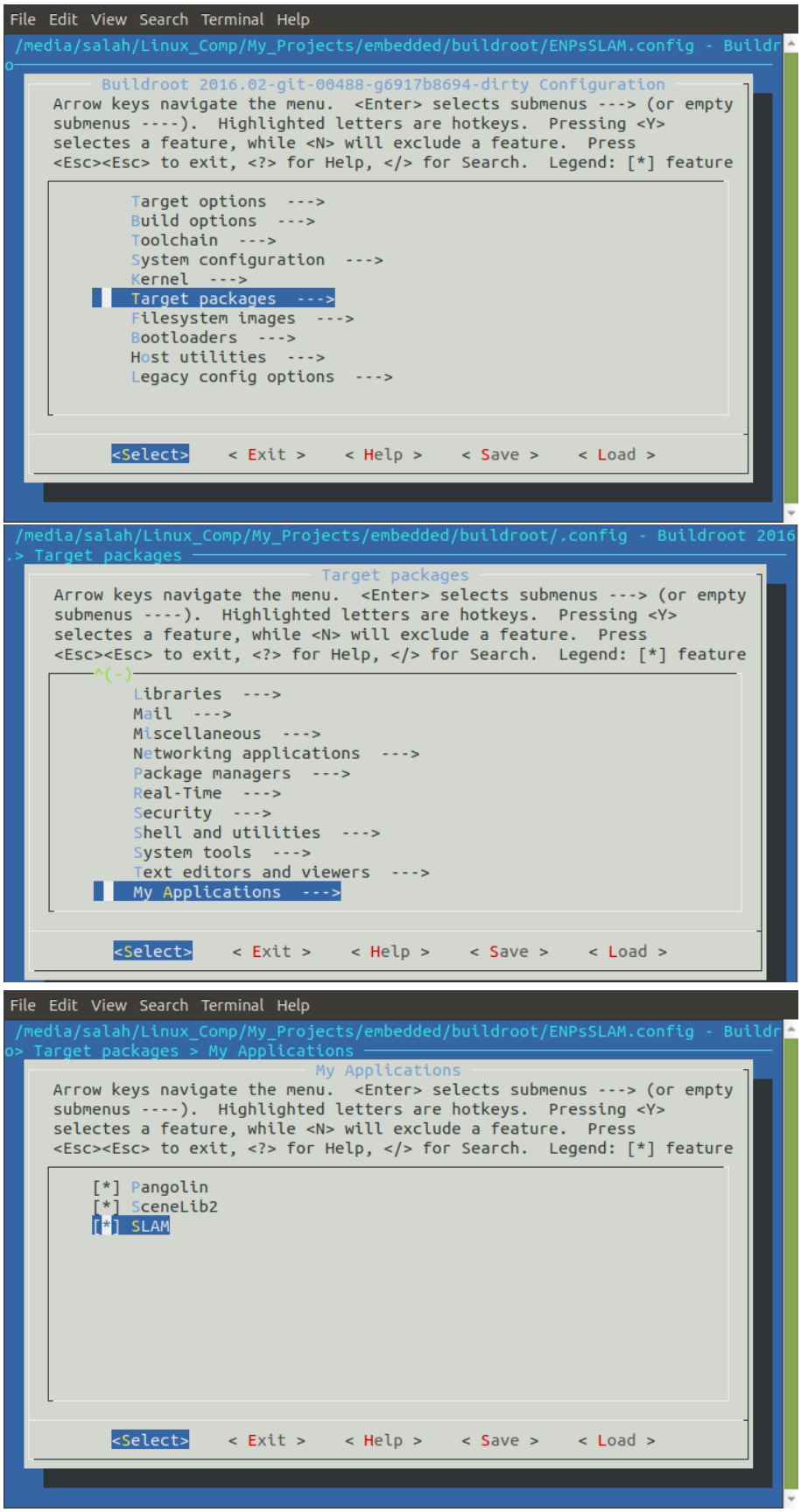


Figure 4.2: Selecting the SLAM package in Buildroot menuconfig

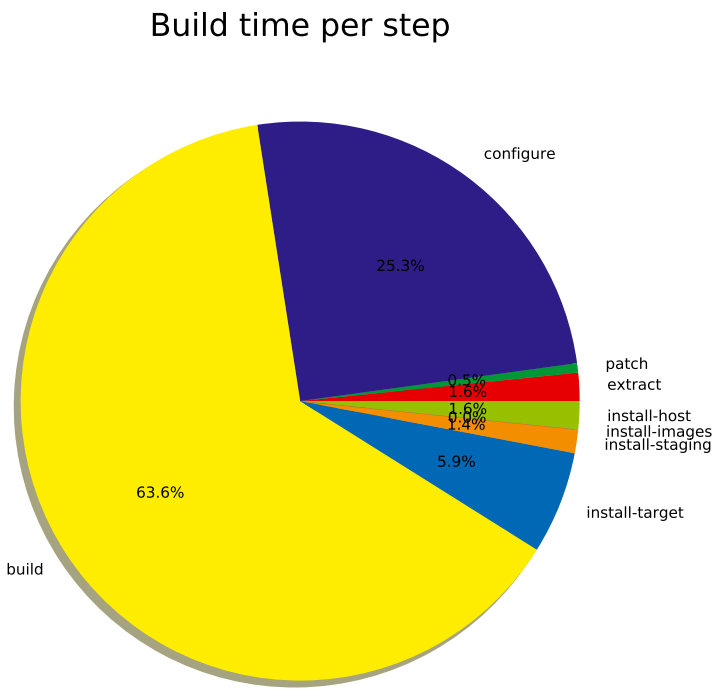


Figure 4.3: Duration fo building steps

A basic implementation of the function (Algorithm 1 requires calculation of 5 loops, in 3 sequential stages, each loop has 11x11 iteration ($\bar{f}$, $\bar{t}$, σ_f, σ_t, $NSSD$).

With this implementation we need to access to the same memory elements ($f(x,y)$ and $t(x,y)$) three times, first to calculate the patches' means (lines 3 to 6), second, to calculate the standard deviations of the two patches (lines 11 to 14) which depend on the result of the previous loop, finally, to calculate the NSSD (line 20 to 22) which depends on the result of the previous loops too.

The implemented function in the SceneLiB2 library, is simplified to have just one loop, so the Eq. (4.11) becomes:

$$NSSD = \frac{1}{n}\left(\frac{S_{f^2}}{\sigma_f^2} + \frac{S_{t^2}}{\sigma_t^2} + n \times k^2 - 2 \times \frac{S_{f\times t}}{\sigma_f \sigma_t} - 2 \times \frac{k \times S_f}{\sigma_f} + 2 \times \frac{k \times S_t}{\sigma_t}\right) \quad (4.17)$$

where :

$S_{f^2} = \sum_{x,y} (f(x,y))^2$ is the sum of the squared values of the reference image patch

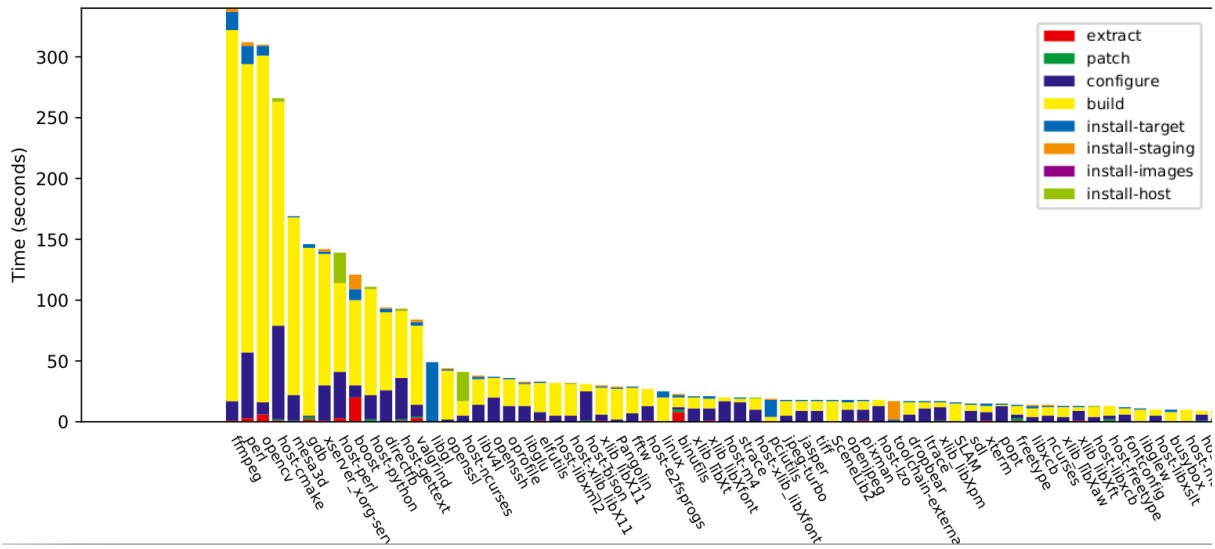


Figure 4.4: A snippet of package build duration histogram

Table 4.1: Profiling result of the SLAM program

function	percentage
correlate2_warning	55.90%
Eigen::internal::gebp_kernel	7.97%
SearchMultipleOverlappingEllipses::search	2.61%
MonoSLAM::elliptical_search	2.60%

pixels.

$S_{t^2} = \sum_{x,y} (t(x,y))^2$ is the sum of the squared values of the new image patch pixels.

$S_f = \sum_{x,y} (f(x,y))$ is the sum of the reference image patch pixels.

$S_t = \sum_{x,y} (t(x,y))$ is the sum of the new image patch pixels.

$S_{f \times t} = \sum_{x,y} (f(x,y) \times t(x,y))$ is the sum the products of the reference image patch pixels and the new image patch pixels.

The algorithm 1 becomes as it is shown in algorithm 2.

This implementation reduces the number of dependent consecutive loops from three to one loop, this will reduce the memory access by preventing the access to the two patches elements more than one time. Inside the loop, there is 5 independent operations executed on each iteration to get 5 values. The number of iterations depends

on the size of the patch. Taking a patch of 11×11 in our case, this will result on 121 iterations. These iterations are executed sequentially on the processor, this will take $5 \times 121 = 605$ operations. Other expensive operations are used after the loop, we have 2 squared root operations and 10 division operations.

The *correlate2_warning* function is called by two other function, *elliptical_search* with 96.2% of calls and *SearchMultipleOverlappingEllipses::search* with 3.8% of calls. To retrieve more information we have to examine the *elliptical_search* function, which has a high number of calls to the *correlate2_warning* function. The Algorithm of the *elliptical_search* is described in Algorithm 3.

For each feature, a search region is delimited using the projection of its predicted position on the screen and its innovation covariance. For this region, the corresponding reference image patch will scan the pixels of this region (lines 4 to 10) to find the new feature position on the screen. On each loop, the reference image patch and a new image patch from the search region are passed to the *correlate2_warning* function (line 5). The projection coordinates of the new feature is determined by the lowest correlation result (lines 6 to 9). The search process will be executed sequentially due to the software implementation nature. This means that the next call to *correlate2_warning* is not called only after the return of the present call, even the two processes are independent. The processing time depends on the size of the search region.

4.5 SLAM program implementation flow

4.5.1 Hardware-software partitioning

In this step, we chose the part of the code to be implemented on the FPGA. Accoding to the definition of Amdahl's law [83], the acceleration of a task is determined by the the amount of the parallelized part and the acceleration of this part. This can be written by the following equation:

$$S = \frac{1}{1 - p + \frac{p}{s}} \tag{4.18}$$

S is the acceleration of the whole task, p is the parallelized part and s is the acceleration of the parallelized part. The theoretical upper limit of the acceleration then will be:

$$S_{limit} = \frac{1}{1 - p} \tag{4.19}$$

the higher the p, the higher the upper limit. The acceleration factor s depends on the execution time of the parallelized part. In hardware acceleration this time is the sum of the computation time and the data transfer time to and from the accelerator. In the hardware/software partionning process, we must choose the most time consuming part of the software with less data transfer to reduce the execution time.

Implementing *correlate2_warning* function as it is, with OpenCL for FPGA is not optimal due to the inefficiency of the data transfer. The *correlate2_warning* function requires the transfer of the reference image patch and the new image patch each time, this transfer has redundant information that can be eliminated. Figure 4.5 shows scan process of the search region by the reference image patch, for each iteration, the *correlate2_warning* function is called, the 11×11 reference image patch and 11×11 image patch from the search region are transferred. Hence, the reference image patch is transferred several times, in addition to the common pixels (dark blue) from two successive image patches, this reduces the efficiency of the data transfer.

To eliminate the transfer of the redundant data, the reference image patch and the search region are transferred entirely to the hardware at the beginning of the process.

From the previous The caller loop of the *correlate2_warning* (Algorithm 3 from lines 4 to 10) is chosen to be implemented in the hardware.

4.5.2 OpenCL Implementation

Despite that the FPGA offers the possibility of designing a parallel architecture, that speeds up the processing, an unoptimized architecture can cause poor performance

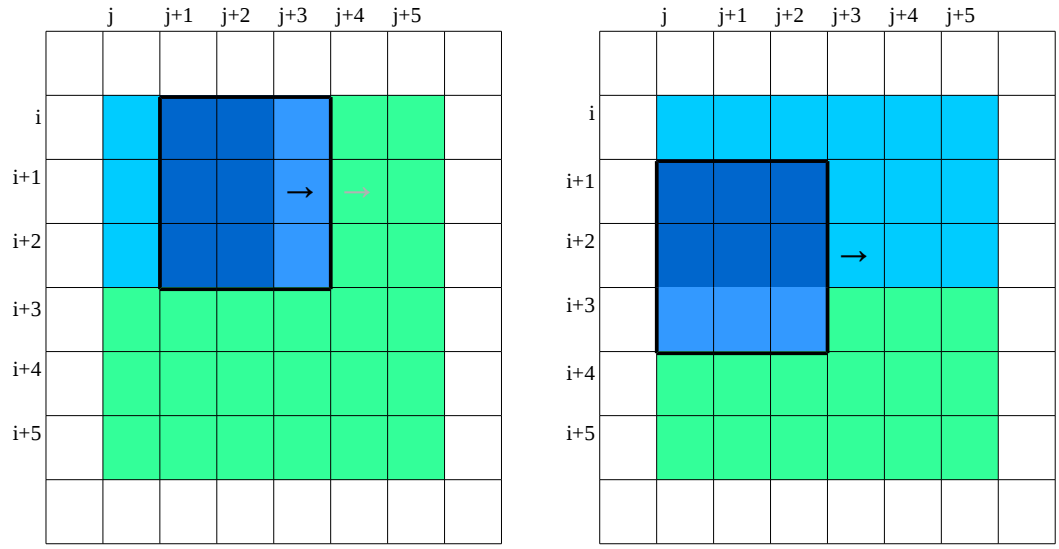


Figure 4.5: scanning process of search region by a reference image patch

comparing to the software version. Designing an accelerator circuit on an FPGA platform with OpenCL, must consider these 3 types of constraints: computation constraints, communication or data transfer constraints and available resources in the FPGA.

The optimization report, generated by the IntelFPGA offline compiler, helps us to optimize the kernel by showing the origin of the stalls which can be a result of a data dependency, memory dependency, or loop iteration dependency.

4.5.2.1 Implementation constraints

As starting point we choose a single work-item kernel, as it is recommended by Intel FPGA [84], since the function does not have an explicit description of multiple concurrent threads. Figure 4.6 shows the execution flow of the OpenCL program. The kernel in this case is similar to the C code. The algorithm is implemented using a pipelined architecture, each stage of the pipeline is an operation of the algorithm. The data passes through the pipeline stages, and each stage process one data per cycle concurrently.

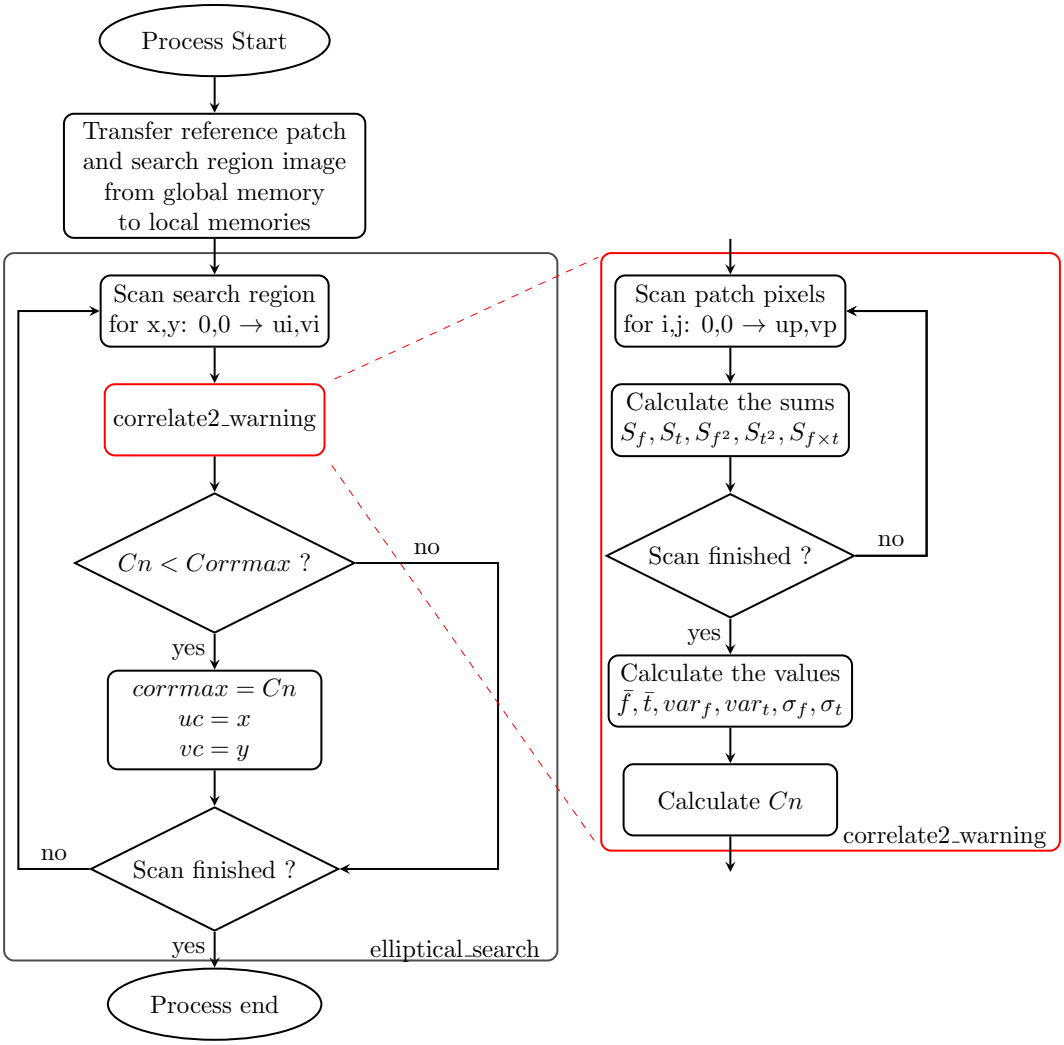


Figure 4.6: Diagram flowchart

Computational constraints The main computational constraint in implementation of this function is the nested loops. The nested loops have an Initiation Interval (II) greater than 1, which is the interval between the launches of two successive loops. This prevents the outer loops from initiating an iteration every clock cycle, and generates stalls in the pipeline.

Data transfer constraints In OpenCL, the data transfer time is an important factor since it has a great influence in the global processing time. In an OpenCL kernel for the FPGA, the only shared memory between the host and the device is the global

memory. The input data transferred between the host and the OpenCL device are initially loaded to the global memory.

The optimization report shows that the load operations in the inner sums loop are stallable, this prevents the outer sums loop to be launched every clock cycle ($II > 1$). The report suggests a dynamic profiling to determine the performance impact.

The second constraint is the kernel parameters transfer constraint. Unlike vectors, reading and writing kernel single parameters are not read in a burst mode. The reading and writing these parameters one by one takes a long time comparing to reading and writing vectors.

FPGA resource constraints Some operations in the algorithm are expensive and use lot of FPGA resources. In this algorithms, division and the square root operations are used many times. This risks that the kernel does not fit in the FPGA.

4.5.2.2 Kernel optimization

Many researchers used OpenCL to implement their system on an FPGA [85–87]. They also explored the optimization techniques to improve the performance of the OpenCL kernels and to better use of the FPGA resources. In this subsection we describe the optimization techniques used in our work.

Unrolling loops We choosed to implement the Algorithm 2, since it presents less nested loops. The loop in this algorithm (from Algorithm2 line 6 to line 12) calculate the five sums S_f, S_t, S_{f^2}, S_{t^2} and $S_{f \times t}$, two accumulate operations and three multiply accumulate operations. These 5 operations are done simultaneously, since they are independent. The inner loop calculates the lines, and the outer loop calculates the columns. To speed up the calculation, the inner loop is fully unrolled to be able to process one line per cycle, and the outer loop can load one line per cycle, with an II of 1. However unrolling loop request more FPGA resources, and generates simultaneous access to the memory, which can cause performance degradation.

Minimizing expensive operations Complex operations require a large FPGA resource usage and cause a long time delay. To minimize the number of complex operations, the 5 divisions by the constant n are replaced by the multiplication by $1/n$. The lines from 19 to 21, containing many division operations, are simplified by reducing the formula to a common denominator, so we will need only a division operation. The two formulas become:

$$G = (\bar{f} \times \sigma_t) - (\bar{t} \times \sigma_f) \tag{4.20}$$

$$\begin{aligned} Numerator = (S_{f^2} \times \bar{t}) + (S_{t^2} \times \bar{f}) + \\ (n \times G^2) - (2 \times S_{f \times t} \times \sigma_f \sigma_t) + \\ (2 \times G) \times ((S_t \times \sigma_f) - (S_f \times \sigma_t)) \end{aligned} \tag{4.21}$$

$$Denominator = \bar{f} \times \bar{t} \tag{4.22}$$

$$NSSD = \frac{1}{n} \times \frac{Numerator}{Denominator} \tag{4.23}$$

The lines from 19 to 21 becomes as shown in Algorithm 4:

Data types Data types has an important impact on the processing speed and the FPGA resources usage.

The images used by the SLAM are black and white, and the pixels are written as a 256 level grayscale. The reference image patch and the search region are coded as a vector of unsigned char, this minimizes the amount of data transferred from the host to the device, and reduce the resources used to memorize and process them.

The result of the five sums calculated in Algorithm 2 lines 6 to 12 are coded as integers, since they are sums and multiplications of unsigned characters. This will reduce the time of the two operations and the resources used to calculate them.

The other values used in the algorithm 2 from lines 13 to the end are written in 32 bits float point.

The output values, the position of the smallest NSSD and its NSSD, are structured so that they can be transferred as vectors instead of reading them one by one, which will reduce the read time.

Data structuring The output of the device is a structure containing the smaller NSSD in the search region with the coordinates of the new image patch corresponding to this NSSD, the structure is interpreted as an array, and the transfer is more efficient. Since it is able to be changed every cycle, according to the comparison between NSSDs. This structure is stored on local memory to prevent stallable store operations to the global memory. At the end, the result is transferred to the global memory to be read by the host.

Cached memory To avoid the stallable operations, the data stored in the global memory are transferred to the local memory. This is shown as the first step in Figure 4.6. The local memory has a zero latency, the load operations are stall free, and the Initiation interval will be equal to 1.

Memory replication The inner sums loop is unrolled, this requires many load operations at one clock cycle, however to support multiple access efficiently to the local memory, it must be replicated. In our case, 11 load operations are needed from each of the two memories containing the reference image patch and the new image patch, the memories are replicated 3 times. The replication reduces the access time, however it increases the resources usage. This optimizations is automatically done by the offline compiler.

4.6 Evaluation and Discussion

In order to implement and evaluate the SLAM system, we used a DE1-SoC board from Terasic containing the Cyclone V SoC. Cyclone V SoC is a low cost and low power SoC from IntelFPGA. It consists of two parts: a dual core ARM Cortex A9 based Hard Processor System (HPS) operating at 925 MHz and an FPGA fabric. A DDR3

Memory is associated with the HPS, and can be accessed from the FPGA through the HPS. Both HPS and FPGA are interconnected with a high throughput datapaths with a bandwidth up to 6400MB/s. DSP blocks in the FPGA provides up to 32.8 Giga floating point operations per second (GFLOPS). Table 4.2 shows the resources available on Cyclone V SoC FPGA 5CSEMA5 used in our work.

Resources		Cyclone V 5CSEMA5
Logic Elements (LE) (K)		85
ALM		32075
Registers		128300
Memory(Kb)	M10K	3970
	MLAB	480
Variable-precision DSP Block		87
18 x 18 Multiplier		174

Table 4.2: Table of resource for Cyclone V SE

The SLAM program and libraries with a custom Linux OS are cross-compiled and built using Buildroot in PC-like machine, targeting the ARM CPU used in the Cyclone V SoC. Buildroot automates the build process and reduce its time. In the end, it produces a small system with only the needed tools and libraries.

The accelerated function is developed in OpenCL using Intel FPGA SDK for OpenCL. The Intel FPGA SDK for OpenCL offline compiler will perform all the steps needed to generate an FPGA hardware. The Intel FPGA SDK for OpenCL offline compiler generates an FPGA programming image of the built hardware that will be used by the HPS to program the FPGA.

The designed OpenCL kernel is compiled with offline compiler using two options : *-fp-relaxed* that relaxes the order of floating-point operations using a balanced tree hardware implementation, and *-fpc* that reduces the floating-point rounding operations. This will reduce the resource usage of the FPGA and speeding up the processing. We used *-profile* with offline compiler to profile the FPGA implementations. The

profiling report of the OpenCL implementation can be exploited using Intel FPGA dynamic profiler for OpenCL. We used a random image of $n \times n$ as a search region and a random image patch of 11×11 from this image as reference patch.

Figure 4.7 shows the execution time of the OpenCL kernels in the FPGA using different optimization techniques (UL: unrolling loops and LM: using local memory) compared to the unoptimized kernel. The kernels having unrolled loops achieve a speedup nearly 5 times compared with the kernels having unoptimized loops. The kernel with local memory as a cache shows a speedup that can achieve 2 times compared with kernel without a cache. However, it has no effect on the kernel having unoptimized loops. This is because the global memory bandwidth suffices to feed the kernel with data in real time.

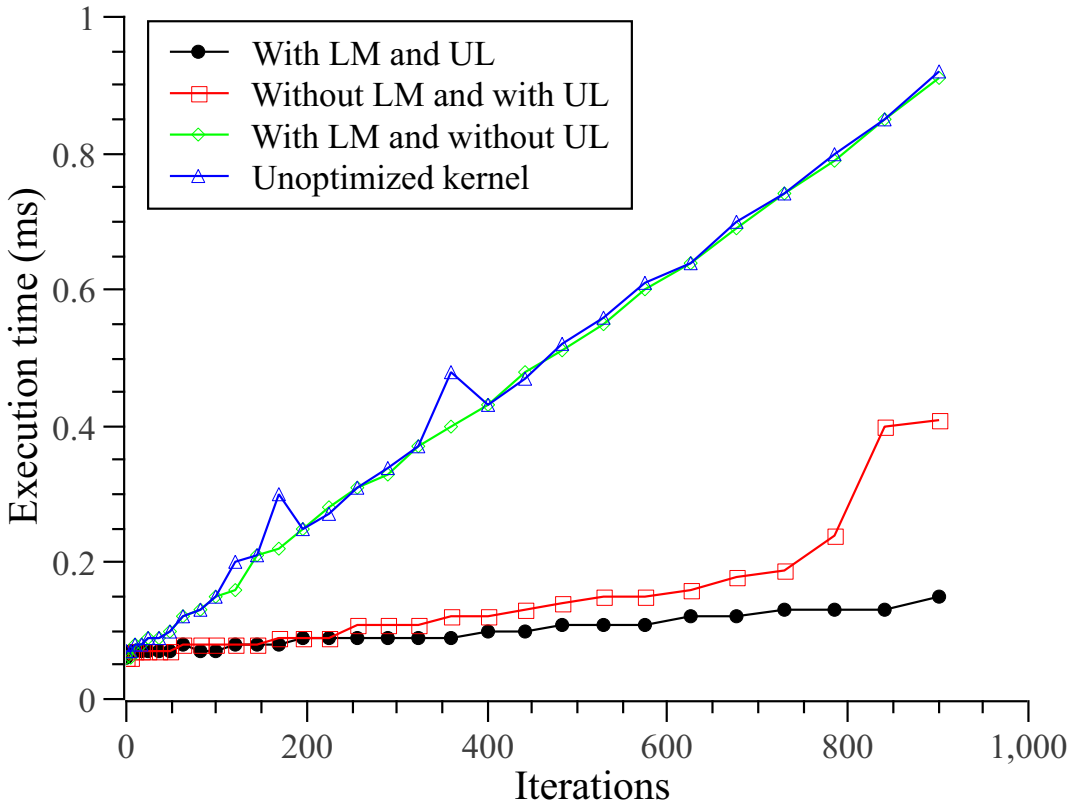


Figure 4.7: Execution speed kernels with/without LM: using local memory and UL: unrolling loops

Reading and writing very used data, requires a high bandwidth between the FPGA and the used memory. Using the global memory as the main memory, to store data, generates a heavy traffic with this memory. The global memory has a theoretical

bandwidth limit (claculated to 4010MB/s by the dynamic prfoiler) and it is subject to stalls. In the other hand, the local memory is a no delay memory, and it is designed to deliver data at the same speed of the FPGA, hence the traffic with the global memory is light. The data traffic with the global memory is shown in Figure 4.8. The kernel with unrolled loops has parallel operations executed simultaneously, which requires more data at a time. This explains the the high data traffic in the kernel with unrolled loops, and less data traffic in the unoptimized kernel.

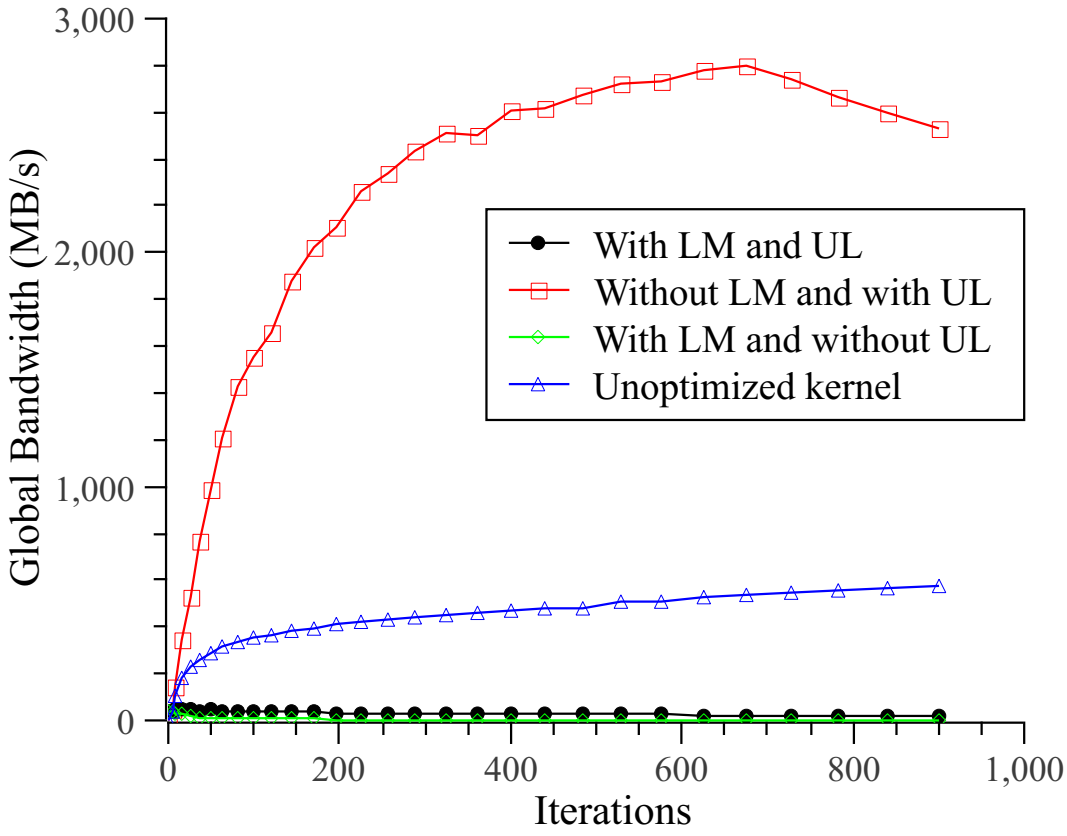


Figure 4.8: The global bandwidth needed to access the global memory

As mentioned in section 4.5.2.2, transferring input data to a local memory is supposed to prevent stalls that can be caused by random access to the global memory. Using the dynamic profiler, we can see in Figure 4.9 the percentage of stalls time when using the global memory to read the input data. The stall time percentage depends, also, on the required data transfer rate, that's why the unoptimized kernel generates lower stalls time percentage than the kernel with unrolled loops. On the other hand,

there is no stalls when using a local memory to read the input data.

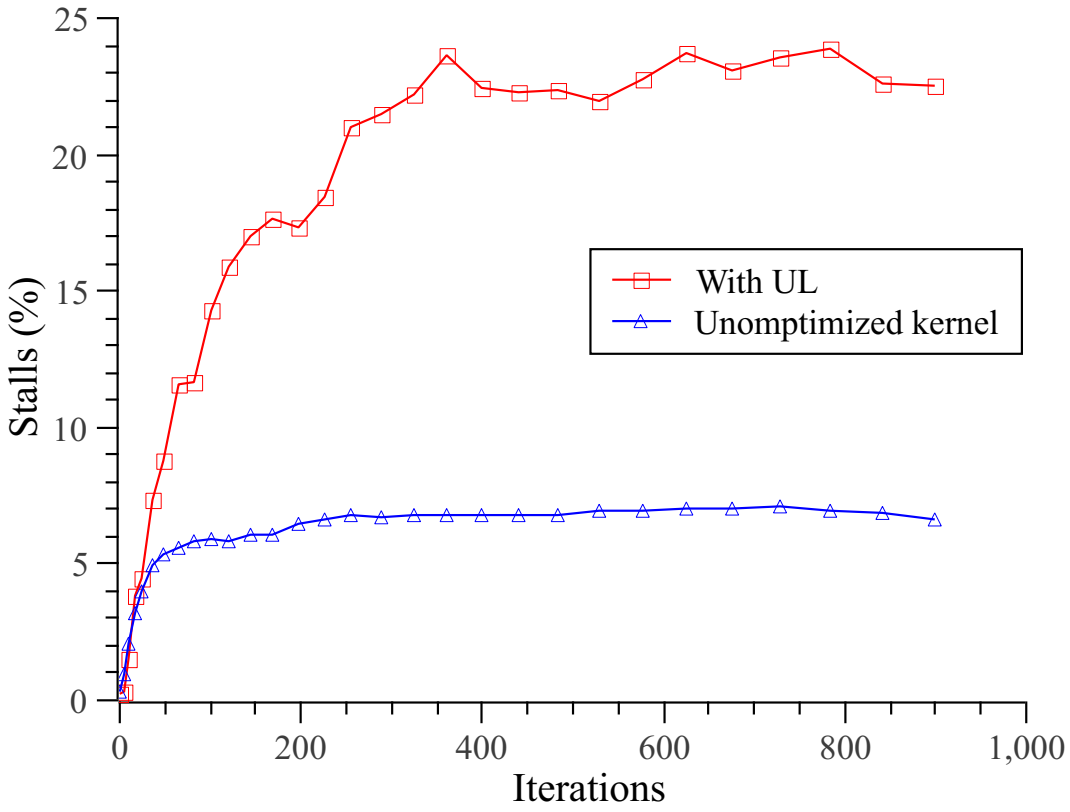


Figure 4.9: Percentage of stalls time in global memory accessing

The optimized kernel is chosen to be used as accelerator for the SLAM application. The table 4.3 shows the report generated by the offline compiler, it contains the FPGA resources used for the optimized kernel and its running frequency for the optimized kernel.

As we can see, in addition to the high usage of the logic utilization (87%), the usage of DSP blocks (76%) and RAM blocks (62%) is high too. Unrolling loops of sums requires replicating computation elements to have a simultaneous processing inside a loop, this explains the logic element utilization. The optimization report shows that the reference image patch and search region memories are replicated 3 times to support multiple access from the unrolled loop, this explains the usage of RAM blocks (62%). Reducing the number of division to 1, optimizes greatly the usage of the FPGA resources, the only floating point division operation used takes 14 DSP of the 87 DSPs in the FPGA (16%).

Compilation report		
ALUTs	28433	
Registers	77650	
Logic utilisation	27978 / 32070	(87%)
I/O pins	115 / 457	(25%)
DSP blocks	66 / 87	(76%)
Memory bits	1095946 / 4065280	(27%)
RAM blocks	246 / 397	(62%)
Kernel fmax (MHz)	125.42	

Table 4.3: Compilation report

Figure 4.10 shows the comparison between the software and OpenCL implementations in terms of execution time, versus the number of iterations needed to scan a search region by the reference image patch. The OpenCL implementation has a short time of processing comparing to the software implementation, and the differences increases with the number of iterations.

To evaluate the OpenCL implementation with the SLAM, we integrate it inside the SLAM by replacing the searching process and the *Correlate2_warning* function. The SLAM is executed to process 300 frames from a 640×480 video sequences. It is configured to keep a maximum of 15 visible features and update 15 features per frame. A handheld camera is used to record the video sequences. These sequences are used to compare the performance of the proposed OpenCL implementation with the pure software implementation.

Using an FPGA circuit to accelerate a function can affect the accuracy of the results. This is due to the change in the order of the operations (balanced tree operations) and the reduction in rounding operations. To evaluate the accuracy of the OpenCL implementation, we compared the results obtained from the OpenCL implementation with those obtained from the software implementation. The NSSD mean error at the output of the accelerated function is about 3.12×10^{-3} with a relative error about 0.092%. The mean error of the position of the matched patch is about 0.091 pixels with

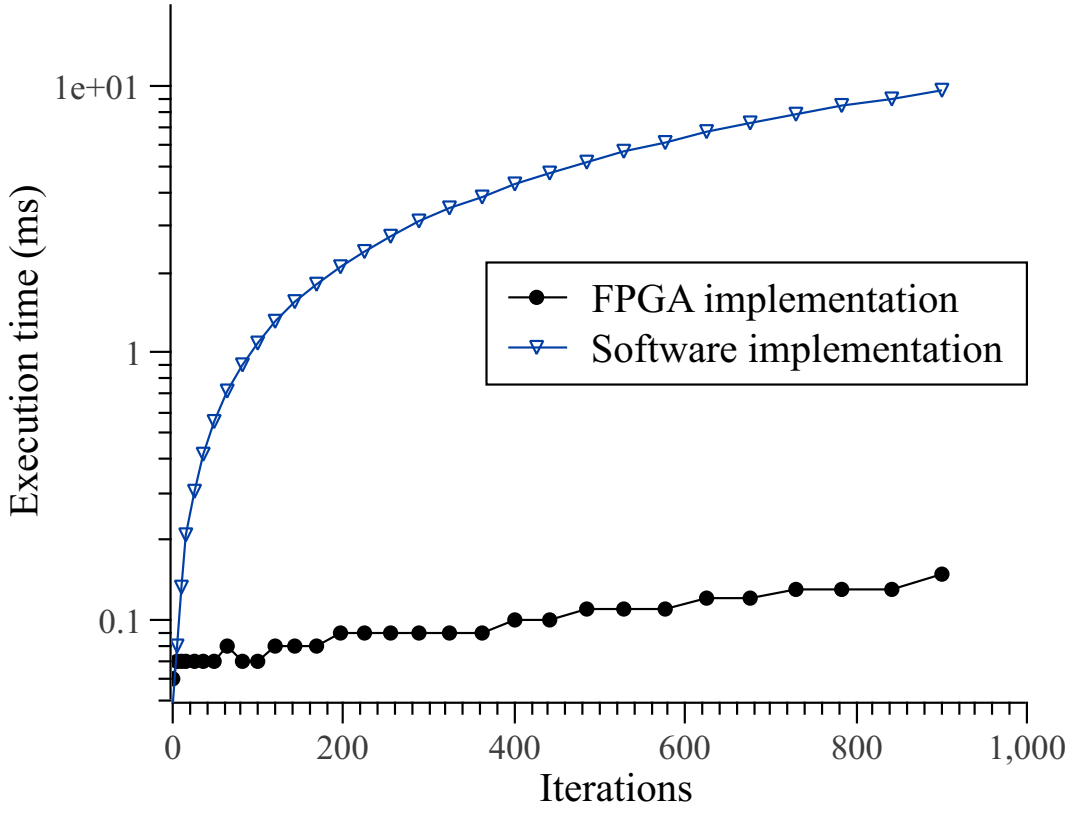


Figure 4.10: Processing time

some irregular results having a maximum error of 34.132 pixels. These errors at the output of the accelerated function have a small impact on the SLAM results in some positions. The calculated mean error of the trajectory of the OpenCL implementation is 0.31 *cm*

The software implementation has an average time of processing of 33.76 *ms* per frame, while the OpenCL implementation has an average of 17.5 *ms* per frame. This means that the OpenCL implementation can operate at a performance up to 57.1 *FPS* and with an acceleration of 1.93 times faster than the software implementation. With 55.9% of the processing time occupied by the *correlate2_warning* function, the theoretical maximum acceleration never achieved is 2.27.

The search regions size measured during the experiment is typically 20 to 24 pixels across. In the software implementation the *correlate2_warning* function has an average processing time of 18.87 *ms* per frame, which is equivalent to an average of 1258 μs per feature.

Table 4.4: Comparison of some SLAM implementation with ours

Works	year	HW type	Full. emb.	Prog. Lang.	Cost	Resolution	Frame rate
Fang *et al.* [32]	2018	FPGA	Yes	HDL	High	640x480	67 fps
Gu *et al.* [29]	2015	FPGA	Yes	HDL	High	*NM*	31 fps
Liu *et al.* [33]	2019	FPGA	No	HDL	Medium	640x480	31.45 fps
Abouzahir *et al.* [34]	2018	PC + FPGA	No	OpenCL	Medium	320x240	102.14 fps
Boikos *et al.* [35]	2017	SoC-FPGA	Yes	HLS	Low	320x240	22 fps
Our	2020	SoC-FPGA	Yes	OpenCL	Low	640x480	57.1 fps

In the FPGA implementation, the average time occupied by the function is 2.61 ms, which is equivalent to an average of 174 μs per feature. This value does not correspond with the result obtained from the profiler which is around 70 to 90 μs. This is due to two reasons: first, the data transfer time from the host memory to the device global memory is not measured, since the kernel has not started yet and so the profiler too. The second reason is the *clFinish()* function that blocks the host program until all the device events are completed. The achieved acceleration of the *correlate2_warning* inside the SLAM is around 7 times faster than the software implementation.

In terms of energy consumption, the power consumption of the SLAM system is estimated using the PowerPlay Power Analyzer from the Intel Quartus tool. The estimated power consumption is 2663.76 mW for the OpenCL implementation versus 1392.92 mW for the software implementation. This difference in terms of power consumption is due to the FPGA accelerating circuit. However in terms of power consumption per frame this difference is reduced, and the power consumption per frame of the OpenCL implementation is about 46.65$mJ/frame$.

The implementation presents many advantages compared to other works in the field. Table 4.4 shows a comparison of some remarkable research works of implementing SLAM on embedded systems with ours.

In our case, we used a low-cost hardware platform to implement SLAM, compared with other works where high-cost [32] [29] or mid-range platforms [33] [34] are used. This is suitable for using the SLAM on low-cost applications. The SoC-FPGA used in this work offers the advantage of using a mobile hard processor in the same chip with the FPGA, and avoids the use of the FPGA resources to implement a soft-

processor [29] [32], or the need for a distant host [34].
The use of OpenCL as a high language to implement the SLAM reduces the development time, and in contrast with HLS used in [35], it is a cross-platform language that can be used on other computing platforms.

4.7 Conclusion

In this section, we presented a case study of implementing an EKF-SLAM using the proposed platform. To achieve this, we first integrated an existing EKF-SLAM into the platform. After profiling the SLAM program, we extracted the most time-consuming part. The analysis of this part of the program helps us in the hardware/software partitioning step. After taking the partitioning decision, the hardware accelerator is designed using the OpenCL.

We listed the design constraints as well as the optimization techniques used to optimize the design. After designing the hardware accelerator, we compared the results using different types of optimization techniques. The optimized accelerator is used in a complete SLAM program and evaluated on the proposed platform.

Finally, a comparison table is made to compare the proposed platform with the implementation works cited in the state of the art in terms of performance, used tools, and cost.

Algorithm 1 Basic NSSD Algorithm

$\bar{f} \leftarrow 0$

$\bar{t} \leftarrow 0$

for $x \in [0; x_0], y \in [0; y_0]$ **do**

    $\bar{f} \leftarrow \bar{f} + f(x, y)$

    $\bar{t} \leftarrow \bar{t} + t(x, y)$

end for

$\bar{f} \leftarrow \bar{f} / (x_0 \times y_0)$

$\bar{t} \leftarrow \bar{t} / (x_0 \times y_0)$

$var_f \leftarrow 0$

$var_t \leftarrow 0$

for $x \in (0; x_0], y \in [0; y_0]$ **do**

    $var_f \leftarrow var_f + (f(x, y) - \bar{f})^2$

    $var_t \leftarrow var_t + (t(x, y) - \bar{t})^2$

end for

$var_f \leftarrow var_f / (x_0 \times y_0)$

$var_t \leftarrow var_t / (x_0 \times y_0)$

$\sigma_f \leftarrow \sqrt{var_f}$

$\sigma_t \leftarrow \sqrt{var_t}$

$NSSD \leftarrow 0$

for $x \in (0; x_0], y \in [0; y_0]$ **do**

    $NSSD \leftarrow NSSD + \left(\frac{f(x,y) - \bar{f}}{\sigma_f} - \frac{t(x,y) - \bar{t}}{\sigma_t} \right)^2$

end for

$NSSD \leftarrow NSSD / (x_0 \times y_0)$

Algorithm 2 Implemented NSSD Algorithm

$S_f \leftarrow 0$

$S_t \leftarrow 0$

$S_{f^2} \leftarrow 0$

$S_{t^2} \leftarrow 0$

$S_{f \times t} \leftarrow 0$

for $x \in [0; x_0], y \in [0; y_0]$ **do**

    $S_f \leftarrow S_f + f(x, y)$

    $S_t \leftarrow S_t + t(x, y)$

    $S_{f^2} \leftarrow S_f + (f(x, y))^2$

    $S_{t^2} \leftarrow S_t + (t(x, y))^2$

    $S_{f \times t} \leftarrow S_{f \times t} + (f(x, y) \times t(x, y))$

end for

$\bar{f} \leftarrow S_f / (x_0 \times y_0)$

$\bar{t} \leftarrow S_t / (x_0 \times y_0)$

$var_f \leftarrow S_{f^2} / n - \bar{f}^2$

$var_t \leftarrow S_{t^2} / n - \bar{t}^2$

$\sigma_f \leftarrow \sqrt{var_f}$

$\sigma_t \leftarrow \sqrt{var_t}$

$k = \bar{f} / \sigma_f - \bar{t} / \sigma_t$

$NSSD \leftarrow \frac{S_{f2}}{var_f} + \frac{S_{t2}}{var_t} + n \times k^2 - 2 \times \frac{S_{f \times t}}{\sigma_f \times \sigma_t} - 2 \times \frac{S_f \times k}{\sigma_f} + 2 \times \frac{S_t \times k}{\sigma_t}$

$NSSD \leftarrow NSSD / (x_0 \times y_0)$

Algorithm 3 *elliptical_search* algorithm

1: $halfwidth, halfweight \leftarrow$ Calculate the dimension of the search box
2: $ucentre, vcentre \leftarrow$ Calculate the center of the box
3: $urelstart, urelfinish, vrelstart, vrelfinish \leftarrow$ Check the limits aren't outside the image
4: **for** $\{urel, vrel\} \leftarrow \{(urelstart : urelfinish), (vrelstart : vrelfinish)\}$ **do**
5: $\quad corr \leftarrow correlate2\_warning(feature, image(urel + ucentre, vrel + vcentre))$
6: $\quad$ **if** $corr < corrmax$ **then**
7: $\quad\quad corrmax \leftarrow corr$
8: $\quad\quad \{u, v\} \leftarrow \{urel + ucentre, vrel + vcentre\}$
9: $\quad$ **end if**
10: **end for**

Algorithm 4 Implemented NSSD Algorithm with OpenCL

19: $G \leftarrow (\bar{f} \times \sigma_t) - (\bar{t} \times \sigma_f)$
20: $NSSD \leftarrow \frac{Numerator}{Denominator}$
21: $NSSD \leftarrow NSSD \times n_{inv}$

GENERAL CONCLUSION

The main purpose of this thesis is to develop a prototyping platform to help design and build embedded systems dedicated to SLAM. This platform is centered on SoCs and open-source bricks. SLAM algorithms are complex and are initially developed on powerful machines. The implementation of SLAM algorithms on embedded systems requires the use of an architecture and an implementation methodology that take into account the constraints of embedded systems.

In this context, a study of some implementations of SLAM algorithms on embedded targets was made. This study directed us towards the use of FPGA based SoC and High level languages.

In the second chapter of the thesis, we explored the embedded systems design approaches. As we talk about platform, a meet-in-th-middle approach is the convenient approach in our case. To efficiently partition the SLAM program between the Hardware and Software parts of the platform, we used Hw/Sw co-design methodology.

The proposed prototyping platform is composed of two parts hardware part and software part. The hardware part of the proposed platform is based on SoC-FPGA. An SoC-FPGA integrates a hard CPU and an FPGA fabric in the same chip. This allows us to benefit from the high parallelism and customization of the FPGA with a high bandwidth data transfer with the CPU. A DE1-SoC board equipped with Cyclone V-SoC, a low-cost SoC-FPGA from IntelFPGA is used as the technology target. This type of SoC-FPGA can be programmed using OpenCL.

The software part is based on the Linux operating system. We used Buildroot, an automated system building tool, to build a light and customized Linux system. It allows also reproducing and retargeting to other platforms.

A **case study** carried on implementation of an EKF-SLAM on the SoC-FPGA and has shown that the result of the platform is comparable, from a performance point of view, with other SLAM implementation work.

Future Work

Prototyping platforms for embedded SLAM are not common as it is mentioned above. These platforms can have many applications in research or in industry. At the end of this thesis, we consider the following perspectives:

- It would be interesting to integrate other types of SLAM. Many SLAM implementations exist as open-source such as ORB-SLAM, LSD-SLAM, etc. They present other types of SLAMs and their implementation on the platform can be investigated.
- In this work, we used OpenCL to design the FPGA accelerator. Although it has several advantages such as porting to other platforms and a high-level design, the use of HLS can be studied and compared with OpenCL.
- As we aim for multi-targeting, the use of different embedded targets is a validation step of the proposed platform. The platform can also be tested with real mobile robots, equipped with many types of sensors in a real environment.

BIBLIOGRAPHY

[1] M. L. BEN MESSAOUD, *PLATEFORME DE PROTOTYPAGE POUR SLAM EMBARQUE*. Mémoire magister, Ecole Militaire Polytechnique, 2018.

[2] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. Kelly, A. J. Davison, M. Luján, M. F. O'Boyle, G. Riley, N. Topham, and S. Furber, "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2015-June, pp. 5783–5790, oct 2015.

[3] B. Bodin, H. Wagstaff, S. Saeedi, L. Nardi, E. Vespa, J. Mawer, A. Nisbet, M. Luj, S. Furber, A. J. Davison, P. H. J. Kelly, and M. F. P. O. Boyle, "SLAMBench2 : Multi-Objective Head-to-Head Benchmarking for Visual SLAM," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2018.

[4] M. Bujanca, P. Gafton, S. Saeedi, A. Nisbet, B. Bodin, O. Michael F.P., A. J. Davison, K. Paul H.J., G. Riley, B. Lennox, M. Lujan, and S. Furber, "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)*, pp. 6351–6358, IEEE, may 2019.

[5] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: Part I," *IEEE Robotics and Automation Magazine*, vol. 13, pp. 99–108, jun 2006.

[6] L. D'Alfonso, A. Grano, P. Muraca, and P. Pugliese, "A polynomial based SLAM algorithm for mobile robots using ultrasonic sensors - Experimental results," in *2013 16th International Conference on Advanced Robotics, ICAR 2013*, pp. 1–6, IEEE, nov 2013.

[7] C. F. Gauss, *Theoria motus corporum coelestium in sectionibus conicis solem ambientium. (Theory of the motion of the celestial bodies moving around the Sun in conic sections)*. 1809.

[8] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Fluids Engineering, Transactions of the ASME*, vol. 82, pp. 35–45, mar 1960.

[9] D. C. Brown, "The Bundle Adjustment — Progress and prospects," in *International Archives of Photogrammetry, Remote Sensing, and Spatial Information Sciences*, vol. 21, p. 33 pp, 1976.

[10] R. C. Smith and P. Cheeseman, "On the Representation and Estimation of Spatial Uncertainty," *The International Journal of Robotics Research*, vol. 5, pp. 56–68, dec 1986.

[11] H. F. Durrant-Whyte, "Uncertain Geometry in Robotics," *IEEE Journal on Robotics and Automation*, vol. 4, no. 1, pp. 23–31, 1988.

[12] H. Durrant-Whyte, D. Rye, and E. Nebot, "Localization of Autonomous Guided Vehicles," in *Robotics Research*, pp. 613–625, London: Springer London, 1996.

[13] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[14] R. Smith, M. Self, and P. Cheeseman, "A stochastic map for uncertain spatial relationships," *Proceedings of the 4th international symposium on Robotics Research*, no. 0262022729, pp. 467–474, 1988.

[15] K. Murphy and S. Russell, "Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks," in *Sequential Monte Carlo Methods in Practice*, pp. 499–515, New York, NY: Springer New York, 2001.

[16] M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit, and Others, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," pp. 593–598, 2002.

[17] J. Folkesson and H. Christensen, "Graphical SLAM - a self-correcting map," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, vol. 1, pp. 383–390 Vol.1, IEEE, apr 2004.

[18] A. J. Davison, "Real-time simultaneous localisation and mapping with a single camera," in *Proceedings Ninth IEEE International Conference on Computer Vision*, pp. 1403–1410 vol.2, IEEE, oct 2003.

[19] C. Hertzberg, R. Wagner, O. Birbach, T. Hammer, and U. Frese, "Experiences in building a visual SLAM system from open source components," in *2011 IEEE International Conference on Robotics and Automation*, pp. 2644–2651, IEEE, may 2011.

[20] 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, pp. 1255–1262, oct 2016.

[21] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard, "An evaluation of the RGB-D SLAM system," pp. 1691–1696, 2012.

[22] A. Davison, I. Reid, N. Molton, and O. Stasse, "MonoSLAM: Real-Time Single Camera SLAM," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, pp. 1052–1067, jun 2007.

[23] A. Gonzalez, J.-M. Codol, and M. Devy, "A C-embedded algorithm for real-time monocular SLAM," in *18th IEEE International Conference on Electronics, Circuits, and Systems*, pp. 665–668, IEEE, dec 2011.

[24] B. Vincke, A. Elouardi, and A. Lambert, "Real time simultaneous localization and mapping: towards low-cost multiprocessor embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2012, p. 5, dec 2012.

[25] B. Vincke, A. Elouardi, A. Lambert, and A. Dine, "SIMD and OpenMP optimization of EKF-SLAM," in *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pp. 712–716, IEEE, apr 2014.

[26] M. Abouzahir, A. Elouardi, S. Bouaziz, R. Latif, and A. Tajer, "Large-scale monocular FastSLAM2.0 acceleration on an embedded heterogeneous architecture," *EURASIP Journal on Advances in Signal Processing*, vol. 2016, p. 88, dec 2016.

[27] V. Bonato, E. Marques, and G. A. Constantinides, "A Floating-point Extended Kalman Filter Implementation for Autonomous Mobile Robots," *Journal of Signal Processing Systems*, vol. 56, pp. 41–50, jul 2009.

[28] D. T. Tertei, J. Piat, and M. Devy, "FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pp. 1–6, IEEE, dec 2014.

[29] M. Gu, K. Guo, W. Wang, Y. Wang, and H. Yang, "An FPGA-based real-Time simultaneous localization and mapping system," in *2015 International Conference on Field Programmable Technology, FPT 2015*, no. 61373026, pp. 200–203, IEEE, dec 2016.

[30] M. Y. I. Idris, H. Arof, N. M. Noor, E. M. Tamil, and Z. Razak, "A co-processor design to accelerate sequential monocular SLAM EKF process," *Measurement*, vol. 45, no. 8, pp. 2141–2152, 2012.

[31] D. Botero, A. Gonzalez, and M. Devy, "Architecture embarquée pour le SLAM monoculaire," in *RFIA 2012 (Reconnaissance des Formes et Intelligence Artificielle)*, (Lyon, France), jan 2012.

[32] 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*, vol. 2018-Janua, pp. 275–278, IEEE, dec 2018.

[33] R. Liu, J. Yang, Y. Chen, and W. Zhao, "ESLAM: An energy-efficient accelerator for real-time ORB-SLAM on FPGA platform," in *Proceedings - Design Automation Conference*, (New York, New York, USA), pp. 1–6, ACM Press, 2019.

[34] M. Abouzahir, A. Elouardi, R. Latif, S. Bouaziz, and A. Tajer, "Embedding SLAM algorithms: Has it come of age?," *Robotics and Autonomous Systems*, vol. 100, pp. 14–26, feb 2018.

[35] K. Boikos and C.-S. Bouganis, "A high-performance system-on-chip architecture for direct tracking for SLAM," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7, IEEE, sep 2017.

[36] K. Boikos and C.-S. Bouganis, "A Scalable FPGA-Based Architecture for Depth Estimation in SLAM," in *Applied Reconfigurable Computing* (C. Hochberger, , B. Nelson, , A. Koch, , R. Woods, , and P. Diniz, eds.), vol. 11444 LNCS, pp. 181–196, Springer International Publishing, feb 2019.

[37] D. Törtei Tertei, J. Piat, and M. Devy, "FPGA design of EKF block accelerator for 3D visual SLAM," *Computers & Electrical Engineering*, vol. 55, pp. 123–137, oct 2016.

[38] K. Boikos, *Custom hardware architectures for embedded high-performance and low-power SLAM*. PhD thesis, 2019.

[39] A. Shaout, A. H. El-mousa, and K. Mattar, "Specification and Modeling of HW/SW CO-Design for Heterogeneous Embedded Systems," *Lecture Notes in Engineering and Computer Science*, vol. 2176, no. 1, pp. 273–278, 2009.

[40] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware / Software Approach*. Wiley, 2001.

[41] R. Dömer, D. D. Gajski, and J. Zhu, "Specification and Design of Embedded Systems," *it - Information Technology*, vol. 40, jan 1998.

[42] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, 2000.

[43] N. Du Lac, "Prototypage des applications temps réel embarquées," *Ref : TIP660WEB - "Automatique et ingénierie système"*, dec 2005.

[44] P. Marwedel, *Embedded System Design*. Embedded Systems, Cham: Springer International Publishing, 2021.

[45] A. Sangiovanni-Vincentelli, "Electronic-system design in the automobile industry," *IEEE Micro*, vol. 23, pp. 8–18, may 2003.

[46] J. Tang, S. Liu, and J.-L. Gaudiot, "Embedded Systems Architecture for SLAM Applications," p. 4, 2017.

[47] S. Furber, *ARM System-on-Chip Architecture*. USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2000.

[48] A. Dine, *Localisation et cartographie simultanées par optimisation de graphe sur architectures hétérogènes pour l'embarqué*. PhD thesis, Université Paris Saclay (COmUE), 2016.

[49] M. Y. I. Idris, N. M. Noor, E. M. Tamil, Z. Razak, and H. Arof, "Parallel Matrix Multiplication Design for Monocular SLAM," in *Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*, pp. 492–497, IEEE, may 2010.

[50] S. Kundu and S. Chattopadhyay, *Network-on-Chip The Next Generation of System-on-Chip Integration*. CRC Press, sep 2015.

[51] "Intel SoC FPGAs Programmable Devices.". *Available* https://www.intel.com/content/www/us/en/products/programmable/soc.html (*consulted* 2019-12-03)

[52] "Xilinx SoCs, MPSoCs and RFSoCs.". *Available* https://www.xilinx.com/products/silicon-devices/soc.html

[53] "SoC FPGAs, Microsemi.". *Available* https://www.microsemi.com/product-directory/fpga-soc/1639-soc-fpgas (*consulted* 2021-03-10)

[54] Terasic, *DE1-SoC User Manual*.

[55] "Vitis High-Level Synthesis.". *Available* https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html (*consulted* 15-03-2021)

[56] "High-Level Synthesis Compiler - Intel® HLS Compiler.". *Available* https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html (*consulted* 2021-03-15)

[57] "HDL Coder - MATLAB & Simulink.". *Available* https://www.mathworks.com/products/hdl-coder.html (*consulted* 2021-03-15)

[58] "OpenCL 1.0 Specification," tech. rep., 2009. *Available* https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf.

[59] "Intel® FPGA SDK for OpenCL™ Software Technology.". *Available* https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html (*consulted* 2019-11-27)

[60] Xilinx, "SDAccel Development Environment.". *Available* https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html (*consulted* 2019-11-27)

[61] "Intel® FPGA SDK for OpenCL™, Programming Guide.". *Available* https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl_programming_guide-17-1.pdf (*consulted* 2019-11-2)

[62] F. PÉTROT, "OS embarqués," *Techniques de l'Ingénieur, Ref : TIP402WEB - "Technologies logicielles Architectures des systèmes"*, jan 2011.

[63] P. FICHEUX, "Linux embarqué," *Techniques de l'Ingénieur, Ref : TIP402WEB - "Technologies logicielles Architectures des systèmes"*, aug 2019.

[64] P. FICHEUX, *Linux embarqué "Avec deux études de cas"*. eyrolles ed., 2005.

[65] D. Beazley, B. Ward, and I. Cooke, "The inside story on shared libraries and dynamic loading," *Computing in Science & Engineering*, vol. 3, no. 5, pp. 90–97, 2001.

[66] G. Medioni and S. B. Kang, *Emerging topics in computer vision*. USA: Prentice Hall PTR, imsc press ed., 2004.

[67] "GitHub - hanmekim/SceneLib2: SceneLib2 is an open-source C++ library for SLAM originally designed and implemented by Professor Andrew Davison at Imperial College London..". *Available* https://github.com/hanmekim/SceneLib2 (*consulted* 2021-04-01)

[68] 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, pp. 1147–1163, oct 2015.

[69] "GitHub - raulmur/ORB_SLAM2: Real-Time SLAM for Monocular, Stereo and RGB-D Cameras, with Loop Detection and Relocalization Capabilities.". *Available* https://github.com/raulmur/ORB_SLAM2 (*consulted* 2021-04-01)

[70] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-Scale Direct Monocular SLAM," in *Computer Vision – ECCV 2014*, pp. 834–849, 2014.

[71] "GitHub - tum-vision/lsd_slam: LSD-SLAM.". *Available* https://github.com/tum-vision/lsd_slam (*consulted* 2021-04-01)

[72] "MRPT.". *Available* https://www.mrpt.org/ (*consulted* 2021-04-01)

[73] J. L. Blanco, J. A. Fernández-Madrigal, and J. González, "Efficient probabilistic Range-Only SLAM," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, pp. 1017–1022, IEEE, 2008.

[74] F. Moreno, J. Blanco, and J. Gonzalez, "Stereo vision specific models for particle filter-based SLAM," *Robotics and Autonomous Systems*, vol. 57, pp. 955–970, sep 2009.

[75] J.-L. Blanco, J. Gonzalez, and J.-A. Fernandez-Madrigal, "An optimal filtering algorithm for non-parametric observation models in robot localization," in *2008 IEEE International Conference on Robotics and Automation*, pp. 461–466, IEEE, may 2008.

[76] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, jan 2006.

[77] R. Kummerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o: A general framework for graph optimization," in *2011 IEEE International Conference on Robotics and Automation*, pp. 3607–3613, IEEE, may 2011.

[78] T. Petazzoni, "Buildroot : a nice, simple and efficient embedded Linux build system," tech. rep., 2012. *Available* https://elinux.org/images/9/9e/Buildroot2.pdf.

[79] S. P. Reiss, "Event-based performance analysis," in *Proceedings - IEEE Workshop on Program Comprehension*, vol. 2003-May, pp. 74–83, IEEE Comput. Soc, 2003.

[80] "Sampling Profiler - Overview | AQTime Documentation.". *Available* https://support.smartbear.com/aqtime/docs/reference/profilers/sampling/about.html (*consulted* 2021-05-05)

[81] S. Bouhoun, R. Sadoun, and M. Adnane, "OpenCL implementation of a SLAM system on an SoC-FPGA," *Journal of Systems Architecture*, vol. 111, p. 101825, dec 2020.

[82] A. J. Davison and D. W. Murray, "Simultaneous localization and map-building using active vision," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 7, pp. 865–880, 2002.

[83] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, (New York, New York, USA), p. 483, ACM Press, 1967.

[84] Intel FPGA, "Intel ® FPGA SDK for OpenCL ™ Pro Edition Best Practices Guides," tech. rep., 2018. *Available* https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf.

[85] S. J. Parker and V. A. Chouliaras, "An OpenCL software compilation framework targeting an SoC-FPGA VLIW chip multiprocessor," *Journal of Systems Architecture*, vol. 68, pp. 17–37, 2016.

[86] K. Shata, M. K. Elteir, and A. A. EL-Zoghabi, "Optimized implementation of OpenCL kernels on FPGAs," *Journal of Systems Architecture*, vol. 97, pp. 491–505, aug 2019.

[87] A. Rodríguez, A. Navarro, R. Asenjo, F. Corbera, R. Gran, D. Suárez, and J. Nunez-Yanez, "Exploring heterogeneous scheduling for edge computing with CPU and FPGA MPSoCs," *Journal of Systems Architecture*, vol. 98, no. June, pp. 27–40, 2019.

APPENDIX A

SCENELIB2 AND PANGOLIN CONFIGURATION FILES

A.1 Pangolin package files

Listing A.1: ./package/Pangolin/Config.in

```
config BR2_PACKAGE_PANGOLIN
        bool "Pangolin"
        depends on BR2_PACKAGE_BOOST
        depends on BR2_PACKAGE_HAS_LIBGL
        depends on BR2_PACKAGE_LIBGLEW
        depends on BR2_PACKAGE_LIBFREEGLUT
        depends on BR2_PACKAGE_LIBGLU
        depends on BR2_PACKAGE_EIGEN
        depends on BR2_PACKAGE_FFMPEG
        depends on BR2_PACKAGE_LIBUSB
        depends on BR2_PACKAGE_LIBPNG
        depends on BR2_PACKAGE_JPEG
        depends on BR2_PACKAGE_TIFF
        help
          Pangolin is a lightweight portable rapid development
             library for managing
          OpenGL display / interaction and abstracting video input.

          http://www.stevenlovegrove.com/?id=44
          https://github.com/stevenlovegrove/Pangolin
```

```
comment "Pangolin need Libraries"
        depends on !(BR2_PACKAGE_BOOST && BR2_PACKAGE_HAS_LIBGL &&
            BR2_PACKAGE_LIBGLEW && BR2_PACKAGE_LIBFREEGLUT &&
            BR2_PACKAGE_LIBGLU && BR2_PACKAGE_EIGEN &&
            BR2_PACKAGE_FFMPEG && BR2_PACKAGE_LIBUSB &&
            BR2_PACKAGE_LIBPNG && BR2_PACKAGE_JPEG &&
            BR2_PACKAGE_TIFF)
```

Listing A.2: ./package/Pangolin/Pangolin.mk

```
#############################################################
#                                                           #
# Pangolin                                                  #
#                                                           #
#############################################################

PANGOLIN_VERSION = 021ed52ca8e355abf7cd2c783e12a316fc07218d
PANGOLIN_SITE = $(call github,stevenlovegrove,Pangolin,$(
    PANGOLIN_VERSION))
PANGOLIN_INSTALL_STAGING = YES
PANGOLIN_INSTALL_TARGET = YES
PANGOLIN_CONF_OPTS = -DCPP11_NO_BOOST=ON -DBUILD_SHARED_LIBS=ON -
    DCMAKE_BUILD_TYPE=Release
PANGOLIN_DEPENDENCIES = boost libgl libglew libfreeglut libglu eigen
    ffmpeg libusb libpng libjpeg tiff opencv

$(eval $(cmake-package))
```

A.2 SceneLib2 package files

Listing A.3: ./package/SceneLib2/Config.in

```
config BR2_PACKAGE_SCENELIB2
        bool "SceneLib2"
        depends on BR2_PACKAGE_BOOST
        depends on BR2_PACKAGE_HAS_LIBGL
        depends on BR2_PACKAGE_LIBGLEW
        depends on BR2_PACKAGE_LIBFREEGLUT
        depends on BR2_PACKAGE_LIBGLU
```

```
        depends on BR2_PACKAGE_EIGEN
        depends on BR2_PACKAGE_FFMPEG
        depends on BR2_PACKAGE_LIBUSB
        depends on BR2_PACKAGE_LIBPNG
        depends on BR2_PACKAGE_JPEG
        depends on BR2_PACKAGE_TIFF
        depends on BR2_PACKAGE_PANGOLIN
        help
          SceneLib2 is an open-source C++ library for SLAM originally
               designed and
          implemented by Andrew Davison and colleagues at the
              University of Oxford.

          http://hanmekim.blogspot.com/2012/10/scenelib2-monoslam-
              open-source-library.html
          https://github.com/hanmekim/SceneLib2

comment "Scenelib needs Libraries"
        depends on !(BR2_PACKAGE_BOOST && BR2_PACKAGE_HAS_LIBGL &&
            BR2_PACKAGE_LIBGLEW && BR2_PACKAGE_LIBFREEGLUT &&
            BR2_PACKAGE_LIBGLU && BR2_PACKAGE_EIGEN &&
            BR2_PACKAGE_FFMPEG && BR2_PACKAGE_LIBUSB &&
            BR2_PACKAGE_LIBPNG && BR2_PACKAGE_JPEG &&
            BR2_PACKAGE_TIFF && BR2_PACKAGE_PANGOLIN)
```

Listing A.4: ./package/SceneLib2/SceneLib2.mk

```
#############################################################
#                                                           #
# SceneLib2                                                 #
#                                                           #
#############################################################

SCENELIB2_VERSION = 39991c61becbebe0c66601fe5c14f8155264be4e
SCENELIB2_SITE = $(call github,hanmekim,SceneLib2,$(SCENELIB2_VERSION
    ))
SCENELIB2_INSTALL_STAGING = YES
SCENELIB2_INSTALL_TARGET = YES
SCENELIB2_CONF_OPTS = -DCMAKE_BUILD_TYPE=Release
SCENELIB2_DEPENDENCIES = boost libgl libglew libfreeglut libglu eigen
```

```
    ffmpeg libusb libpng libjpeg tiff opencv Pangolin

$(eval $(cmake-package))
```

A.3 MonoSLAM package files

Listing A.5: ./package/SLAM/Config.in

```
config BR2_PACKAGE_SLAM
        bool "SLAM"
        depends on BR2_PACKAGE_BOOST
        depends on BR2_PACKAGE_HAS_LIBGL
        depends on BR2_PACKAGE_LIBGLEW
        depends on BR2_PACKAGE_LIBFREEGLUT
        depends on BR2_PACKAGE_LIBGLU
        depends on BR2_PACKAGE_EIGEN
        depends on BR2_PACKAGE_FFMPEG
        depends on BR2_PACKAGE_LIBUSB
        depends on BR2_PACKAGE_LIBPNG
        depends on BR2_PACKAGE_JPEG
        depends on BR2_PACKAGE_TIFF
        depends on BR2_PACKAGE_PANGOLIN
        depends on BR2_PACKAGE_SCENELIB2
        help
          SLAM is a tiny application of SLAM using SceneLib2 Library

          http://eln.enp.edu.dz
          salah.bouhoun@g.enp.edu.dz

comment "SLAM needs Libraries"
        depends on !(BR2_PACKAGE_SCENELIB2 && BR2_PACKAGE_BOOST &&
            BR2_PACKAGE_HAS_LIBGL && BR2_PACKAGE_LIBGLEW &&
            BR2_PACKAGE_LIBFREEGLUT && BR2_PACKAGE_LIBGLU &&
            BR2_PACKAGE_EIGEN && BR2_PACKAGE_FFMPEG &&
            BR2_PACKAGE_LIBUSB && BR2_PACKAGE_LIBPNG &&
            BR2_PACKAGE_JPEG &&  BR2_PACKAGE_TIFF &&
            BR2_PACKAGE_PANGOLIN)
```

Listing A.6: ./package/SLAM/SLAM.mk

```
############################################################
#                                                          #
# SLAM                                                     #
#                                                          #
############################################################

SLAM_SOURCE = SLAM-0.0.tar.gz
SLAM_SITE = /home/salah/My_Projects/SceneLib/SceneLib2_test/
    my_example
SLAM_SITE_METHOD= = local
SLAM_INSTALL_STAGING = NO
SLAM_INSTALL_TARGET = YES
SLAM_CONF_OPTS = -DCMAKE_BUILD_TYPE=Release
SLAM_DEPENDENCIES = boost libgl libglew libfreeglut libglu eigen
    ffmpeg libusb libpng libjpeg tiff opencv Pangolin SceneLib2

$(eval $(cmake-package))
```

A.4 Configuration Menu file

Listing A.7: ./packageConfig.in

```
menu "Target packages"
.
.
.

menu "My Applications"
source "package/Pangolin/Config.in"
source "package/SceneLib2/Config.in"
source "package/SLAM/Config.in"
endmenu

endmenu
```