TECHNICAL UNIVERSITY OF CRETE
DIPLOMA THESIS

Acceleration on a Reconfigurable Logic Platform of the ORB-SLAM2 Algorithm for Autonomous Underwater Vehicles

Author:
Maria MARAGKAKI

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Aggelos BLETSAS
Dr. Euripides SOTIRIADES

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

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

November 24, 2022

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Acceleration on a Reconfigurable Logic Platform of the ORB-SLAM2 Algorithm for Autonomous Underwater Vehicles

by Maria MARAGKAKI

Over the last few years the use of visual Simultaneous Localization and Mapping (vSLAM) algorithms gained widespread development and use in all areas, e.g., self-driving cars, robots, aerial drones, autonomous underwater vehicles and more. Autonomous underwater vehicles have various applications ranging from garbage collection in shallow ports and port mapping, to finding holes in fishery nets. Underwater scenarios are complex and costly due to the large amount of sensors needed such as Doppler Velocity Log (DVL) sensors, depth sensors etc. The use of vSLAM algorithms in these applications is important, leading to a need for real time implementation on low-power platforms. In this case either a platform with a fast processor but with high power consumption is used in order to have the real time implementation, or a low-power consumption processor with lower processing power in frames per second is used, resulting to undesirably slow system performance. Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) can offer real time implementation with low energy cost. In this thesis we have developed an FPGA-based architecture to accelerate the most time consuming part of the ORB-SLAM2 algorithm, i.e. the Oriented FAST and Rotated BRIEF (ORB) feature extraction part. The proposed architecture requires per image 60% less energy vs. the software implementation of the ORB part of ORB-SLAM2 algorithm, while maintaining competitive performance vs. a high-end processor.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Acceleration on a Reconfigurable Logic Platform of the ORB-SLAM2 Algorithm for Autonomous Underwater Vehicles

by Maria MARAGKAKI

Τα τελευταία χρόνια η χρήση των αλγορίθμων visual Simultaneous Localization and Mapping (vSLAM) απέκτησε ευρεία ανάπτυξη και χρήση σε όλους τους τομείς όπως αυτόνομα αυτοκίνητα, robots , εναέρια drones, αυτόνομα υποβρύχια οχήματα και άλλα. Τα αυτόνομα υποβρύχια οχήματα έχουν πληθώρα εφαρμογών, από τη συλλογή σκουπιδιών σε ρηχά λιμάνια έως την εύρεση τρυπών σε δίχτυα ψαρέματος. Τα υποβρύχια σενάρια είναι περίπλοκα και ακριβά λόγω το μεγάλου αριθμού αισθητήρων που χρειάζονται, όπως, αισθητήρες DVL , αισθητήρες βάθους κλπ. Η χρήση των αλγορίθμων vSLAM σε τέτοιες εφαρμογές είναι σημαντική, οδηγώντας στην ανάγκη μιας πραγματικού χρόνου υλοποίησης σε χαμηλής ενεργειακής κατανάλωσης πλατφόρμες. Σε αυτή τη περίπτωση χρησιμοποιείται είτε μια πλατφόρμα με έναν γρήγορο επεξεργαστή αλλά με υψηλή κατανάλωση ενέργειας προκειμένου να επιτευχθεί η υλοποίηση σε πραγματικό χρόνο, ή μια χαμηλής ενεργειακής κατανάλωσης πλατφόρμα αλλά με την επεξεργασία εικόνων ανα δευτερόλεπτο να είναι μικρότερη, με αποτέλεσμα την ανεπιθύμητα αργή απόδοση του συστήματος. Οι FPGAs καθώς και οι GPUs μπορούν να προσφέρουν υλοποίηση σε πραγματικό χρόνο με χαμηλό ενεργειακό κόστος. Στην παρούσα διπλωματική αναπτύξαμε μια αρχιτεκτονική βασισμένη σε FPGA προκειμένου να επιταχύνουμε το πιο χρονοβόρο κομμάτι του αλγορίθμου ORB-SLAM2 δηλ. Το κομμάτι της εξαγωγή των σημείων αναφοράς ORB . Η προτεινόμενη αρχιτεκτονική είναι κατά 60% λιγότερο ενεργειακά απαιτητική ανά εικόνα σε σχέση με την υλοποίηση του αλγορίθμου σε λογισμικό του ORB κομματιού του αλγορίθμου ORB-SLAM2 , διατηρώντας παράλληλα ικανοποιητική απόδοση έναντι ενός επεξεργαστή υψηλής τεχνολογίας.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Apostolos Dollas for his support and guidance throughout the work of this thesis and also for giving me the opportunity of being member of his team and MHL laboratory. Also, I would like to thank Dr. Euripides Sotiriades for his valuable help and support during this thesis.

Furthermore, I would like to thank my thesis committee member, Prof. Aggelos Bletsas, for evaluating my work.

Moreover, I would like to thank the staff of MHL laboratory and special thanks to Mr. Pavlos Malakonakis for his valuable help with the HW tools.

Last but not least, I would like thank all of my friends and family who stood by me and supported me throughout my studies. Thanks all of you for your support.

Maria Maragkaki,
Athens, 2022

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

SLAM	**S**ilmutaneous **L**ocalization **A**nd **M**apping
UAV	**U**nmanned **A**erial **V**ehicle
AUV	**A**utonomous **U**nderwater **V**ehicle
LiDAR	**L**ight **D**etection **A**nd **R**anging
DVL	**D**oppler **V**elocity **L**og
IMU	**I**nertial **M**easurement **U**nit
vSLAM	**V**isual **S**ilmutaneous **L**ocalization **A**nd **M**apping
BA	**B**undle **A**djustment
FAST	**F**eatures **F**rom **A**ccelerated **S**egment**T**est
BRIEF	**B**inary **R**obust **I**ndependent **E**lementary **F**eatures
rBRIEF	**r**otated **B**inary **R**obust **I**ndependent **E**lementary **F**eatures
ORB	**O**riented FAST and **R**otated **B**RIEF
BoW	**B**ag **o**f **W**ords
ROV	**R**emotely **O**perated **V**ehicle
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessor **U**nit
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
LUT	**L**ook **U**p **T**able
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory

Dedicated to my family and friends…

Chapter 1

Introduction

The SLAM algorithm is a fundamental technique of pose estimation and location in the map was build by the surrounding environment. The use of SLAM algorithms has various applications in everyday life. Over the last few years, SLAM algorithms are increasingly used in underwater fields. Underwater SLAM is a complex and costly research area due to the large amount of sensors required. This led to the need to develop algorithms that use a stereo or monocular camera as the only sensor, these algorithms called visual SLAM algorithms (vSLAM). One such algorithm is **ORB-SLAM2** algorithm. ORB method is a feature based method of SLAM algorithms witch is a high effective and robust method. This method combined with a low-power embedded platform can give a real-time implementation with low energy cost witch is the request for underwater vehicles.

1.1 Motivation

Over the years, the applications of algorithms accelerated with FPGAs increased more and more. An application running on a Central Processing Unit (CPUs) may be easier to write and implement, but it is not time efficient due to low parallelism and also not energy efficient due to the high power consumption of CPUs.

Graphics Processing Units (GPUs), provides the ability of high parallelism of the code and there is also relatively easy of their code implementation. However, heir power consumption can be really high. Field Programmable Gate Arrays (FPGAs), on the other hand, apart from ability of high parallelism are considered to be more power efficient solution, because FPGAs consist of only hardware functions while GPUs tend to be highly power consuming as

they need it to facilitate software programmability therefore consist of much gates.

1.2 Scientific Goals and Contributions

The main goal of this diploma thesis is to design and implement an accelerator of the most time consuming part of ORB-SLAM2 algorithm the ORB feature extraction part. We will propose an FPGA accelerator as when this work began the available resources of the laboratory did not meet our needs. This work can also be developed for a GPU such as Jetson which did not exist in the laboratory at the time. We focused at the ORB part of the algorithm because takes about half the time of the algorithm and can achieve high parallelism due to the non-dependence of the frames on each other.

This thesis aims to develop an accelerator that will be applicable to unmanned submarines, so the main goal is to reduce as much as it is possible the power consumption as well as and achieve a real-time implementation. A CPU in such a submarine may be achieve real time implementation but the energy cost is much higher. There are CPUs with lower energy cost but the are not able to process capacity of frames per second. With the use of the FPGA a very good amount of frames per second can be processed as well as and very low power consumption.

The ORB-SLAM2 algorithm in order to process the frames uses the *OpenCV* [1] library. Xilinx provides the corresponding library named *xfOpencv* [2]. One more goal of this thesis is to use the corresponding functions that ORB uses such as FAST for extracting features, resize for implementing the image pyramid. Process the image with these function instead of process the image pixel by pixel and achieve high parallelism in order to compare the results with those from the software implementation.

The ORB-SLAM2 algorithm, studied in this work, was fully designed with a system architecture, simulation, place and route, and performance evaluation from post place-and-route simulations for an actual platform, in this case the ZCU-102 platform from Xilinx.

1.3 Thesis Outline

- **Chapter 2- Theoretical Background:** The theoretical background of SLAM, vSLAM, ORB-SLAM algorithms, with emphasis on the ORB part of ORB-SLAM2 algorithm.
- **Chapter 3 - Related Work:** The related work of Undrwater SLAM algorithm and the harware implementation of ORB-SLAM2 algorithm. Also the datasets used for the thesis approach.
- **Chapter 4- System Modeling:** The system modeling and CPU Analysis.
- **Chapter 5 - System Architecture:** The description of this work's architecture design with FPGA.
- **Chapter 6 - FPGA design:** The tools and platform used in order to implement the design.
- **Chapter 7 - Verification on an FPGA platform and Performance Evaluation:** The results of the design and the verification of the system.
- **Chapter 8 - Conclusions and Related Work:** Conclusion and future work that possible make better this design.

Chapter 2

Theoretical Background

This chapter presents the main purpose of the SLAM algorithm, as well as some of its most basic versions; these versions are described below. Some basic information about the main structure of the SLAM algorithm as well as and the approach of visual SLAM are presented. The purpose and the constraints of Underwater SLAM are analyzed. Lastly, the ORB-SLAM2 which is the main algorithm studied in this thesis is presented, together with theoretical information about the implementation of ORB-SLAM2.

2.1 Simultaneous Localization and Mapping (SLAM)

Over the recent years automation in all areas is increasing rapidly. Self-driving cars, robot vacuum sweepers, unmanned aerial vehicle (UAV) - commonly known as drones, autonomous underwater vehicles (AUV) are used more and more either for civilian and household purposes(self-driving cars, vacuums) or for military purposes (AUVs, UAVs). All these technologies have a common background: the SLAM method.

Simultaneous localization and mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it [3][4]. SLAM algorithms allow the vehicle to map out unknown environments. Engineers use the map information to carry out tasks such as path planning and obstacle avoidance. SLAM algorithms are divided into five main parts: landmark extraction, data association, state estimation, state update and landmark update. Each SLAM algorithm solves every part different for this reason there are many different versions. SLAM algorithms have a wide range of applications. Those algorithms are used in navigation robotic mapping and odometry for virtual reality or augmented reality. They have been implemented

into many fields such as self-driving cars [5], UAVs [6], AUVs [7], planetary rovers [8], newer domestic robots and even inside the human body. Some of the sensors that is used in SLAM alorithms are DVL sensors, LiDAR, inertial measurement unit (IMU), sonar sensors, cameras etc. The approaches that use cameras as sensor called visual SLAM approaches and are presented in the next section.

2.2 Visual SLAM (vSLAM)

Visual SLAM is the approach of SLAM algorithm with a single camera. Visual SLAM uses images acquired from cameras and other image sensors. Is a specific type of SLAM system that leverages 3D vision to perform location and mapping functions when neither the environment nor the location of the sensor is known. vSLAM extracts keyframes from each input image, after extraction follows the data association, the keyframes that have been detected are stored into a database, the algorithm searches in the database if the current keyframes have been shown again. Subsequently, the state estimation and state update is applied and finally the map is updated. The whole procedure is presented in the image below.

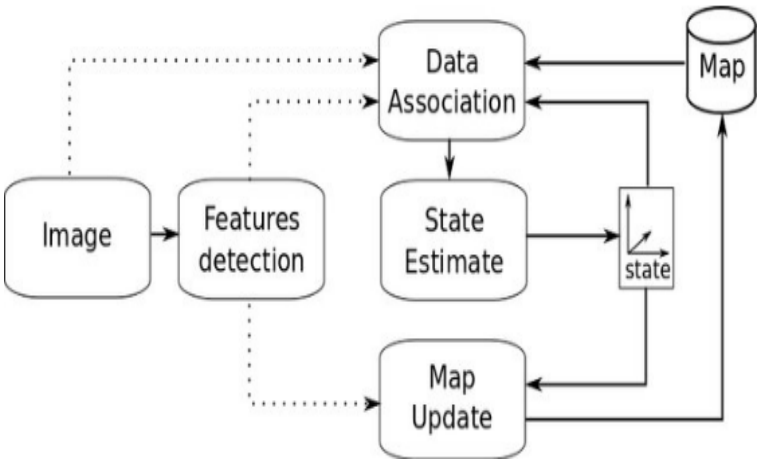

FIGURE 2.1: SLAM algorithm

The need to create navigation systems has a leading role in the development of Visual SLAM(vSLAM) algorithms. Visual SLAM refers to the complex process of SLAM but using only visual inputs from a camera. The problem of Visual SLAM was originally proposed from Andrew J. Davison and David

W. Murray in 1998 [9]. SLAM problem is widely known on autonomous navigation. vSLAM algorithms have been applied in various fields such as self-driving cars [10], unmanned aerial vehicles (UAVs) [11], AUVs [12], planetary rovers [13], newer domestic robots and even inside the human body. There are many vSLAM algorithms and the categorized into 3 general categories; feature-based, direct and RGB-D approaches.

- **Feature-based** methods are split in two categories filter-based and Bundle Adjustment-based
  - **filter-based** method according to the literature [14] the size of the environment is proportional to the computational complexity.
  - **BA-based** method which can handle more complex environments
- **Direct** based methods directly use an input image without using any feature detector or descriptor. These methods mostly used on dense environments and they can run in real-time on CPUs as mentioned at [14].
- **RGB-D** methods use as input an RGB-D image or video from an RGB-D camera and can extract useful informations from the depth value.

2.3 Underwater SLAM

SLAM has various applications on many fields, as shown above. One of the main applications of the algorithm is underwater. SLAM in underwater environments is a complex and costly area of research due to the need for integrating different specialized sensors to obtain a reliable estimate. Also there are many obstacles in underwater environments, like the turbidity of water, the currents, the reduced visibility, are key obstacles to implementing the algorithm.

A first approach to implement underwater SLAM has been developed at the University of Sydney in 2000 [15]. In this research they apply the SLAM algorithm to estimate the motion of a submersible vehicle. The information for the environment had been taken from a sonar. A first approach on vSLAM on underwater environment had been at MIT in 2004 [16].

SLAM in underwater environments is very useful in locating corals, trashes, oil slicks etc. Also is very useful in locating and exploring archaeological sites, bottoms and surfaces of marines. Nevertheless, underwater SLAM

steel remains an unsolved problem because of the water restrictions such as the water currents, the turbulence of the water, and lighting conditions (sunny or cloudy days). If it is a sunny day and the robot is in about 1-3m depth, then SLAM algorithms cannot be effective because of the reflection of light on the water; on the other hand, if it is a cloudy day then there is no reflection so the algorithm is effective. If the robot is in a big depth then artificial lighting is necessary. An approach that trying to solve the problem of water currents presents at University of Rio Grande in 2016 [17].

The approach presented above is based on another work which presents a new underwater SLAM system called DolphinSLAM [18]. DolphinSLAM algorithm is based on the UnderWater Simulation, which is a simulation that represents an underwater environment. It has its robot, the Girona500, where it consists of a sonar, a camera, an IMU and a DVL.

Another work that tries to solve the underwater problem is presented in this paper [19]. The algorithm in this approach is based on the ORB-SLAM algorithm [20]. ORB-SLAM is an approach based on Oriented FAST and Rotated BRIEF (ORB) keypoint detector which are used to match features in consecutive images. In contrast with SURF and SIFT detectors, FAST corner detector is as it names said fastest, it is computationally efficient and it's suitable for real time applications because of this high-speed performance. The algorithm has been tested in many different cases on mobile robots as well as on underwater cases. In the above approach datasets collected from a ROV and examined the behavior of the algorithm in different cases e.g. at different depths, different lighting conditions etc.

2.3.1 ORB-SLAM

ORB-SLAM is a keyframe and feature-based Monocular SLAM. It operates in real-time in large environments, being able to close loops and perform camera relocalisation from very different viewpoints. The Oriended FAST and Rotated BRIEF (ORB) keypoint detector which are used to match features in consecutive images. The ORB detector is based on FAST feature detector [21] and BRIEF descriptor [22]

The algorithm is divided into seven separate modules : feature extraction, data association, initialization, tracking, relocalization, local mapping and loop closure as presented in the image above.

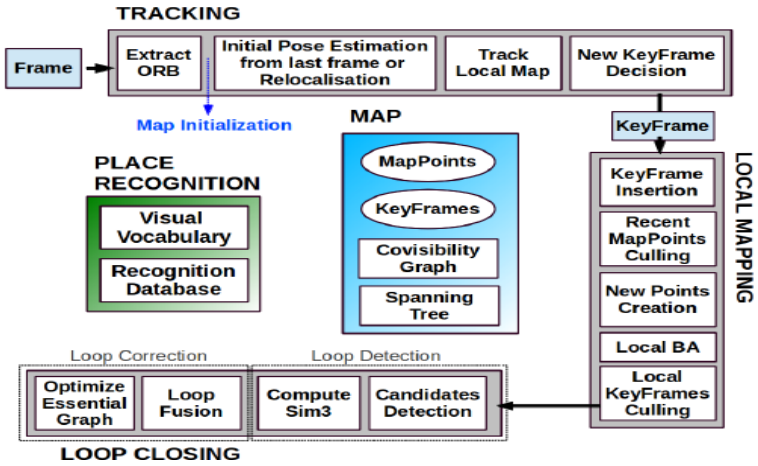

FIGURE 2.2: ORB-SLAM architecture [20]

The ORB features that are used in the algorithm offers real-time implementation on large environments, real-time loop closing and real-time camera relocalization when the algorithm is lost in the environment.

There are three main threads on the algorithm that run as shown in the image above. The tracking, the local mapping and the loop closing frame.

The tracking thread, is the frame that is responsible for tracking and insert new keyframes. The local mapping thread performs local BA on the new keyframes to achieve optimal reconstruction in the surroundings of the camera pose. The loop closing searches if the new keyframe closes a loop. Covisibility Graph and Essestial Graph are storing information about the covisibility between keyframes. When a new keyframe is inserted, it is included in the tree linked to the keyframe which shares most point observations. Finally a Bag of Words Place Recognition is applied. The system performs loop detection and relocalization based on DBoW2[23]. The algorithm searches on a visual vocabulary to closes the loop.

2.3.2 ORB-SLAM2

ORB-SLAM2[24] is an approach based on ORB-SLAM algorithm but with the difference that it has been added a thread after the final thread of ORB-SLAM, the full BA thread and it processes an input pre-processing.

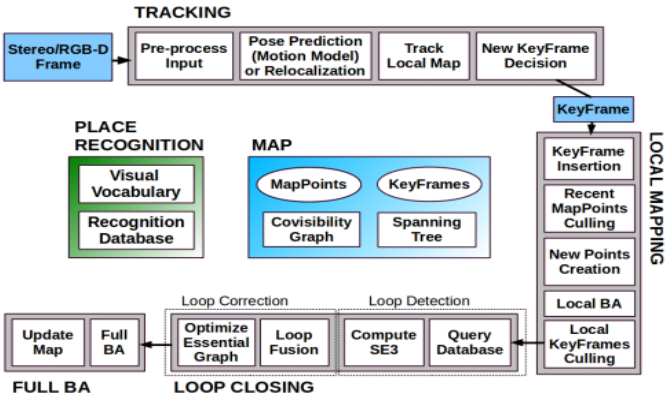

FIGURE 2.3: ORB-SLAM2 architecture [24]

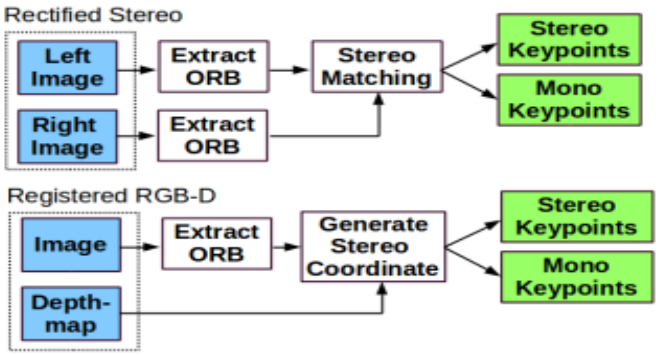

FIGURE 2.4: Input pre-processing[24]

ORB-SLAM2 is the first system for monocular, stereo and RGB-D cameras. This system is using close and far points and monocular obsrvations that makes the system more accurate. The system is the same with the one mentioned before but with the difference that the final thread launched a fourth thread that performs full BA adjustment, to compute the optimal structure and motion solution. Also the system pre-processes the input, extract features and then discards the input so the rest of the system operates with the features and be independent of the kind of image input.

- **Stereo keypoints** are defined by three cordinates $x_s = (u_L, v_L, u_R)$ with (u_L, v_L) being the coordinates on the left image and u_R being the coordinate on the right image. The system extracts ORB in both images and for every left ORB searches a match in the rigtht image. And so the keypoint is generated.

- **RGB-D** for each feature with coordinates (u_L, v_L), the depth is transformed into the right coordinate as proposed in this paper [25].
- **Monocular keypoints** have two coorndinates $x_m = (u_L, v_L)$

A stereo keypoint is classified as close if its associated depth is less than 4o times the stereo/RGB-D baseline. otherwise as far [26].

Bundle Adjustment (BA) is the problem of simultaneously refining the 3D coordinates describing the scene geometry, the parameters of the relative motion, and the optical characteristics of the camera employed to acquire the images, according to an optimality criterion involving the corresponding image projections of all points. The ORB-SLAM2 system performs three times BA. Performs BA to optimize the camera pose in the tracking thread (motion-only BA), to optimize a local window of keyframes and points in the local mapping thread (local BA), and after a loop closure to optimize all keyframes and points (full BA).

- **Motion-only BA** optimizes the camera orientation $R \in SO(3)$ and position $t \in \mathbb{R}^3$, minimizing the reprojection error between matched 3D points $X^i \in \mathbb{R}^3$ in world coordinates and keypoints $x(.)^i$, either monocular $x_m^i \in \mathbb{R}^2$ or stereo $x_s^i \in \mathbb{R}^3$, with $i \in X$ the set of all matches:

$$\{R, t\} = argmin_{R,t} \sum_{i \in \chi} \rho(\left\| x_{(.)}^i - \pi_{(.)}(RX^i + t) \right\|_{\Sigma}^2)$$

  where ρ is the robust Huber cost function [27] and Σ is the covariance matrix associated to the scale of the keypoint. Position t is proportional to the image size and keypoints x are smaller than the image size because there are some points of the image. $\pi(.)$ are the projection functions monocular π_m and rectified stereo π_s,.

$$\pi_m(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}) = \begin{bmatrix} f_x \frac{X}{Z} + c_x \\ f_y \frac{Y}{Z} + c_y \end{bmatrix}$$

$$\pi_s(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}) = \begin{bmatrix} f_x \frac{X}{Z} + c_x \\ f_y \frac{Y}{Z} + c_y \\ f_x \frac{X-b}{Z} + c_x \end{bmatrix}$$

- **Local BA** optimizes a set of covisible keyframes K_L and all points seen in those keyframes P_L. Keyframes are proportional to the size of the

dataset viz the number of the images to be processed. All other keyframes K_F , not in K_L, observing points in P_L contribute to the cost function but remain fixed in the optimization. Defining X_k as the set of matches between points in P_L and keypoints in a keyframe k, the optimization problem is the following:

$$\{X^i, R_l, t_l | i \in P_L, l \in K_L\} = argmin_{X^i, R_l, t_l} \sum_{k \in K_L \cup K_F} \sum_{j \in X_k} \rho(E_{kj})$$

$$E_{kj} = \left\| x^j_{(.)} - \pi_{(.)}(R_k X^j + t_k) \right\|^2_\Sigma$$

Where X^i is the keypoint, R_l is the camera orientation, t_l is the position and ρ is the robust huber function, all of the above were mentioned in the previous paragraph.

- **Full BA** is the specific case of local BA, where all keyframes and points in the map are optimized, except the origin keyframe that is fixed to eliminate the gauge freedom.

The base difference with the ORB-SLAM is the full BA that system performs at the end. With full BA the scale drift that occurs with monocular SLAM no longer exists because the scale/depth information makes scale observable. More information about the functions can fount at [24]

2.4 ORB

ORB as it mentioned before cames from FAST feature detection and BRIEF feature descriptor[28]. ORB is a fast and efficient algorithm and is rotation invariant and resistant to noise.

2.4.1 Features from Accelerated Segment Test (FAST)

FAST features are detecting with the parameter of intensity threshold between the center pixel and those in a circular ring about the center. The cicrcular radius is defined from the user. After detecting the points Harris corner measurement is applied to order the FAST keypoints and select the top N points. A scale pyramid of the image is applied in order to produce FAST features at each level of the pyramid.

2.4.2 oFAST: FAST Keypoint Orientation

FAST does not produce a measure of cornerness, in order to solve the large responses along edges a measure of corner orientation is computed via intensity centroid [29]. The intensity centroid assumes that a corner's intensity is offset from its center, and this vector may be used to impute an orientation.

The moment of a patch is defined as:

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

where I(x, y) is the intensity of the pixel at the relative position x, y within the patch. Patch is a piece of the image so it is proportional to the image size and to how big would be the piece for process e.g. 30x30 or 10x10 etc. The centroid is defined as:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$$

Then the vector with the angle form the center to the centroid is calculated as follow:

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

Where atan2 is the quadrant-aware version of arctan.

2.4.3 BRIEF

The BRIEF descriptor is a bit string description of an image patch (**p**) constructed from a set of binary intensity tests(τ). A binary test is calculated as follow:

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

The feature is defined as a vector of n binary tests:

$$f_n(p) := \sum_{1 \leq i \leq n} 2^{i-1} \tau(p; x_i, y_i)$$

It is important to smooth the image before performing this tests, for example with a Gaussian blur filter. The vector length usually is n = 256.

2.4.4 rBRIEF: Steered BRIEF

It is important the BRIEF to be invariant to in-plane rotation. An efficient way to do this is to steer BRIEF according to the orientation of keypoints. So, for any feature set of n binary tests at location (x_i, y_i), define the 2 × n matrix

$$S = \begin{pmatrix} x_1, ..., x_n \\ y_1, ..., y_n \end{pmatrix}$$

Using the patch orientation θ and the corresponding rotation matrix R_θ, we construct a "steered" version S_θ of S:

$$S_\theta = R_\theta S$$

For more information about the ORB algorithm see [28]

2.5 Theoretical knowledge sources

The aforementioned theoretical background was mostly obtained from the papers attached above and some other sources from sites as Wikepedia.

Chapter 3

Related Work

In order to achieve real-time implementation and energy consumption of image processing on SLAM we can use FPGAs. FPGAs can easily accelerate processes. This advantage make the FPGAs attractive to accelerate processes like image processing, feature extraction etc. These processes can be easily parallelized spatial and temporal. Due to this parallelism FPGAs is very useful and in the SLAM algorithms. The first approaches implementing the algorithm with FPGAs started around 2012. Some of the main approaches presented on the following literature, [30], [31], [32], SLAM algorithms apply to 3 different categories of environments which are Sparse, Semi-Dense and Dense depending on the needs of the environment there have been created a lot of different approaches of SLAM algorithms. Underwater environment are either Semi-Dense either Dense due to the fishes, the corals, sometimes the trashes and even the boats. Because of these dense environments it's necessary to accelerate the feature extraction and the image processing in order to take a real time implementation and more reference points. A first approach on underwater SLAM using FPGAs has been from Computer Vision Group U.P.M., Madrid, Spain in 2009 [33]. A lot of work have been done on underwater SLAM with FPGAs that achieves acceleration on feature extraction and less energy requirements than the respective software applications. Some of them presented below [34], [35].

3.1 DolphinSLAM

DolphinSLAM algorithm [18] is one of the first approaches for underwater SLAM. DolphinSLAM is an extention of RatSLAM for 2D ground robots. This approach is 3D based SLAM that uses a Continuous Attractor Neural

Network (CANN) to localize and manage low resolution images and imaging the sonar data.

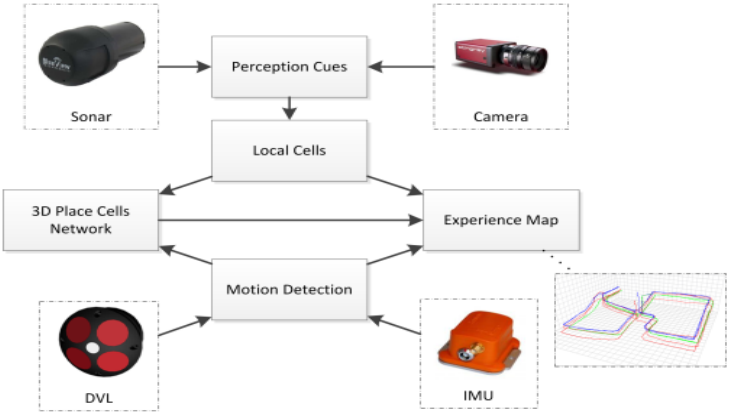

FIGURE 3.1: DolphinSLAM architecture[18]

As shown in the image above, the algorithm is devided into the following parts: Perception Cues, Local Cells, 3D Place Cells Network, Experience Map and Motion Detection.

- **Perception Cues** This module receives information from a Sonar sensor and from a camera and applies Bag of Words(BoW) algorithm for feature detection and description.
- **Local Cells** This modules represents what the robot is perceiving. It applies FabMap algorithm that estimates if the perception was seen for the first time or if came from a place already visited
- **3D Place Cells Network** This module represents a CANN in which each neuron is responsible for mapping a specific area of the environment. Furthermore, this module estimates the robot's position at the same time, guided by external inputs. The module is divided into four steps, Excitation, Path Integration, External Connection Learning, Input activation and Activity Normalization.
- **Experience Map** This module is responsible for creating of the environment map. Every time a new place is visited a new node is created. In each node of the experience graph store the Local View Cell (l_i), the active Place Cell (r'_x, y', z') and the robot position (p_j). There are two steps for this module, the experience activation step which is sets a threshold and if the activity is greater than this threshold then the activity is

activated. And the Experience creation step which if there is no active experience, new one is created.

- **Motion Detection** This module receives information from a Doppler Velocity Log (DVL) and from an Inertial Measurement Unit (IMU). These sensors give information about the velocity (DVL) and for robot orientation (IMU). (The output of the module is the relative motion between two consecutive poses, synchronized with the camera motion between two frames)

This approach has been tested on two ways. The first way was the Under Water Simulator [36], which is a simulator that represents an underwater environment. A scenario was created that it consists from pipes and offshore oil extraction structures. The robot is a simulated Girona 500 [37] with a camera, a sonar sensor, a DVL and an IMU. The second scenario was the ARACATI Dataset which consist from many elements such as boats and stakes which serve as landmarks inputs for the system. Both of the sets gives satisfactory results.

We tried to download and run the DolhinSLAM algorithm on the UWSim environment but due to problems on the versions of some packages, that has been updated and we cannot find the right versions it was impossible to run the algorithm, also there was some datasets and BoW dictionaries that there wasn't inside the github files so it was impossible to run the algorithm. After many tries to find and setup the necessary packages, it was impossible to run the algorithm, so we are going to implement a different version of SLAM that is also used in underwater environments the last years, the ORB-SLAM.

3.2 SLAM with FPGA

According to the type of the environment the SLAM algorithm are dived into the follow categories: sparse, semi-dense and dense. Semi-dense and dense environments due to their nature have large computational demands and their is need to accelerate them with the use of an FPGA in order to achieve real -time implementation. FPGAs can achieve real time implementation with low power consumption so they used to solve SLAM dense problems. Many researches have been at the years [38], [39], [40]. As shown at this researches can increases the throughput of the whole application by a factor of 2.

A High-Performance System-on-Chip Architecture for Direct Tracking for SLAM proposed from Konstantinos Boikos and Christos-Savvas Bouganis [41]. They try to accelerate the Tracking part of the algorithm they use a high bandwidth streaming architecture for a high-performance solution. They achieve a tracking rate of more than 22 $\frac{frames}{second}$ with an embedded power budget and achieves a 5× improvement over previous work on FPGA SoCs.

Another implementation, is that of SMG-SLAM algorithm from Grigorios Mingas and his team they design a system that is up to 14.83 times faster compared to the software algorithm without significant loss in accuracy [32].

3.3 ORB-SLAM with FPGA

The most consuming part of the ORB-SLAM algorithm is the ORB extraction. There are many researches the recent years, that have trying to accelerate the ORB-SLAM with an FPGA. Weikang Fang et. al [42], trying to accelerate the ORB feature extraction part in order to achieve real-time implementation with an extended battery life. An overview of the architecture and the results are presented below.

ORB architecture results				
	Clock Freq. (GHz)	Latency (ms)	Throughput (FPS)	Energy (mJ/frame)
Weikang Fang Proposed Design	0.203	14.8	67	68
ARM Krait	2.26	30	33	75
Inten Core i5	2.9	25	40	400
Improvement vs ARM	-	51%	103%	9%
Improvement vs Intel	-	41%	68%	83%

TABLE 3.1: ORB architecture results [42]

Another approach that trying to accelerate the feature extraction and matching is this from Runze Liu et al [43], compared with Intel i7 and ARM Cortex-A9 CPUs on TUM dataset, our FPGA realization achieves up to 3× and 31× frame rate improvement, as well as up to 71× and 25× energy efficiency improvement, respectively. An overwie of the architecture is presented below.

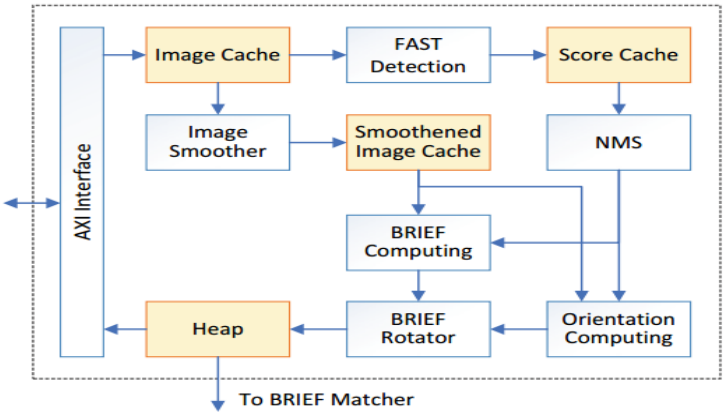

FIGURE 3.2: ORB Hardware architecture 2 [43]

3.4 Datasets

In order to test the right behavior of an algorithm there is need of a dataset. The dataset for underwater environments can be either a dataset from a simulation e.g. UWSim [36] an open source dataset collection from UWSim provided by Amanda Duarte et al. [44] either on a real underwater environment with a ROV. It was difficult to find an underwater dataset for our needs.

The datasets presented below are recorded in three different real underwater environments with an ROV. The environments are a harbor at a depth of a few meters, a first archaeological site at a depth of 270 meters and a second site at a depth of 380 meters [45].

The main dataset used on this approach was the following approach.

ORB-SLAM as it mentioned above provides real-time implementation. Therefore, this approach of SLAM, have been tested for underwater robots was from F.Hidalgo and his team [46]. This approach has 9 different cases at 9 different places all close to the Marina in Fremantle at Western Australia. These cases are:

- **Boat** This dataset has been taken a sunny day at different depths, between 4 and 7m depending on the spot. This case has 5 sub-cases at 5 different spots.
- **Marina in Fremantle** This dataset has 3 different sub-cases each one on different lighting conditions and each of sub-case has also sub-cases on different depths etc. The three different sub-cases were on a sunny day

at 2-3 meters, on a cloudy day also at 2-3 meters depth and a cloudy night with the help of a light at about 0.5 meters.

- **Omeo wreck** This dataset has been taken a cloudy day at 3-4 meters depth.
- **Pool** This dataset has been taken at the pool of the lab a cloudy day at 2 meters depth.
- **River at Bicton** This dataset has two differnt sub-cases. One on a sunny day at 2-3 meters depth with different lighting conditions and one on a cloudy day at 0.5-1 meter.
- **River at North Fremantle** This dataset was on sunny day at about 50 cm .
- **River at UWA** This dataset was on a cloudy day at different depths and with different lighting conditions.
- **Woodman point** This dataset was on a cloudy day at 1.5 meter depth.

According to the above experiments the conclusion of the authors was that in different lighting conditions, at different depths and according to the environment around the ROV the result of the algorithm is different. For example if we have a cloudy day at a small depth around 1-3 meters and with many rocks or algae then the keypoints that will be detected there will be much more than those on a sunny day at 1-3 meteres with same enviroment conditions. Also if the environment has no algae or rocks it is difficult to find keypoints. More about the parameters of the algorithm will be referred above.

3.5 Thesis Approach

Underwater SLAM is a challenging topic for underwater vehicles in long-term operation due to the limitations of subsea localization sensors and perception sensors for mapping. The most underwater environments are dense and demands low power consumption. So it's big need to achieve real-time implementation on low-power platforms on those environments. A way to achieve this is to accelerate algorithms with FPGA. Much of SLAM cases have been applied on underwater environments as mentioned and much of SLAM cases have been accelerated with FPGA. However the ORB-SLAM2 case have not been accelerated yet with FPGA on underwater environements.

The purpose of this thesis is the acceleration the ORB feature extraction part of ORB2-SLAM algorithm. The acceleration of the ORB will not take place at the time, in the underwater case we are studying is very important the energy efficiency. On the ROVs the energy efficiency is important. We want to explore large areas, so as much battery our ROV have larger areas will be explored. The part of the algorithm which will accelerate is the ORB Extraction. After CPU analysis the ORB Extraction is the most time consuming class of the algorithm. We aim to accelerate the algorithm on FPGA and not on GPU beacause at the onset of this thesis GPUs were too power-hungry (the Jetson series was not available yet), as well as the availability of the Xilinx ZCU-102 platform in the Microprocessor and Hardware Lab of the Technical University of Crete. The ZCU-102 was deemed to be appropriate, and indeed it turned out to be so.

Chapter 4

System Modeling

The main algorithm is written in C++. This version of the algorithm can be compiled without the ROS operating system which was necessary for all the other versions of SLAM algorithms. We use the monocular case of the algorithm.The inputs are a visual vocabulary, a .yaml file and the path to the sequence folder.

The vocabulary consists of visual words which are a discretization of the descriptor space. This vocabulary is created with the ORB descriptors extracted from a large set of images. If the images are general enough, the same vocabulary can be used for different environments. The vocabulary used for this design was the default Vocabulary of ORB-SLAM2.

The .yaml file is a file which contains some default parameters of the algorithm such as the number of features per image, the scale factor of the image pyramid, the levels of the image pyramid, the features must be extracted at each cell of the image grid etc.

In order to compute the image keypoints on a frame, corner detection is used. A corner is a point whose local neighborhood stands in two dominant and different edge directions. In other words, a corner can be interpreted as the junction of two edges, where an edge is a sudden change in image brightness. Corners are the important features in the image, and they are generally termed as interest points which are invariant to translation, rotation and illumination. Although corners are only a small percentage of the image, they contain the most important features in restoring image information, and they can be used to minimize the amount of processed data for motion tracking, image stitching, building 2D mosaics, stereo vision, image representation and other related computer vision areas.

The approach is based on the main algorithm of ORB-SLAM2 and the datasets is available online and implements three different underwater environments such as pools, marinas and open sea with varying water conditions such as cloudy or sunny day and in many different depths.

According to the paper above there are two types of underwater environments structured and unstructured. In the first category, the structured environment, are included the marine platforms, ports etc. that is, the environment that cointains a base line in which the ROV can move and had easily recognizable keypoints. The second category, the unstructured, contains the mostly natural environment such as seabed ets. at these environment the keypoints are more difficult to be detected.

There are many constraints that must be taken into account. These constraints are the water turbidity, the lighting conditions the depth, the reflection of the water and the sandy areas . As observed at the paper, the results at the different constraints was different. For example as many as sand the area has as less as the features extracted, on the sunny days because of the reflection of the water the features that extracted at closed on time images where different.

The examples will presented below, are from the run of the ORB-SLAM2 algorithm which we can find in github [47] with the dataset mentioned at chapter 3.4. The algorithm has some dataset that the designers have developed but for our needs we used the dataset mentioned before.

Some examples are given below. In the first example, as it is shown in figure 4.1, is presented the case in which it was a cloudy day, so there is no reflections on the water from the sun and the depth in about 2-3 meters. As is seems, there are some rocks and algea so it is possible to detect many keypoints and start creating a the map, as it shown in figure 4.2.

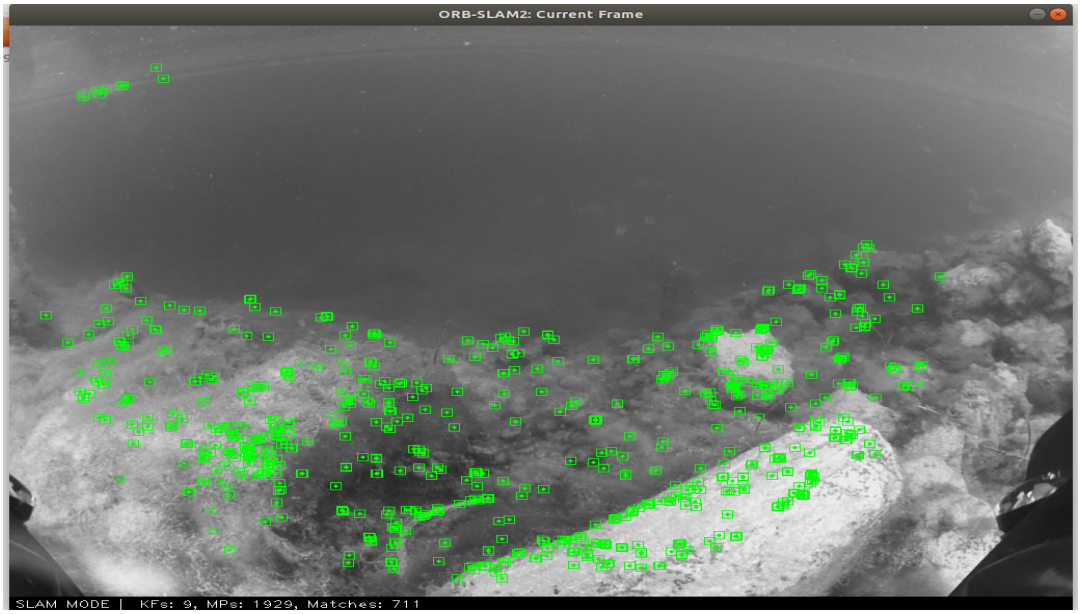

FIGURE 4.1: Surface to 2-3 meters, Cloudy day from the run of ORB-SLAM2 with our dataset

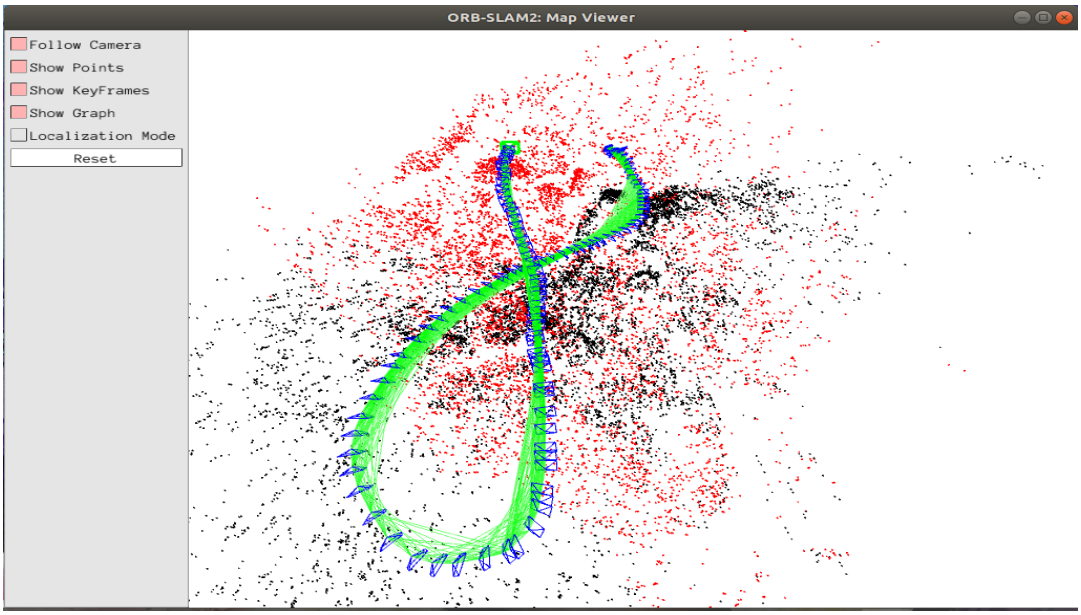

FIGURE 4.2: Map created from the keypoints from the run of ORB-SLAM2 with our dataset

In this figure we can see the route created in the map from the surrounding environment. The ORB-SLAM2 algorithm detect the keypoint as shown in figure 4.1 save the keypoints and creates the route has followed. The red and black points are the keypoints founded and the green line is the route has followed.

Below we can see an another case in which is a sunny day and the depth is about 50 cm.

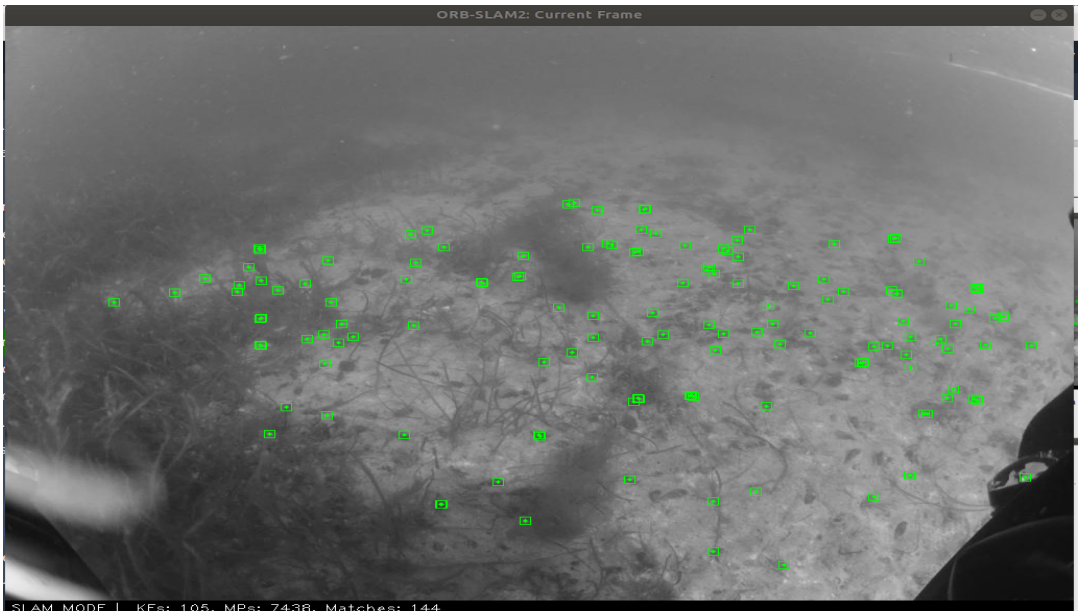

FIGURE 4.3: Surface to 50 cm, Sunny day from the run of ORB-SLAM2 with our dataset

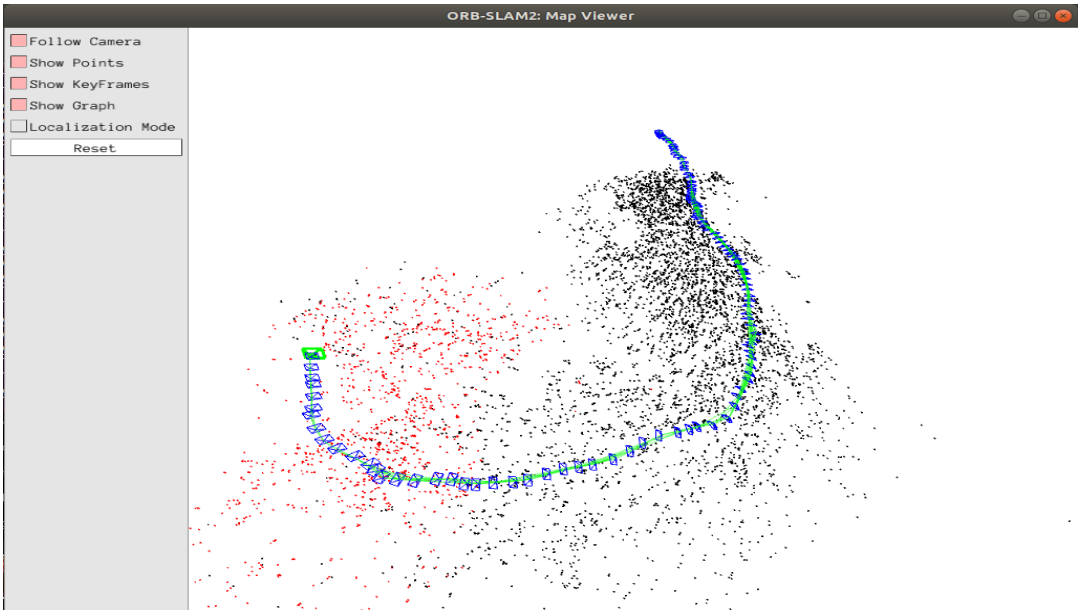

FIGURE 4.4: Map created from the keypoints from the run of ORB-SLAM2 with our dataset

Except from the cases that the algorithm succeed finding keypoints and build a map there are and some cases that the algorithm can not find keypoint due to the underwater restrictions (turbidity of water, lighting conditions etc.).

In the second example, as it shown in figure 4.5 is presented the case in which the depth is about 3-4 meters, but there is only sand, so it is impossible to find keypopints and start creating a map.

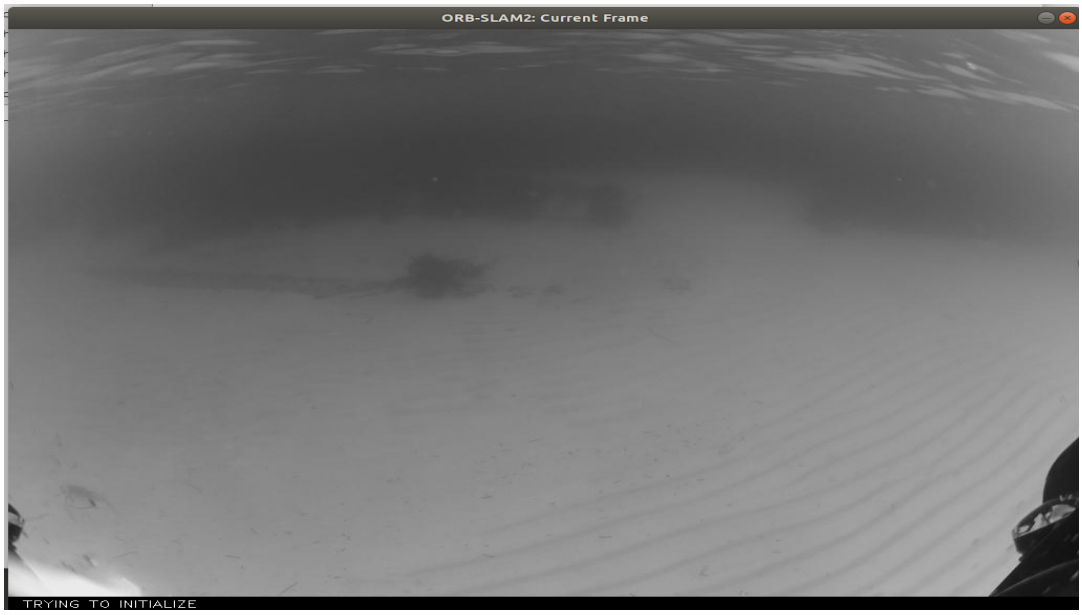

FIGURE 4.5: Surface to 2-3 meters, Sunny day

Some other cases that the algorithm failed, is the case in which the ROV is at the surface and it is a suuny day, in this case due to the reflection it is impossible to detect keypoints. Last but not least, the case in which the ROV is at surface, with artificial lights so there is reflection and the area is without rocks or algea so it is difficult to find keypoints.

In conclusion, these experiments shows that according to the constraints the results are very different. If it is a sunny day and the depth is small the algorithm can't recognize frames as in the case which is cloudy or the depth is bigger. We run many different cases in order to validate the algorithm but here are presented a few of them. The same datasets used for the hardware validation. More details are presented in the chapter 7.

The default parameters defined at .yaml file are the follows:

Target number of features	2000
Scale Factor	1.2
Number of levels	8
Initial FAST threshold	20
Minimum FAST threshold	7

TABLE 4.1: ORB initial parameters

The dataset that this paper provides is the main dataset of our algorithm.

4.1 CPU analysis

To analyse the performance of the algorithm chrono library of C++ is used for measuring the time needed on the extraction part of the algorithm. The *std* :: *chrono* :: *steady_clock* function was used for measuring the time. Before the call of the Extractor function was placed a time stamp and after the call another one. The calls of the extraction function is as many as the frames of the input dataset. So the finally time spent in the function is almost the half time of the whole algorithm. The analysis shows that almost the half time of the algorithm is spent on the ORB extraction part. For example for the Marina if Fremantle dataset, the execution of the ORB part of algorithm was **220** seconds. The ORB feature extraction consumes about *50%* of the time of the whole algorithm.

Afterwords, the execution time of ORB Extractor class was analysed and as shown in the bellow figure the most time consuming part of Extraction is the FAST keypoint detection.

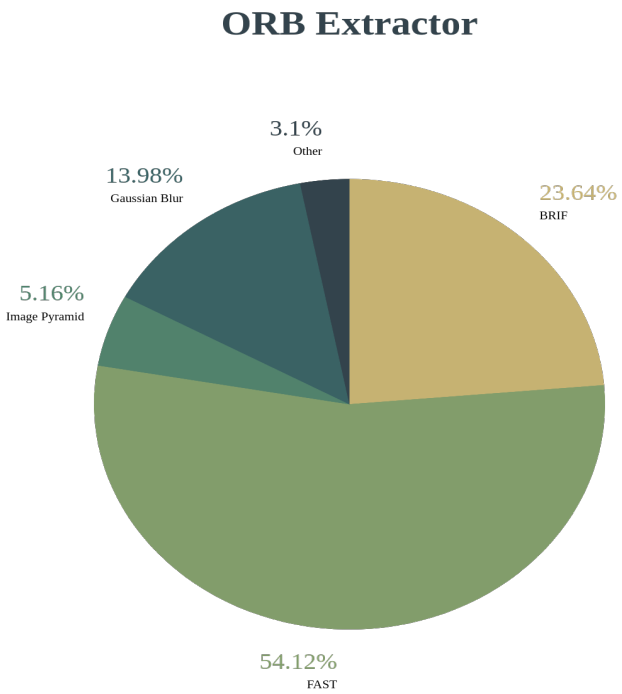

FIGURE 4.6: ORB Extraction execution time

4.2 ORB feature detector

A block diagram of the main parts of the ORB feature detector algorithm is shown below.

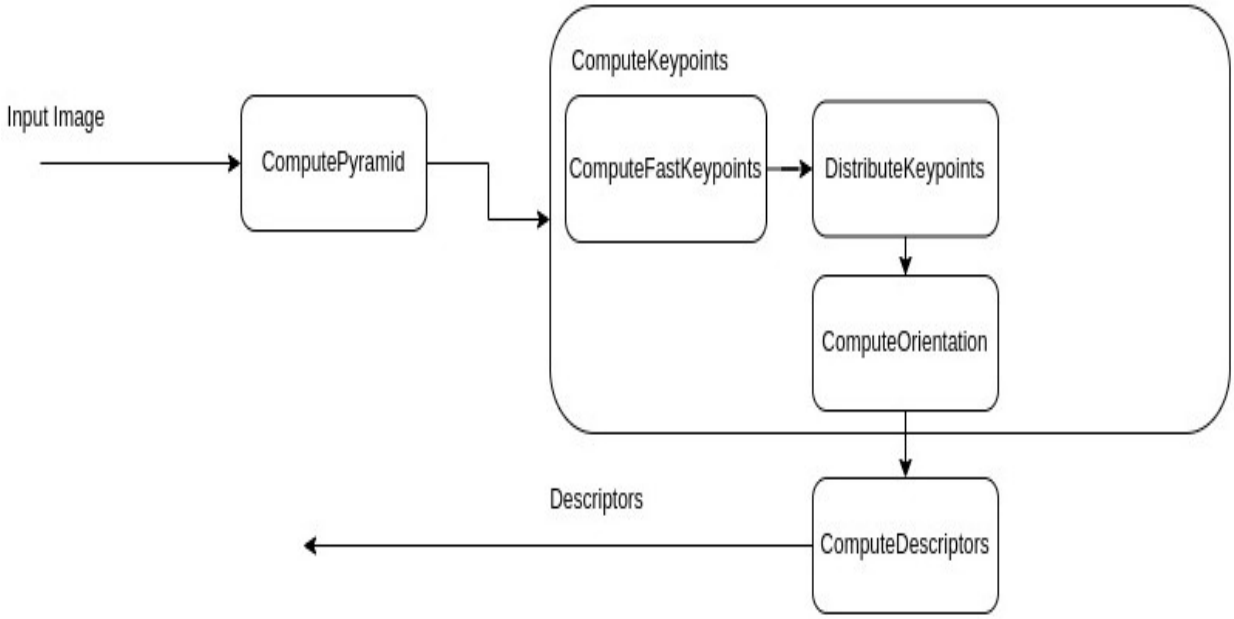

FIGURE 4.7: ORB feature detection main algorithm

These are the main components of the ORB feature detection part of the ORB-SLAM2 algorithm. All functions are executed sequentially. All the parts one by one are described below. The main steps are followed at the ORB extraction part of the algorithm are the following:

1. **ComputePyramid:** An image pyramid is calculated and stored into a predefined table size, this size is defined at the .yaml in which are defined all the necessary setting for the algorithm
2. **ComputeKeypoints:**
   (a) **ComputeFastKeypoints:** Calculate the keypoints by dividing the image into 30x30 patches in each level separately.
   (b) **DistributeKeypoints:** After the calculation of the keypoints in a level a distribution is followed. An OctTree is implementing in order to sort an choose the best keypoints from the response of each keypoint.

(c) **ComputeOrientation:** After the distribution of the keypoints is complete, the orientation of each keypoints is calculating.

3. **ComputeDescriptors:** Computes the ORB descriptors of the extracted keypoints

Firstly, the pyramid of the current frame is computed. Algorithm 1 shows how the pyramid is computed. It is obtained by down-sampling step by step, and stops sampling until a certain termination condition is reached. The application of image pyramid in SLAM is mainly used to solve the problem of scale invariance. Scale invariance is achieved by constructing an image pyramid and detecting corner points on each layer of the pyramid

In order to reduse the size of the image the resize function is used instead of PyrDown. The reason is that the PyrDown function is more computational cost in contrast with Resize also the PyrDown function applies sampling so some information are lost.

Algorithm 1 Compute Pyramid

```
1: procedure ComputePyramid(image)
2:     for Every level of the pyramid do
3:         imagePyr(level)
4:         if level ≠ 0 then
5:             resizeImage
6:             makeBorder
7:         else
8:             originalimage
```

After calculating the pyramid, we search for keypoints at every image in the pyramid. Each image of the pyramid is divided into a 30*30 grid, extract the FAST corner points in each grid, and ensure the uniform distribution of the corner points. Algorithm 2 is present the main functionality of the function.

After the computation of the keypoints of each cell of the grid, they are added to the list of the keypoints of all the images. This part of the algorithm divide the current frame into nodes and from its node keeps the most important feature points which. In order to ensure an homogeneous distribution, we are trying to extract at least five corners per cell. According to the resolution of the image the number of features extracted at each frame is either 1000 if the image resolution is 512 × 384 to 752 × 480 either 2000 if the resolution is higher. The function that implements this is the **DistributeOctTree** function.

Algorithm 2 Compute Keypoints

```
procedure ComputeKeyPointsOctTree
    for Every level of the pyramid do
        Initialization of some parameters
        for Every Row of the image do
            for Every Column of the image do
                FAST(image, keypoints, initthreshold)   ▷ Compute the fast keypoints of the specific patch of the image
                if keypoints is empty then
                    FAST(image, keypoints, minthreshold)
                if keypoints is not empty then
                    for Every keypoint do
                        Add the keypoints to the container vToDistributKeys   ▷ Vector of keypoints
        keypoints ← DistributeOctTree(vToDistributKeys)   ▷ Perform culling
        for Keypoints size do
            Add border to coordinates and scale information
    for Every leve of the pyramid do computeOrientation(image,keypointsAtEachLevel)   ▷ Calculate the direction of each feature point
```

When the necessary keypoints have been calculated, then we compute the orientation, viz the direction of each point. At each level of the image pyramid, the orientation of each keyopoint is calculated.

Algorithm 3 Orientation

```
procedure ComputeOrientation(image, keypoints, umax)
    for each keypoins do
        keypont− > angle = IC_Angle(image, keypoint− > pt, umax)   ▷ Compute the angle of the keypoint
```

Algorithm 4 is calculated according to the subsection 2.4.2

The final step is to compute the ORB descriptor for each keypoint. The selection rule is a static array, according to the angle of the key point, this array called pattern. Converting, and then comparing the size of the converted two pixel values, a total of 256 dimensions, stored in 32 charType matrix. So algorithm 5 implements the descriptor.

A GaussianBlur is applied at each level of the image before the computation of the descriptors.

Algorithm 4 Compute Angle

procedure IC_Angle(image, keypoint->pt, umax)
  for $u = -half\_patch\_size; u <= half\_patch\_size; ++u$ **do**
    $m\_10 += u * center[U]$ ▷ pointer traversal method gradation value of each pixel
  for $V = 1; u <= half_patch_size; ++u$ **do**
    for $u = -d; u <= d; ++u$ **do**
      $val\_plus = center[u + v * step], val\_minus = center[u - v * step]$
      $v\_sum += (val\_plus - val\_minus)$
      $m\_10 += u * (val\_plus + val\_minus)$
    $m\_01 += v * v\_sum$
  return $fastAtan2(m\_01, m\_10)$

Algorithm 5 Compute Descriptor

procedure $computeDescriptor$(image, keypoint, descriptors, pattern)
  for each keypoint **do**
    $computeOrbDescriptor(keypoint[i], image, pattern[0], descriptors.ptr((int)i)$

Algorithm 6 Compute ORB Descriptor

procedure $computeOrbDescriptor$(keypoint, image, pattern, desc)
$defineGET_VALUE(idx)$
  $center[cvRound(pattern[idx].x * b + pattern[idx].y * a) * step + cvRound(pattern[idx].x * a - pattern[idx].y * b)]$
  for $i = 0; i < 32; i++, pattern += 16$ **do**
    $GET_VALUE(0); t1 = GET_VALUE(1);$
    $val = t0 < t1;$
    $GET_VALUE(2); t1 = GET_VALUE(3);$
    $val| = (t0 < t1) << 1;$
    $GET_VALUE(4); t1 = GET_VALUE(5);$
    $val| = (t0 < t1) << 2;$
    $GET_VALUE(6); t1 = GET_VALUE(7);$
    $val| = (t0 < t1) << 3;$
    $GET_VALUE(8); t1 = GET_VALUE(9);$
    $val| = (t0 < t1) << 4;$
    $GET_VALUE(10); t1 = GET_VALUE(11);$
    $val| = (t0 < t1) << 5;$
    $GET_VALUE(12); t1 = GET_VALUE(13);$
    $val| = (t0 < t1) << 6;$
    $GET_VALUE(14); t1 = GET_VALUE(15);$
    $val| = (t0 < t1) << 7;$
    $desc[i] = val$

In order to reduce the execution time and achieve power optimization, a kernel was created for the execution of the part of FAST keypoints extraction.

As the Algorithm 2 shows the triple for loop of this part of the algortithm(1 for every level, 1 for each row of the grid and 1 for each column of the grid), slows down the execution of the program. For example, for the first level of the image, the size of the image is 1024*768, according to the above the image will cut in 30*30 grids so the number of rows and cols that has to be proccessed are 20 and 27 accordingly. So the final times that FAST funcion will be execute is 540 times for the first image. Every 30*30 grid has no dependence between the, so we tried to pipeline those two loops.

Chapter 5

System Architecture

In this chapter we present the main design which was developed in the Thesis. This architecture designed for the Xilinx ZCU 102 Evaluation Kit as a platform. In order to design the architecture, Vitis IDE was used. A detailed description of the tool is presented in 6. The first steps was set up the ORB algorithm in Vitis IDE. The main algorithm can found at github [47]. As mentioned before the most computing expensive part of the algorithm is the ORB Extractor part so this will be the kernel of our design.

5.1 Host Configuration

The Host is the part of the algorithm that runs at the CPU of the FPGA and triggers the kernel. So as the software implementation reads the images from an external folder the same does and the Host Configuration and passes the necessary inputs to the Kernel. These inputs are the corresponding input image as an unsigned int of 8 bits, the rows and the columns of the image in order to calculate the xf::cv::Mat element the corresponding element of Mat in OpenCV. The Kernel gives as output the final vector of the descriptors of the image.

5.2 Kernel Configuration

Our approach is dividing into 4 main parts:

- **Resize module:** Computes the image Pyramid
- **ComputeModule:** The main compute module. Extracts the FAST keypoints at each level of the pyramid.
- **Compute Orientation:** Computes the orientation of a keypoint.

- **Compute Descriptors:** Computes the descriptor of the keypoint.

Each of the 4 for modules will be explained below.

ORB extraction in order to extract keypoints from the images uses FAST function from OpenCV library instead of FAST function in our design xf::cv::fast is the corresponding function from xfOpenCV Library of Vitis, more details are presenting in the next chapter. Some more functions used are Crop, Sobel, Resize, Gaussian Blur, duplicateImage.

Every keypoint is described from some values such as the coordinates of the keypoint, the angle, the class_id, the octave, the response and the size. In software we can access all this element by defining a keypoint with the corresponding class cv::KeyPoint of the OpenCV library. In order to store the necessary values from each keypoint in our kernel a struct was created. This struct keeps the neseccary values so that our kernel implemented. These values are x and y e.g. the coordinates of the keypoint, the angle for the orientation, the octave so that the level from which it was extracted is known, the descriptor of each keypoint and the size

A block diagram of the approach is presented bellow:

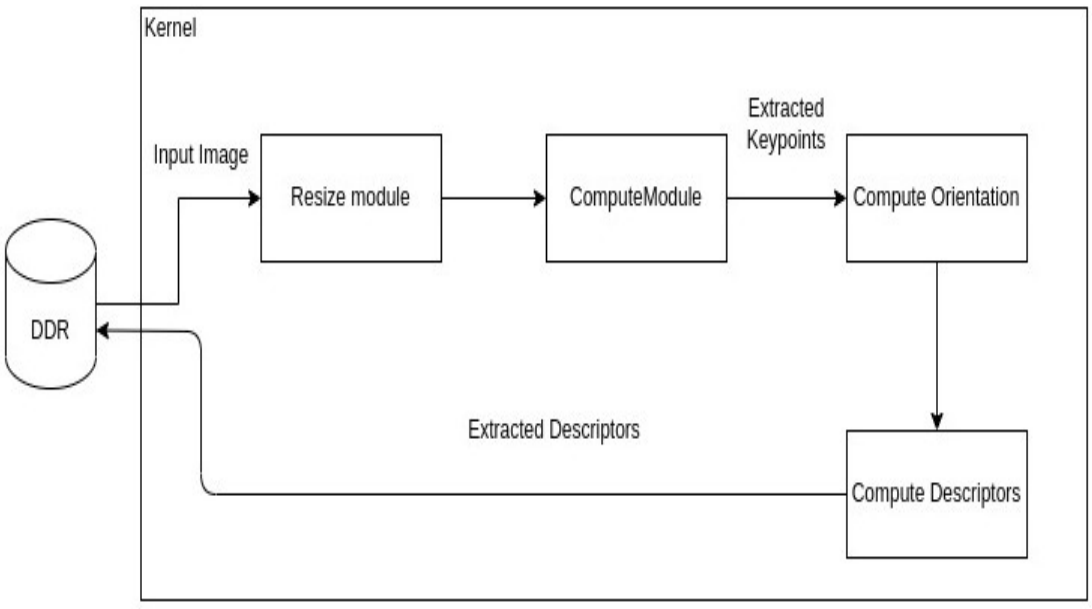

FIGURE 5.1: Diagram of kernel

5.2.1 Resize Module

The image pyramid computes sequentially a pyramid of the main image. That is resizes each time the main image. The eight level resize of the image gives us the advantage of pipelining the resize and compute keypoints, so every time a resized image produced, fed as input to the compute module in order to compute the keypoints, the main function of *xf::opencv* library was used for the pyramid is the resize function.

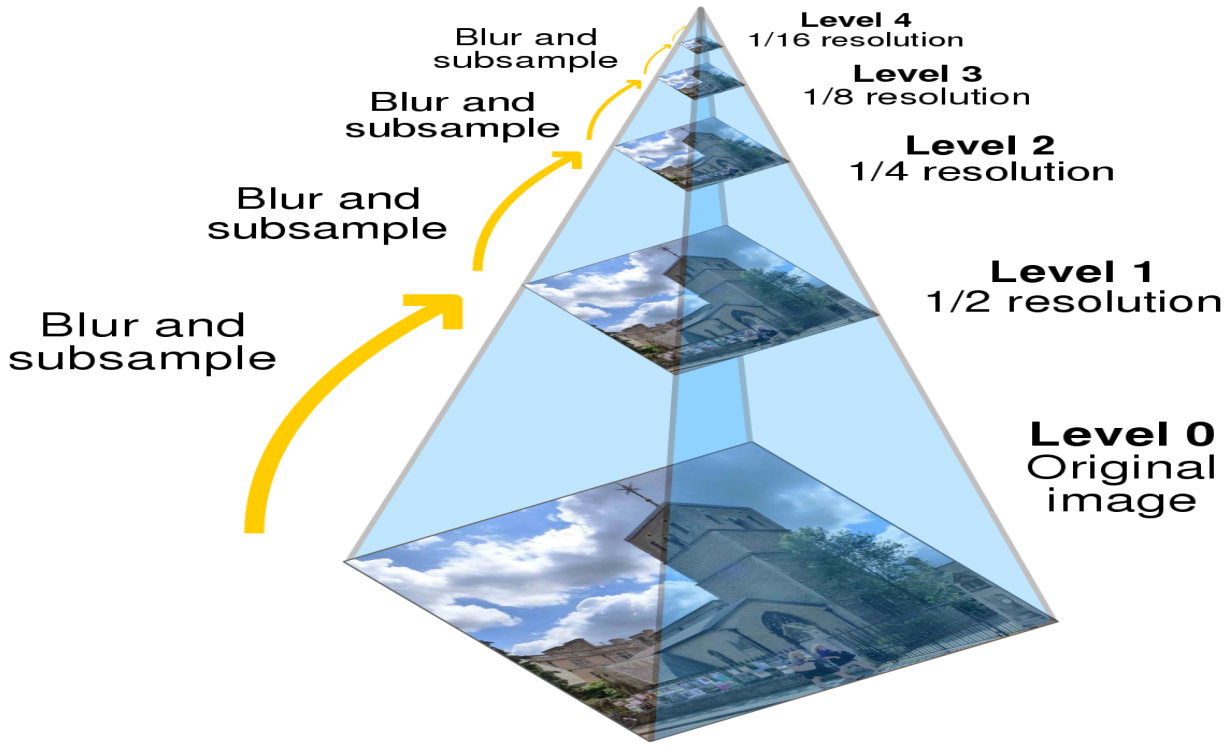

FIGURE 5.2: Image pyramid [48]

5.2.2 Compute Module

This is the main module of the approach. The purpose is to pipeline the computation of the keypoints between the patches this is possible because there is no dependency between the patches. The keypoints in the original algorithm are computed in two phases firstly with an initial threshold usually 20 and if no keypoints found with this threshold then it computes again the keypoints with a minimum threshold usually 7. In our case we duplicate the input image in order to process FAST in parallel for the two thresholds. The first thing the module do is to crop the image in the 30*30 patch in order to achiieve this we use the crop function of *xf::opencv* after cropping the two images we process FAST in parallel for them. When FAST finished we are

starting the extraction of the keypoints when at least one keypoint of the high threshold found, the keypoints of the minimum threshold are discarded. When a keypoint found the orientation and the descriptor of the keypoint is calculating immediately. With this way we can achieve pipelining between the patches and also pipelining between the computation of orientation and descripors. Finally, the descriptors are writing into the output buffer and fed back to the DDR memory.

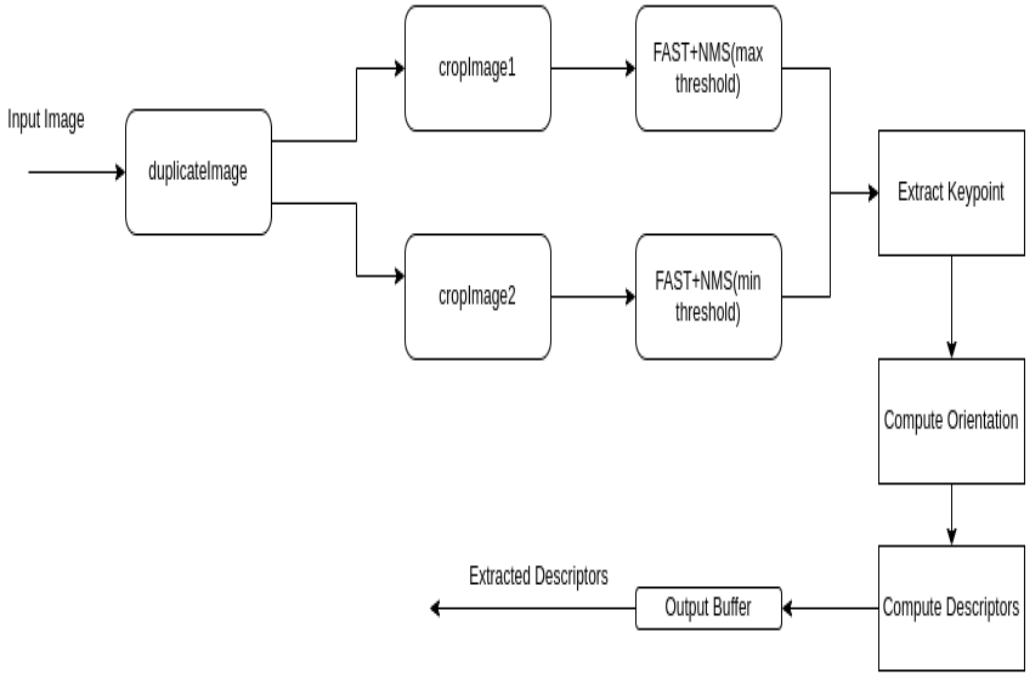

FIGURE 5.3: Compute Module

5.2.3 Compute Orientation

This module takes as input the keypoint extracted in order to compute the orientation with the given algorithm 4. The computation of the moments has been parallelized because of there is no dependence between the values needed for the moment calculation.

5.2.4 Compute Descriptors

This module takes as input the Gaussian Blured image as well as and the orientation of the each keypoint and a 32-bit descriptor is calculated for its keypoint. In order to calculate the orientation a predefined array of patternX

and patternY is used. Those two array are completely partitioned in order to calculate the 32-bit descriptor in parallel.

Chapter 6

FPGA Design

This chapter details the design of the architecture described in chapter 5 as well as the tools used in order to implement the design in the ZCU 102 Evaluaton Kit.

6.1 Tools Used

The FAST keypoint extraction of ORB-SLAM2 algorithm, was implemented and optimized for FPGA platfrom using the Vitis Unified Software Platform [49]. Vitis Unified Softwre Platform is a Software design by Xilinx that compbines all aspects of Xilinx software development into one unified environment. The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on x86 or embedded processors, with OpenCL API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++, OpenCL C, or RTL. The Vitis software platform promotes concurrent development and test of the Hardware and Software elements of an heterogeneous application. In this case both host program and kernel was developed using C++ programming language. Vitis splits the code into two parts. The first part is the host code and the second is the kernel code with a communication channel between them. The host code is the main code, the code that runs into the CPU using API abstractions like OpenCL. The kernel code concerns the code that run into the FPGA.

6.1.1 Vitis Software Platform (Accelerated Flow Application Development)

In the Vitis core development kit, the application consists of a host program and a hardware accelerated kernel with the two being connected through a communication channel between them. The host program, written in C/C++ and using API abstractions like OpenCL, runs on a host processor (such as an x86 server or an Arm processor for embedded platforms), while hardware accelerated kernels run within the programmable logic (PL) region of a Xilinx device. As mentioned in the present case both host and kernel was developed with C++.

The API calls, managed by XRT. Xilinx Runtime Library [50] allows the developers continue using familiar languages like C/C++ and Python and also facilitates communication between the application code and the accelerated-kernels.

Communication between the host and the kernel, including control and data transfers, occurs across the PCIe® bus or an AXI bus for embedded platforms. While control information is transferred between specific memory locations in the hardware, global memory is used to transfer data between the host program and the kernels. Global memory is accessible by both the host processor and hardware accelerators, while host memory is only accessible by the host application.

Vitis Software Platform provides some build targets that makes easier for the developer to debug and run the code. There are three different build targets:

- **Software emulation** Both host and kernel are compiled and run on the host processor. This allows to identify easily syntax errors and perform debugging of he kernel code running together with application, and verifying the behavior of the system.
- **Hardware emulation** The kernel code is compiled into a hardware model (RTL), which is run in a dedicated simulator. This render easily the testing of the kernel code because is running into a simulation of the FPGA
- **Hardware** The kernel code is compiled into a hardware model (RTL) and then implemented on the FPGA, resulting in a binary that will run on the actual FPGA

Vitis Vision Libraries

The main purpose of Vitis Vision Librarie is to achieve real-time performance and flexibility to manage a range of frame resolutions and adaptable throughput requirements, while being power-efficient. Vitis Vision Labrarie provides a familiar interface like the OpenCV the xfOpencv librarie which implements the most functions of OpenCV in terms of FPGA and can achieve high-performance with low power. In the present case the functions used was FAST, resize, crop and GaussianBlur.

Xilinx FPGAs can provide an important acceleration percentage over the traditional CPU/GPU performance. They provide custom architectures capable of implementing any function that can run on a processor, resulting in better performance at lower power dissipation. These acceleration can provide an ideal balance between performance and power.

The main advantage of FPGAs and programmabled devices in constrant with CPUs and GPUs is that FPGAs are fully customizable architectures. This gives to the developer the advantage of creating computing units that are optimized for application needs. In constrant, CPUs and GPUs have predefined architecture, fixed cores and instruction set.

Think of a CPU as a group of workshops, with each one employing a very skilled worker. These workers have access to general purpose tools that let them build almost anything. Each worker crafts one item at a time, successively using different tools to turn raw material into finished goods. This sequential transformation process can require many steps, depending on the nature of the task. The workshops are independent, and the workers can all be doing different tasks without distractions or coordination problems.

A GPU also has workshops and workers, but it has considerably more of them, and the workers are much more specialized. They have access to only specific tools and can do fewer things, but they do them very efficiently. GPU workers function best when they do the same few tasks repeatedly, and when all of them are doing the same thing at the same time. After all, with so many different workers, it is more efficient to give them all the same orders.

Programmable devices take this workshop analogy into the industrial age. If CPUs and GPUs are groups of individual workers taking sequential steps to transform inputs into outputs, programmable devices are factories with assembly lines and conveyer belts. Raw materials are progressively transformed into finished goods by groups of workers dispatched along assembly

lines. Each worker performs the same task repeatedly and the partially finished product is transferred from worker to worker on the conveyer belt. This results in a much higher production throughput.

In the present case the kernel code is the Fast extraction of ORB part of the algorithm. The host code takes the arguments that will be passed to the kernel and put these arguments into a buffer. That is, it moves the arguments to the global memory so they can be visible from the kernel. Then is launch the kernel, the kernel is running and writes the result back to the global memory in order to be visible from the host code.

ORB-SLAM2 algorithm in order to process the input images uses the OpenCV library [1]. So it is need to implement this library into terms of FPGA. The Xilinx provides us this library called xfopencv [2]. Is a set of 60+ kernels optimized for Xilinx FPGAs and SoCs, based on the OpenCV computer vision library. All the library functions are following the same format.

- The functions are designed as templates and the images have to be provided as xf::Mat.
- All functions are defined in the xf namespace.
- Some of the major template arguments are:
  - Maximum size of the image to be processed
  - Datatype defining the properties of each pixel
  - Number of pixels to be processed per clock cycle
  - Other compile-time arguments relevent to the functionality.

So, the main purpose is to replace the Opencv functions of FAST part and the GaussianBlur of ORB feature detection class with these from the XfOpencv function, and measure the response of these functions.

6.1.2 Vitis High-Level Synthesis (HLS)

Vitis HLS [51] is a high-level synthesis tool that allows C, C++, and OpenCL™ functions to become hardwired onto the device logic fabric and RAM/DSP blocks. Vitis HLS implements hardware kernels in the Vitis application acceleration development flow and uses C/C++ code for developing RTL IP for Xilinx device designs in the Vivado Design Suite.

Synthesis Report

A synthesis report is created whenever HLS successfully synthesizes an IP Block, showing various performance and resource utilization metrics. Using the synthesis report, the designer can easily find and target the bottleneck to further optimize their design in terms of both performance and resources

- **Latency** The number of clock cycles required for a complete run of module or loop.
- **Iteration Latency** The number of clock cycles required for running a single iteration of a module or loop.
- **Iteration/Initiation Interval(II)** The number of clock cycles required before a module can accept new input or a loop can initiate a new iteration
- **Pipelined** Whether a module or loop is implemented using a pipelined architecture
- **Area** The number of hardware resources a module requires for its implementation into the target FPGA. The hardware resource types are Block RAM (BRAM) and Ultra RAM (URAM), Digital Signal Processing (DSP) units, Flip Flops (FF), and Lookup Tables (LUT). A table is also given on the detailed report, showing the number of hardware resources required for every hardware component type, which include DSPs, Expressions, First-In-First-Out (FIFO) queues, Instances, Memories, Multiplexers, and Registers.

Optimization Directives

Vitis IDE as Vivado HLS on the kernel provides a set of optimization directives for optimizing the latency, throughput and resource utilization of the exported IP block. The directives can be added directly to the kernel code. Below are presented some of the basic optimization directives.

- **Interface** The top-level function's arguments have to be mapped to RTL ports to configure the IP block's functionality. The interface directive specifies each argument's port type.
- **Stream** By default, the top-level function's array arguments are implemented as RAM channels. However, when data are being produced

or consumed sequentially, a more efficient data type is to use FIFOs, which can be specified using the stream directive.

- **Pipeline** Given an Initiation Interval (II) parameter, the pipeline directive reduces the number of clock cycles a function or loop can accept new inputs, targeting II clock cycles, by allowing the overlapped execution of operations.
- **Unroll** Given a factor, the unroll directive unrolls a loop factor times, creating multiple instances of the loop body, that can then be scheduled independently or run in parallel
- **Loop Flatten** Allows perfectly nested loops, loops that no logic is injected between them, to get collapsed into a single loop, reducing latency. Essentially, it handles all the indexing logic of the loop flattening
- **Loop Merge** Merges consecutive loops, often initialization loops, reducing overall latency and resource utilization.
- **Resource** Specifies the resource for a variable to get implemented.
- **Array Parititon** By default, every array is implemented as a set of at least one BRAM unit with a single read and a single write port. The array partition directive partitions an array into multiple smaller arrays or assigns each array's element to its register. This partitioning increases the read and write ports of the array on the hardware level, allowing for parallel I/O and computations. In the potential expense of more memory instances and more register, array partitioning can improve overall throughput and performance of memory bounded applications.
- **Array Map** The array map directive combines multiple small arrays into a single large one, to avoid BRAM waste on small arrays, which can occupy a BRAN unit for just a few elements.
- **Array Reshape** Reshapes an array of many elements of small bit-width to an array of fewer elements but of higher bit-width, increasing the sequential BRAM access speeds
- **Data Pack** Similar to the array reshape directive, the data pack directive combines struct data fields to a single scalar of higher bit-width.
- **Dataflow** Enables parallel execution of functions and loops, increasing throughput and latency.

- **Inline** Similar to C/C++ macro preprocessor functionality, the inline directive injects a function's body to each of its calls, reducing latency and initiation interval due to lower function call overhead.
- **Allocation** Limits the number of hardware resources used for implementing the IP block, and may result in hardware sharing and latency increase.
- **Latency** Limits the minimum and maximum latency in clock cycles.

6.1.3 Vitis Analyser

Vitis Analyzer GUI and Window Manager is a useful tool that helps to understand better the system. It provides code reports, summaries and system diagrams. It is spited into three parts the **Report Navigator**, **Report** and **Source Code view**

- **Report Navigator** It provides all the summary files and reports of the project. Also it is easy to open HLS Project or Vivado Project, by right clicking on the Compile Summary or the Run Summary respectively.
- **Reports** It shows all the contents of the summary files. All the reports related to the Compile Summary, Link Summary, or Run Summary are grouped together within a single container.
- **Source Code** The kernel source code.It lets the developer view and edit kernel source code based on the feedback of the report

6.1.4 Vivado IDE

Vivado IDE provides a friendly to the developer GUI. All Vivado Design Suite tools integrate a native TCL interface, which can be accessed from IDE's GUI and the TCL console. Vivado IDE can compile, synthesize, implement, place and route FPGA hardware designs written in high-level languages such as C/C++, and HDLs such as VHDL and Verilog. In addition, using the IP Integrator tool, hardware systems can be designed by graphically connecting IP blocks and configuring them through their GUI, with no coding involved, hence, accelerating the design process. Integration automation features such as auto-connecting and auto-configuring blocks further accelerate the design process. IP blocks can be created using the integrated IP Packaging functionality for VHDL or Verilog designs and via the Xilinx HLS tool for C/C++ designs. FPGA Implementation for free, including but not limited to on and

off-chip-network IPs, memory blocks and memory management IPs, I/O interface IPs, and even various compute IPs. There are also additional IP's that can be purchased from Xilinx or even other vendors and developers

6.2 FPGA Platforms

The platform used in order to accelerate the application was the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [52]. The ZCU102 consists of a quad-core Arm Cortex-A53, dual-core Cortex-R5F real-time processors, and a Mali-400 MP2 graphics processing unit based on Xilinx's 16nm FinFET+ programmable logic fabric

6.3 Architecture Modules

In this section all the modules of the proposed architecture design will be described. Additionally, it will be analyzed witch of the optimization directives used as well as how the Vitis tools (Analyzer, HLS) used in order to detect the bootlenecks and achieve pipeline.

6.3.1 CropImage and DuplicateImage and FAST

In this subsection we will analyze the highlighted modules from the diagram above.

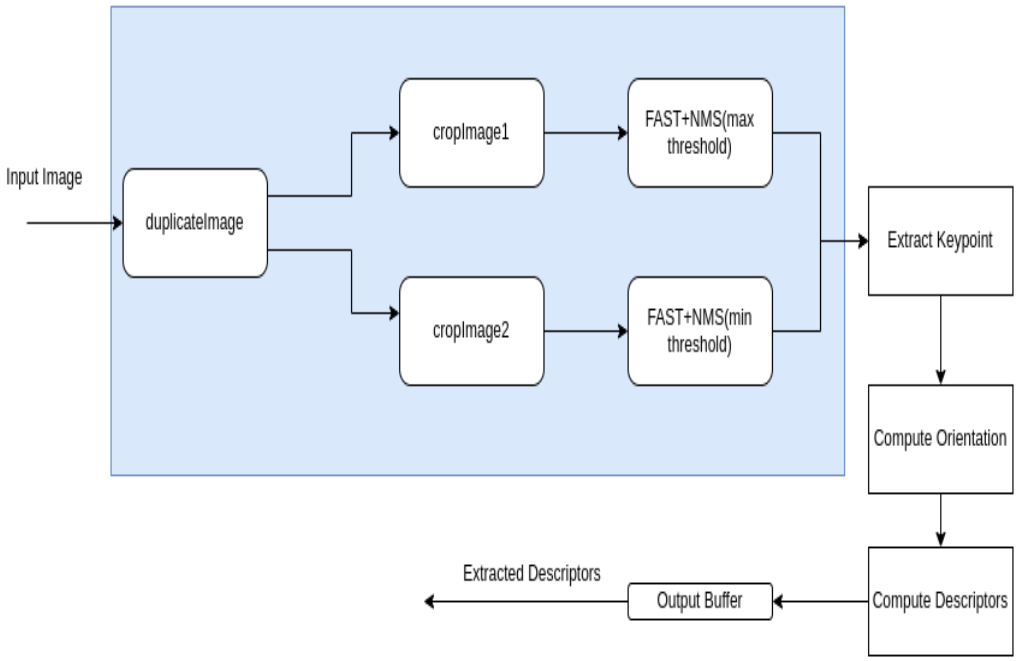

FIGURE 6.1: The modules from the block diagram

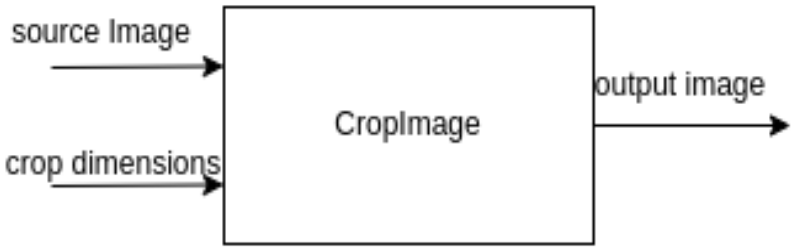

FIGURE 6.2: xfOpencv CropImage

FIGURE 6.3: xfOpencv FAST

In this module three main functions of the *xfOpenCV* library provided by Xilinx was used. These functions was the FAST in order to compute the FAST keypoints of the image, the crop in order to crop the image into 30*30 grids and the duplicate in order to process in parallel the computation of the keypoints with the low and the high threshold.

As we can see at the implementation of algorithm 2, the main image is divided into 30*30 grids. In the software implementation the division is carried out with the following equation.

$$image.rowRange(iniY, maxY).colRange(iniX, maxX)$$

The bellow equation is the input image of the FAST function and it is the 30*30 grid of the image. The hardware implementation of the Mat class of OpenCV library does not allow us to use the *rowRange* and *colRange* methods so we have to find another to divide the image. In order to achive this we used the *xf::cv::crop* function. So we call the crop function depending on the size of the image. If the image is 768*1024 we will need about 884 grids to process all the image. A first thing we tried was to cut the image at the beginning of the code and store the images into a fifo and every time to withdraw data from that fifo but it does not fit at the FPGA. So every time we crop the hole image into a 30*30 grid.

After copping the main image the 30*30 image is ready to fed at the *xf::cv::fast* function in order to calculate the FAST keypoints of that grid. As seen from algorithm 2 the first step is to calculate the FAST keypoints with the initThreshold and if no keypoints found then the FAST keypoints is calculating with the minthreshold. Those two computations can proceed in parallel.

The way to achieve this is to duplicate the input image with the *xf::cv::duplicate*, that is, to create two copies of the input image and process in parallel the computation of crop and FAST functions. In order to process this computation of low and high threshold in parallel the **Dataflow** directive was used. As mentioned before the dataflow enables the parallel execution of functions, in this way the computation of FAST keypoints with min and max threshold process in parallel. After the computation the keypoints are extracted. A diagram of the Dataflow pipeline is shown bellow. We can see that separate grids of the image can executed in parallel.

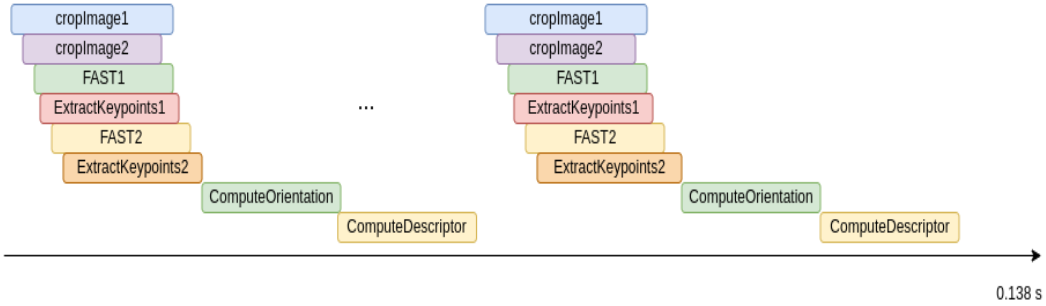

FIGURE 6.4: Dataflow pipeline of the min and max threshold

The figure 6.4 it's about one frame with its 8 levels and the FAST computation is about the computation of the FAST in a grid of a level of the frame.

Extract Keypoints

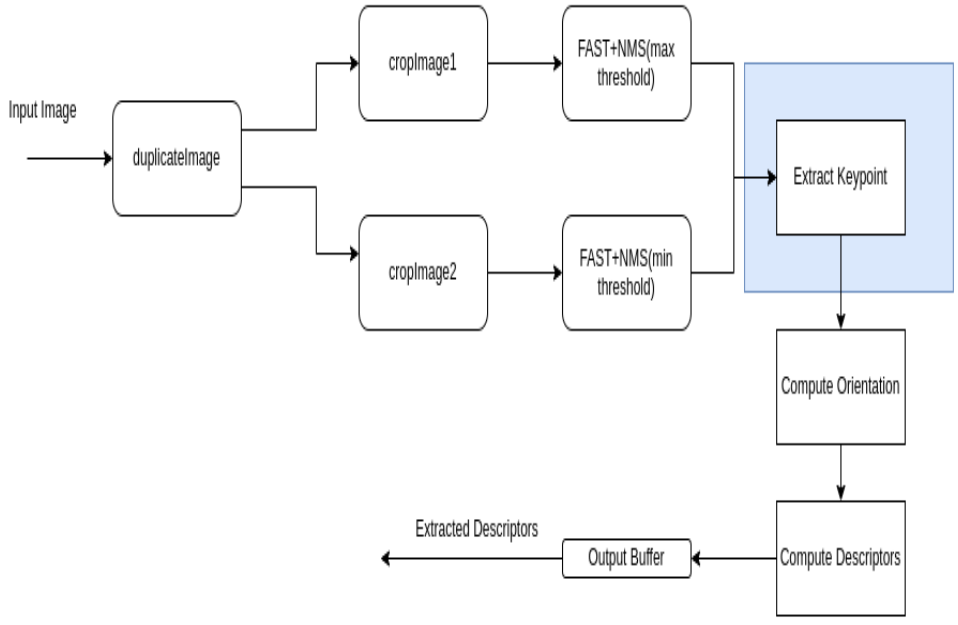

FIGURE 6.5: Extract Keypoints Module

The FAST function takes as input the Mat image in which will find the keypoints, the desired threshold and gives as output a new Mat image in the size of input image in witch the keypoints are marked with the 255 value. So after the computation of keypoints, the extraction process starts. The pixel of the

image are read one by one inside a nested for loop, if the value of the pixel is 255 then is stored as keypoint and fed as input in the Compute Orientation and Descriptors module in order to compute the Descriptors. Those two for loopa are pipelined by using the **Pipeline** directive with initiation interval of 1. The algorithm followed is shown below.

Algorithm 7 Extract Keypoints

```
    procedure EXTRACTKEYPOINTS
2:      for Every row do
            for Every col do
4:              input_pixel ← Input_Image(row, cols)   ▷ Read every pixel from the grid
                if input_pixel ≡ 255 then
6:                  Storethekeypoint
                    CallComputeOrientation
8:                  CallComputeDescriptor
```

6.3.2 Compute Orientation

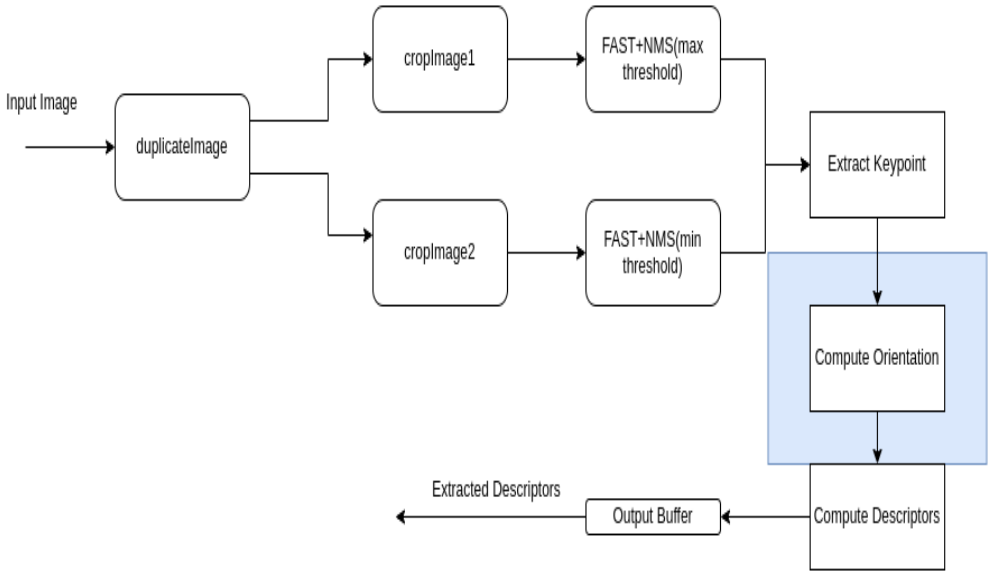

FIGURE 6.6: Compute Orientation Module

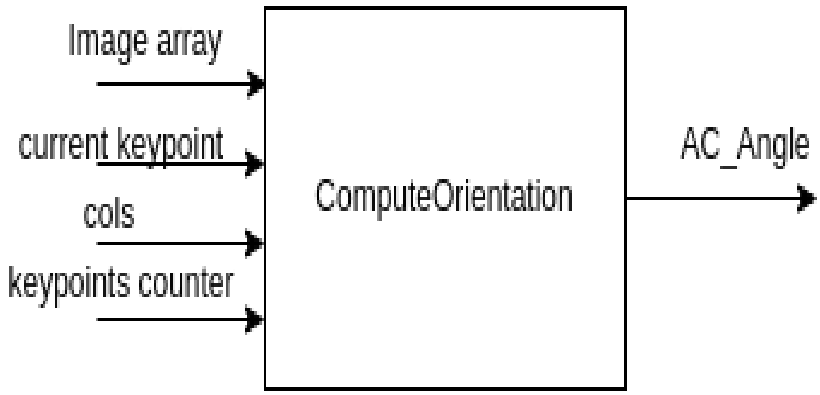

FIGURE 6.7: Compute Orientation

This module implements the algorithm 4. In the main algorithm after the computation of all keypoints, their orientation is computing as follows, algortihm 3 calls algorithm 4 as many times as the number of keypoints is. Instead of this we call only algorithm 4 every time a keypoint is extracted. With this way we cut off a for loop and we achieve pipeline as well as until the extraction of keypoint is done the calculation of the orientation in completed and fed at the compute descriptor module, however the extraction of the next keypoint has began. A diagram of the pipeline is shown below.

As we can see from the figures 6.8 and 6.9 the algorithm takes as input a frame computes the Image Pyramid in the beginning and the for every level computes the keypoints, the orientations and finally the descriptors. So, the compute pyramid part is calculated one time for every frame instead of the other parts that are calculated for every 30*30 grid for every level of the image. We deepen at this part because of there is no dependence between the grid instead of the pyramid that there is dependence between the next and the previous level of the pyramid. We detect that there is no dependency between the grids of the image so we pipeline this part of the algorithm.

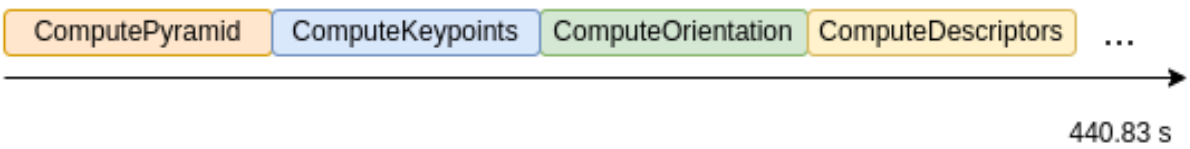

FIGURE 6.8: Without Dataflow Pipelining

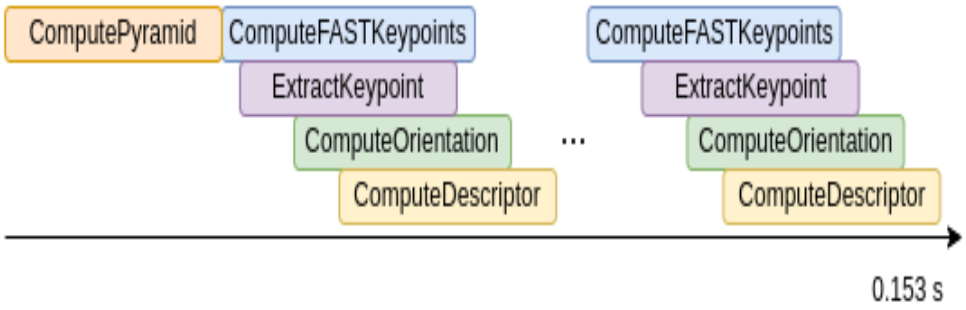

FIGURE 6.9: With Dataflow Pipelining

As it seems at the implementation of the algorithm 4 the calculation of the IC angle of the keypoints is computed in two phase. The first phase is initializing the parameter m10 with a single for loop and the second phase is computing both the parameters m10, m01 with a nested for loop all these for loop are fully pipelined with the **Pipeline** directive. In order to pipeline these loops with II of 1 we need to fully partition the array of the image is needed, this done with the **Array Partition** directive.

6.3.3 Compute Descriptors

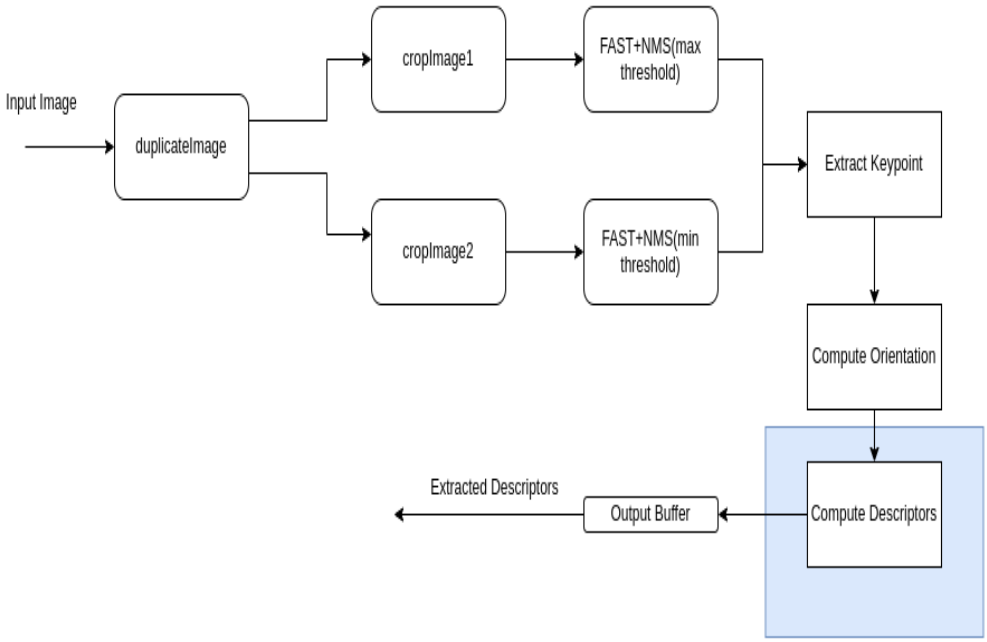

FIGURE 6.10: Compute Descriptors Module

FIGURE 6.11: Compute Descriptors

This module implements the algorithm 6. As in the previous module the Descriptors in computing after the calculation of all keypoints and all the angles of them. So, algorithm 5 calls algorithm 6 as many times as the numbers of keypoints is. As mentioned before instead of this implementation we compute the descriptors imidiately after the extraction of keypoint and the computation of its angle.

As we can see from the implementation of algorithm 6 there is a single for loop that calculates the 32-bit descriptor of the keypoint. In order to pipeline this loop, there two things that must be done. The first is to partition the array of the image that is accessed at ever iteration of the loop, and the second is to partition the pattern array. The pattern array is a predefined array with coordinates that used for the calculation of the descriptor. This array consist of 1024 pair of coordinates, we split this array into two different arrays one for the x coordinates and one for the y coordinates and we implement those arrays as read only ram with multiple read ports this implemented with the **Resource** directive and we fully partition the arrays. In this way the 32 iteration loop of the computation of the descriptors is fully pipelined.

In conclusion, as described above every time a keypoint is extracted the orientation and the descriptor is computed instead of the software implementation in which firstly all the keypoints are computed, then the orientation and after orientation the descriptors are computed. We choose this way because we can cut of 2 large for loops those for the orientation and the descriptor computation and also we can increase the pipeline.

Chapter 7

Verification on an FPGA platform and Performance Evaluation

In this chapter we will analyze the results of our design vs. the theoretical performance. The design was performed with the Xilinx Vitis HLS and Vivado IDE tools and was carried out past the place and route step of the design on the ZCU102 Evaluation Kit. We will present and analyse some performance metrics such as latency, throughput, power ans energy consumption and the results of validation will be presented. Lastly, we will compare those metrics with the corresponding ones from software implementation.

7.1 Validation

In order to validate the proposed solution a suite of 20 images was used. We run the software algorithm in order to calculate the FAST features of those 20 images and repeat the process with the proposed architecture. Above we can see the results.

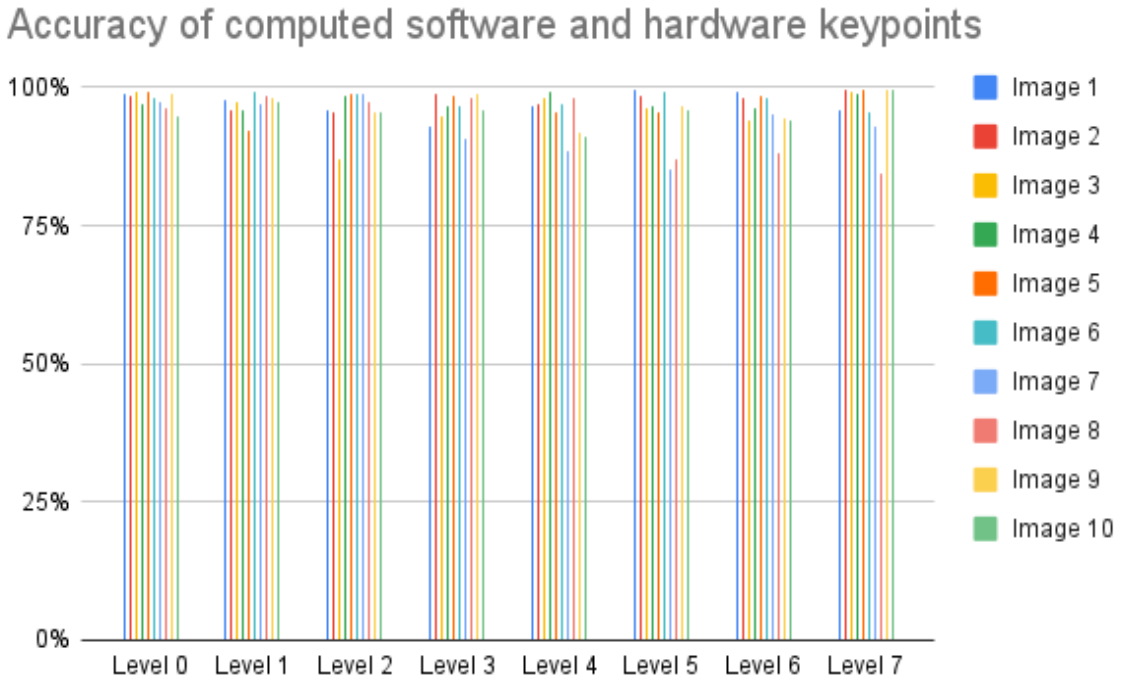

FIGURE 7.1: Percentage of keypoints found in hardware vs. the software model, per image and per level

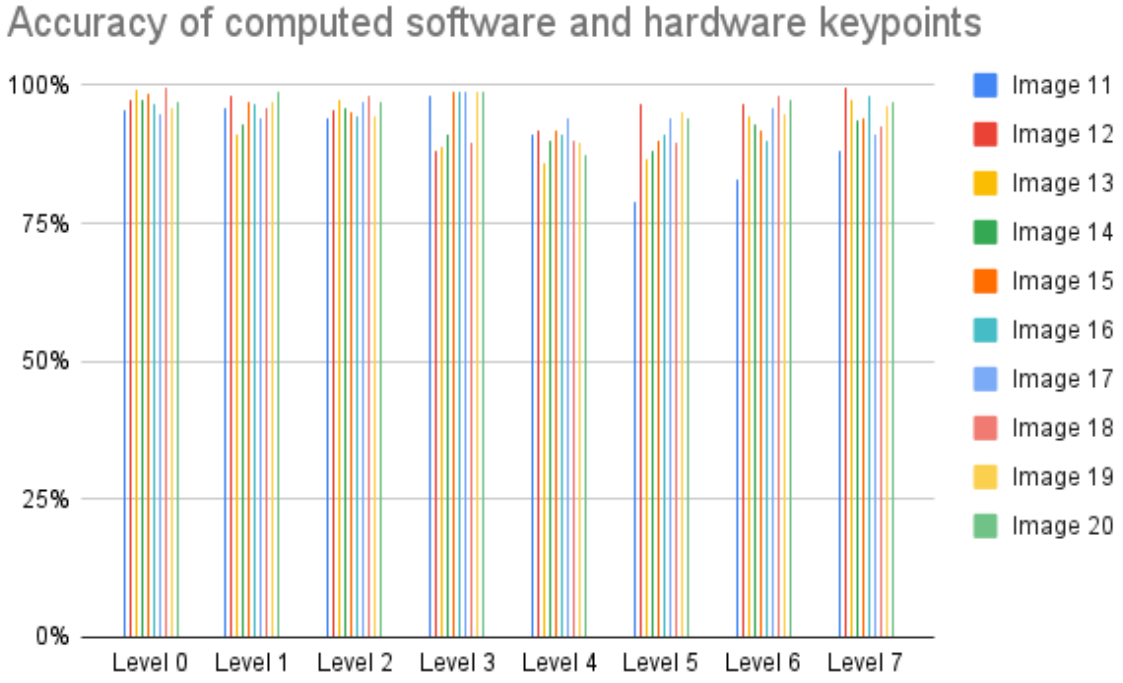

FIGURE 7.2: Percentage of keypoints found in hardware vs. the software model, per image and per level

As we can see the FAST keypoints extracted from both Software and Hardware are very similar with a loss 5-10% per level. Also, hardware in some times can find some more keypoints compared with software but there are not enough to discuss further. This experiment done for many datasets but there much in order to present thm all.

7.2 Specification of Compared Platforms

The design which we described in previous chapters and section will be compared vs. the code of the ORB-SLAM2 that has run on an Intel i7-6500U. The proposed architecture was designed to run on the ZCU102 Evaluation board but due to technical problems with the platform the results shown below refer to post place and route simulations with the Vitis HLS and Vivado IDE tools. Post place and route simulations are highly accurate, as all issues (including timing ones) have been fully addressed, i.e. the design fits on the resources of the platform, the maximum clock frequency is highly accurate, and all internal timings and interfaces to external DDR memory are accurate. It should be noted that DDR is not a bottleneck in this design, as the frame rate is very low, hence it is the on-FPGA processing itself which determines the latency.

7.2.1 Intel i7-6500U

Intel i7-6500U	
Cores / Threads	2 / 4
Max Turbo Frequency	3.10 GHz
TDP	15W
Max Memory Bandwidth	34.1 GB/s
Lithography	14 nm

TABLE 7.1: CPU characteristics

7.2.2 Proposed Solution

Here is the final resources usage of our solution.

Resources usage	
Clock Frequency(MHz)	205.5
BRAM	66%
DSPs	1%
FF	3%
LUTs	12%

TABLE 7.2: ZCU102 Resources usage

7.3 Performance Metrics

7.3.1 Latency

Latency, is the time required for accomplishing a single task. It is preferred to be as low as possible to finish tasks as quickly as possible from the time they are issued. Latency is given by the following equation.

$$Latency = \frac{1}{v} = \frac{T}{W}$$

Where, v is the task's execution speed, T is the task's execution time, W is the task's execution workload.

7.3.2 Throughput

Throughput, is a measure of how many units of information a system can process in a given amount of time. It is preferred to be as high as possible to generate as much work as possible in the unit time. Throughput is given by the following equation.

$$Throughput = r * v * A = \frac{r * A * W}{T} = \frac{r * A}{L}$$

Where, r is the execution density, A is the execution capacity.

7.3.3 Power Consumption

Power consumption is defined as the energy consumed per unit time for accomplishing a specific task. Average power consumption is always preferred to be as low as possible to increase the system's energy efficiency, minimizing energy losses. In addition, low power consumption leads to simpler system designs and lower building costs. It is usually measured in Watts (w) or kilo-Watts (kW).

7.3.4 Energy Consumption

Energy consumption is defined as the energy required for accomplishing a specific task in a specific time amount. Energy consumption is also preferred to be as low as possible while accomplishing the given task within the time constraints, to minimize the operational costs. It is usually measured in Joule (J) or kiloJoule (kJ).

7.4 Final Performance

The Dataset used in order to test the design was the Marina in Fremantle at Western Australia as mentioned in subsection 3.4. Specifically we used one of the sub datasets of the Marina in Fremantle with 2-3 meters depth and cloudy day. This dataset was choosen because in this depth and weather conditions more keypoints are found (as characterized by the algorithm and not the implementation). It consits of 2,064 images with initial resolutuion of 768x1024 pixels. All the images are processed as gray-scale.

The final results of the architecture are shown below.

Performance Results		
	CPU	Proposed Solution
Clock Frequency(MHz)	3100	205
Throughput (Images/s)	9.5	6.5
Latency(s)	0.1065	0.153
Execution time of 2,064 images(s)	220	315.8
Total On-Chip Power(Watt)	15	4.12
Energy Cons./Image (Joule)	1.57	0.633
Images/Joule	0.633	1.58

TABLE 7.3: Proposed Architecture Performance results

As mentioned, the performance results refers to theoretical results after performing Place and Route as we encountered technical problems and was impossible running the implementation at the FPGA. The Energy Consumption/Image metric is calculated with the following equation.

$$\frac{EnergyConsumption}{Image} = \frac{TotalPower}{Throughput}$$

The Images/Joule metric is calculated with the following equation.

$$\frac{Images}{Joule} = \frac{Throughtput}{TotalPower}$$

The total speedup achive is shown in the following table.

Speedup Results		
	CPU	Proposed Solution
Throughput	1x	0.68x
Latency	1x	0.69x
Power Efficiency	1x	3.64x
Energy Efficiency	1x	2.5x

TABLE 7.4: Proposed Architecture Speedup results

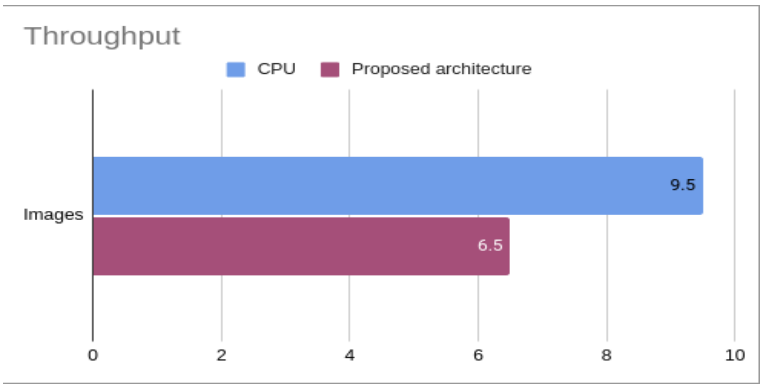

FIGURE 7.3: Throughput Final Result

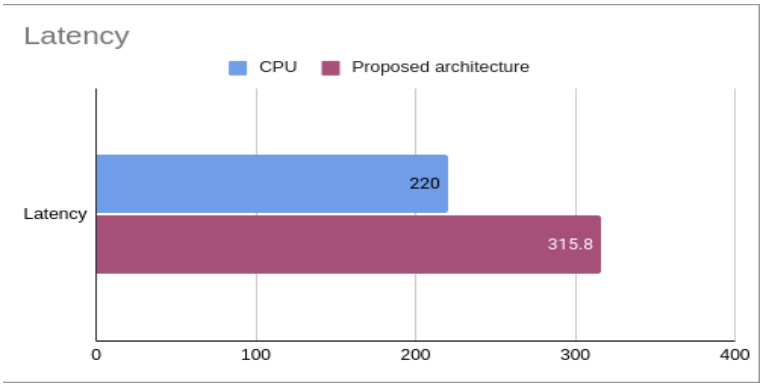

FIGURE 7.4: Latency Final Result

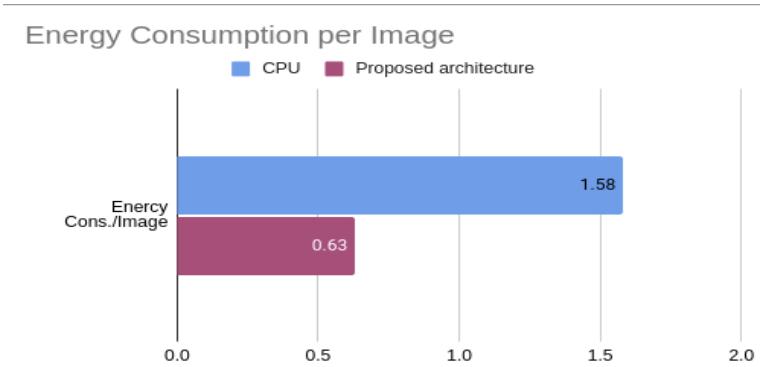

FIGURE 7.5: Energy Consumption per Image Final Result

As we can see from the results above, in terms of Throughput and Latency the CPU solution is faster than the proposed solution but the energy witch requires is bigger than the proposed solution. In this thesis we aim to use this solution in unmanned submarines, so we need real time implementation with low energy cost in order the submarine can explore as much time as it is possible. The proposed solution might be more slow but is about *60%* less energy costly than the CPU. At this point let us mention that the metrics of the CPU were taken with an Intel i7 processor witch may be not a choice for a submarine due to energy consumption, a choice for a submarine may be an Intel Pentium with low energy consumption this leads to the result that the performance will not be the same as the Intel i7 it will be similar or even less than the proposed solution.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Autonomous Underwater Submarines have many useful application such as garbage collection in shallow ports, mapping ports even finding holes in fishing nets and more other. In order to achieve this we need a real time implementation algorithm with low energy cost. This can be achieved with the use of FPGA. The purposed solution may be not meet the timing constraints but it is much better in terms of energy. If this solution is extended incorporated into the ORB-SLAM2 algorithm and accelerate some other timing costly parts of the algorithm such as ORB maching part, pipeline the execution of images as we will mention in chapter 8.2 the solution will be timing and energy effective.

8.2 Future Work

This work is easily expandable and also can be applied into newer platforms such as GPUs. Some of future works can be done will be mentioned bellow.

- **Apply the accelerator into ORB-SLAM2** As mentioned this work refers to the acceleration of the most timing consuming part of the ORB-SLAM2 algorithm the ORB feature extraction part. This accelerator can replace the ORB extraction of the ORB-SLAM2 algorithm.
- **Accelerate other parts of algorithm** Some other parts of the algorithm even the hole algorithm can be accelerated

- **Apply the architecture in a GPU** When this work starts the only available resources was the ZCU 102 board so we can not apply the architecture in a GPU. Now there are more powerful GPUs like Jetson in witch this architecture solution can be applied.
- **Better Memory management** The purpose of our work was not only to reduce the energy cost but also the comparison of the functions of opencv with counterpart of xfopencv, so we process the entire image. A future solution will be the use of FIFOs so that the process can be pixel by pixel and achieve a better pipeline.
- **Better resource managment** ZCU 102 has the ability of using multiple compute units. This means that there can be multiple copies of the same kernel, so it can parallel process many images.

References

[1] *OpenCV Library*. https://opencv.org/.

[2] *xfOpenCV Library*. https://github.com/Xilinx/xfopencv.

[3] H. Durrant-Whyte and T. Bailey. "Simultaneous localization and mapping: part I". In: *IEEE Robotics Automation Magazine* 13.2 (2006), pp. 99–110. DOI: 10.1109/MRA.2006.1638022.

[4] H. Durrant-Whyte and T. Bailey. "Simultaneous localization and mapping(SLAM): part II". In: (Aug. 2006).

[5] Talha Takleh Omar Takleh et al. "A Brief Survey on SLAM Methods in Autonomous Vehicle". In: *International Journal of Engineering and Technology(UAE)* 7 (Nov. 2018), pp. 38–43. DOI: 10.14419/ijet.v7i4.27.22477.

[6] Zoran Sjanic et al. "Solving the SLAM Problem for Unmanned Aerial Vehicles Using Smoothed Estimates". In: *Proceedings of the 18th World Congress of the International Federation of Automatic Control (IFAC)* (Aug. 2011).

[7] Felipe Guth et al. "Underwater SLAM: Challenges, state of the art, algorithms and a new biologically-inspired approach". In: *Proceedings of the IEEE RAS and EMBS International Conference on Biomedical Robotics and Biomechatronics* (Aug. 2014), pp. 981–986. DOI: 10.1109/BIOROB.2014.6913908.

[8] Dimitrios Geromichalos et al. "SLAM for autonomous planetary rovers with global localization". In: *Journal of Field Robotics* 37.5 (2020), pp. 830–847. DOI: https://doi.org/10.1002/rob.21943.

[9] Andrew J. Davison and David W. Murray. "Mobile Robot Localisation using Active Visual Sensing". In: (1998).

[10] Germán Rosand et al. "Visual SLAM for Driverless Cars : A Brief Survey". In: (2012).

[11] Rickard Karlsson et al. "Utilizing Model Structure for Efficient Simultaneous Localization and Mapping for a UAV Application". In: (2008).

[12] John Folkesson and John Leonard. "Autonomy through SLAM for an Underwater Robot". In: ().

[13] Stephen Se, Tim Barfoot, and Piotr Jasiobedzki. “Visual Motion Estimation and Terrain Modeling for Planetary Rovers”. In: (2005).

[14] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. “Visual SLAM algorithms: a survey from 2010 to 2016”. In: (2017).

[15] S.B. Williams et al. “Autonomous underwater simultaneous localisation and map building”. In: (2002).

[16] R. Eustice, O. Pizarro, and H. Singh. “Visually augmented navigation in an unstructured environment using a delayed state history”. In: (2004).

[17] Guilherme Zaffari et al. “Effects of Water Currents in a Continuous Attractor Neural Network for SLAM Applications”. In: (2016).

[18] Luan Silveira et al. “An Open-source Bio-inspired Solution to Underwater SLAM”. In: (2015).

[19] Franco Hidalgo, Chris Kahlefendt, and Thomas Braunl. “Monocular ORB-SLAM Application in Underwater Scenarios”. In: (2018).

[20] Raul Mur-Artal and J.M.M.Montiel. “ORB-SLAM : A Versatile and Accurate Monocular SLAM System”. In: (2015).

[21] Edward Rosten and Tom Drummond. “Machine Learning for High-Speed Corner Detection”. In: (2006).

[22] Michael Calonder et al. “BRIEF Binary Robust Independent Elementary Features”. In: (2010).

[23] Dorian Galvez-López and Juan D. Tardos. “Bags of Binary Words for Fast Place Recognition in Image Sequences”. In: (2012).

[24] Juan D. Tardos Raul Mur-Artal. “ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras”. In: *IEEE Transactions on Robotics* ().

[25] Hauke Strasdat et al. “Double window optimisation for constant time visual SLAM”. In: (2011).

[26] Lina M. Paz et al. “Large-Scale 6-DOF SLAM With Stereo-in-Hand”. In: (2008).

[27] *Robust Hyber Cost function*. https://en.wikipedia.org/wiki/Huber_loss.

[28] Ethan Rublee et al. “ORB: an efficient alternative to SIFT or SURF”. In: (2014).

[29] Paul L. Rosin. “Measuring Corner Properties”. In: (1999).

[30] Quentin Gautier, Alric Althoff, and Ryan Kastner. “FPGA Architectures for Real-time Dense SLAM”. In: (2019).

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

[32] Grigorios Mingas, Emmanouil Tsardoulias, and Loukas Petrou. "An FPGA implementation of the SMG-SLAM algorithm". In: (2012).

[33] Jorge Artieda et al. "Visual 3-D SLAM from UAVs". In: (2009).

[34] Camilo Sánchez-Ferreira et al. "Development of a stereo vision measurement architecture for an underwater robot". In: (2013).

[35] Emanuel Trabes and Mario A. Jordan. "Self-tuning of a sunlight-deflickering filter for moving scenes underwater". In: (2015).

[36] Sanjay K. Dhurandher et al. "UWSim: A Simulator for Underwater Sensor Networks". In: (2008).

[37] David Ribas et al. "The Girona 500, a multipurpose autonomous underwater vehicle". In: (2011).

[38] Quentin Gautier, Alric Althoff, and Ryan Kastner. "FPGA Architectures for Real-time Dense SLAM". In: 2160-052X (2019), pp. 83–90. DOI: 10.1109/ASAP.2019.00-25.

[39] Keisuke Sugiura and Hiroki Matsutani. "An FPGA Acceleration and Optimization Techniques for 2D LiDAR SLAM Algorithm". In: (May 2020).

[40] Konstantinos Boikos and Christos-Savvas Bouganis. "Semi-Dense SLAM on an FPGA SoC". In: ().

[41] Konstantinos Boikos and Christos-Savvas Bouganis. "A-High-Performance-System-on-Chip-Architecture-for-Direct-Tracking-for-SLAM". In: (2017).

[42] Weikang Fangy et al. "FPGA-based ORB Feature Extraction for Real-Time Visual SLAM". In: (2017).

[43] Runze Liu et al. "eSLAM: An Energy-Efficient Accelerator for Real-Time ORB-SLAM on FPGA Platform". In: (2019).

[44] Amanda C. Duarte et al. "Towards Comparison of Underwater SLAM Methods: An Open Dataset Collection". In: (2018).

[45] Maxime Ferrera et al. "AQUALOC: An Underwater Dataset for Visual-Inertial-Pressure Localization." In: (2019).

[46] Franco Hidalgo Chris Kahlefendt Thomas Braunl. "Monocular ORB-SLAM Application in Underwater Scenarios". In: (2018).

[47] *ORB-SLAM2 algorithm*. https://github.com/raulmur/ORB_SLAM2.

[48] *Image Pyrmaid*. https://www.google.com/url?sa=i&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FPyramid_(image_processing)&psig=AOvVaw2lGolGs3NK9S_5lJbrx3QM&ust=1665416992099000&source=images&cd=vfe&ved=0CAwQjRxqFwoTCIj69vO_0_oCFQAAAAAdAAAAABAE.

[49] *Vitis High-Level Synthesis*. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[50] *Xilinx Runtime Library*. https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[51] *Vitis High-Level Synthesis*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[52] *Zynq Ultrascale+ Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.