VICTOR HUGO SCHULZ

EMBEDDED LANDMARK ACQUISITION SYSTEM FOR VISUAL SLAM USING STAR IDENTIFICATION BASED STEREO CORRESPONDENCE DESCRIPTOR

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre do Programa de Pós-Graduação em Informática, apresentada ao Departamento de Informática, Setor de Ciências Exatas da Universidade Federal do Paraná.

Orientador: Prof. Dr. Eduardo Todt

CURITIBA

2015

VICTOR HUGO SCHULZ

EMBEDDED LANDMARK ACQUISITION SYSTEM FOR VISUAL SLAM USING STAR IDENTIFICATION BASED STEREO CORRESPONDENCE DESCRIPTOR

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre do Programa de Pós-Graduação em Informática, apresentada ao Departamento de Informática, Setor de Ciências Exatas da Universidade Federal do Paraná.

Orientador: Prof. Dr. Eduardo Todt

CURITIBA

2015

S388e Schulz, Victor Hugo
Embedded landmark acquisition system for visual slam using star identification based stereo correspondence descriptor/ Victor Hugo Schulz. – Curitiba, 2015.
162 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2015.

Orientador: Eduardo Todt .
Bibliografia: p. 158-162.

1. Visão por computador. 2. Sistemas embarcados (Computadores). 3. Profundidade - Percepção. I. Universidade Federal do Paraná. II.Todt, Eduardo. III. Título.

CDD: 006.37

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Victor Hugo Schulz, avaliamos o trabalho intitulado, *"Embedded landmark acquisition system for visual SLAM using star identification based stereo correspondence descriptor"*, cuja defesa foi realizada no dia 13 de abril de 2015, às 09:30 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:
(X)**aprovação** do candidato. ( )**reprovação** do candidato.

Curitiba, 13 de abril de 2015.

Prof. Dr. Eduardo Todt
PPGInf/UFPR – Orientador

Prof. Dr. Eduardo Augusto Bezerra
UFSC - Membro Externo

Prof. Dr. Bruno Müller
UFPR – Membro Externo

Prof. Dr. Roberto André Hexsel
PPGInf/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

DEDICATÓRIA

Dedico este trabalho ao meu pai, Jorge Gruhn Schulz, à minha mãe, Lurdes Eumar Schulz, e à minha namorada Letícia da Silva Aléo pelo incentivo, amor e apoio durante as diversas etapas do trabalho. Dedico também a meu orientador, Prof. Dr. Eduardo Todt, por toda a ajuda e inspiração durante a elaboração, contribuindo muito para um resultado que nada mais é do que um reflexo de sua paixão sobre o assunto.

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Dr. Eduardo Todt, por todo tempo que dedicou a este trabalho e ao meu colega Leonardo de Amaral Vidal por ter compartilhado toda a experiência de sua jornada no mestrado, o que me permitiu ir muito mais longe no meu próprio caminho. Agradeço também a Jeferson Fernando Guardezi, Felipe Bombardelli, Nathaly Gasparin, Nicole Salomons, Icaro Oliveira e Luiz Augusto Volpi do Nascimento, que me ajudaram diretamente ou discutiram ideias que enriqueceram a qualidade deste trabalho.

ABSTRACT

The use of cameras as the main sensors in Simultaneous Localization and Mapping, what is called Visual SLAM, has risen recently due to the fall in camera prices. While images bring richer information than other typical SLAM sensors, such as lasers and sonars, there is significant extra processing cost when they are used. The extra depth information available from stereo camera setups makes them preferable for SLAM applications. In this particular approach, great part of the added processing cost comes from extracting unique points or image patches in both stereo images and solving the correspondence problem between them. With this information, the horizontal disparity between the pair can be used to retrieve depth information.

This work explores the use of an embedded system-on-a-chip (SoC) platform that integrates a multicore ARM processor with FPGA fabric as a stereo vision processing module. The Harris and Stephens corner detector (Harris & Stephens, 1988) is used to find Point of Interests (POIs) in stereo images in a hardware soft co-processor synthesized in the FPGA to speed up feature extraction and relieve this highly parallelizable process from the main embedded processor. Remaining tasks such as image correction from camera calibration, finding unique descriptor for the detected features and the correspondence between POIs in the stereo pair are solved in software running on the main processor. The proposed architecture for the co-processor enabled the corner extraction task to be performed in about half the time taken by the main processor without aid.

After finding the POIs, for each point a unique descriptor is needed for finding the correspondent POI in the other image. This work also proposes an innovative descriptor that considers a global two-dimensional spatial relationship between the detected points to describe them individually. In each image, every point in the cloud of points detected by the Harris and Stephens algorithm is described by considering only the relative position between it and its neighbors. When position alone is considered, a starry night pattern is formed by the POIs. With the POI pattern being considered as stars, the descriptors already used in star identification problems can be reapplied to uniquely identify POIs. A prototype of the descriptor based on the Padgett and KreutzDelgado's grid algorithm (Padgett & KreutzDelgado, 1997) is written and the results compared with common descriptors used for this purpose, showing that two-dimensional spatial information alone can be used to solve the correspondence problem. The number of useful matches was comparable to what was

obtained with SIFT, the best performing descriptor in this matter, while the speed was superior to BRIEF, the fastest descriptor used in the comparison, on the ARM platform, with a speedup of 1.64 and 1.40 on the tested datasets.

Keywords: Harris; FPGA; SLAM; Reconfigurable Hardware; VHDL; Image Processing; Stereo Vision; Computer Vision; Hybrid Architecture; Embedded Systems; Point Of Interest; Keypoints; Matching; Stereo Correspondence; Star Identification; Feature Description; Depth Perception.

RESUMO

O uso de câmeras como sensores principais em Localização e Mapeamento Simultâneos (*Simultaneous Localization and Mapping*), o que é denominado SLAM Visual (*Visual SLAM*), tem crescido recentemente devido à queda nos preços das câmeras. Ao mesmo tempo em que imagens trazem informações mais ricas do que outros sensores típicos empregados em aplicações SLAM, como lasers e sonares, há um custo adicional de processamento significativo quando elas são utilizadas. A informação de profundidade adicional proveniente de configurações estéreo de câmeras às fazem mais interessantes para aplicações SLAM. Nesta abordagem em especial, grande parte do custo de processamento adicional vem da extração de pontos únicos ou pedaços em ambas as imagens em estéreo e da solução do problema de correspondência entre eles. Com posse dessa informação, a disparidade horizontal entre o par de imagens pode ser utilizada para recuperar a informação de profundidade.

Esse trabalho explora a utilização de uma plataforma embarcada do tipo *system-on-a-chip* (SoC) que integra um processador ARM multinúcleo com lógica FPGA como um módulo de processamento para visão estéreo. O detector de cantos Harris e Stephens (Harris & Stephens, 1988) é usado para encontrar pontos de interesse (*Points of Interest*, POIs) em imagens estéreo em um coprocessador *soft* sintetizado no FPGA para acelerar a extração de características e livrar o processador principal deste processo altamente paralelizável. As tarefas restantes tais como correção das imagens pela calibração de câmeras, encontrar um descritor único para as características detectadas e a correspondência entre os POIs no par de imagens estéreo são solucionadas em software executando no processador principal. A arquitetura proposta para o coprocessador permite que a tarefa de extração de cantos seja executada em aproximadamente metade do tempo necessário pelo processador principal sem auxílio algum.

Após encontrar os POIs, para cada um dos pontos um descritor único é necessário para que seja possível encontrar o POI correspondente na outra imagem. Esse trabalho também propõe um descritor inovador que considera o relacionamento espacial bidimensional global entre os pontos detectados para descrevê-los individualmente. Para cada imagem, cada ponto da nuvem de pontos detectada pelo algoritmo de Harris e Stephens é descrito considerando-se apenas as posições relativas entre ele e seus vizinhos. Quando somente a posição é considerada, um padrão de céu estrelado noturno é formado pelos POIs. Com o

padrão de POIs sendo considerado como estrelas, descritores já utilizados em problemas de identificação de estrelas podem ser reaplicados para identificar unicamente POIs. Um protótipo do descritor baseado do algoritmo de grade de Padgett e KreutzDelgado (Padgett & KreutzDelgado, 1997) é escrito e seus resultados comparados com os descritores normalmente utilizados para este propósito, mostrando que a informação espacial bidimensional pode ser utilizada por si só para resolver o problema de correspondência. O número de correspondências úteis é comparável ao atingido com o SIFT, o descritor com melhor desempenho neste quesito, enquanto a velocidade foi superior ao BRIEF, o descritor mais rápido utilizado na comparação, na plataforma ARM, com um *speedup* de 1,64 e 1,40 nas bases de dados dos testes.

Palavras-chave: Harris; FPGA; SLAM; Hardware Reconfigurável; VHDL; Processamento de Imagem; Visão Estéreo; Computer Vision; Arquitetura Híbrida; Sistemas Embarcados; Pontos de Interesse; Keypoints; Correspondência; Correspondência Estéreo; Identificação de Estrelas; Descrição de Características; Percepção de Profundidade.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1. INTRODUCTION

Mobile robotics is a relatively young field, comprised of multiple distinct disciplines, from mechanical, electrical and electronic engineering to computer, cognitive and social sciences. One of its main concerns is giving robots the ability of moving on their own, that is, without human supervision, on varied environments (Siegwart, Nourbakhsh, & Scaramuzza, 2011).

Within this subject, one problem requires attention: the possibility of a robot being placed in an unknown environment at an unknown location and being able to construct a consistent map of its surroundings while localizing itself within it as it moves. This is known as the Simultaneous Localization and Mapping problem, or SLAM. While SLAM can be considered a solved problem, some substantial issues remain in the practical realization of more general SLAM solutions, and using perceptually rich maps as a part of a SLAM algorithm (Durrant-Whyte & Bailey, 2006).

For SLAM, the information available to the robot at any time comes from odometry and exteroceptive sensors such as lasers, ultrasound and cameras. In real world, this information is corrupted by noise, which increases the difficulty of the problem (Siegwart et al., 2011).

The most common solution for the SLAM problem is based on the Extended Kalman Filter (EKF). In EKF-SLAM the map is modeled by a Gaussian distribution, where the robot pose and landmark positions are represented by a state vector for the means and a matrix for the covariance (Solà, 2013). When an observation is made, both the state vector and the joint covariance matrix are updated. In practice, this leads to a quadratically growth on the computation with the number of landmarks (Durrant-Whyte & Bailey, 2006).

The high complexity that arises from using rich maps populated with many landmarks calls for implementations that benefit from parallel architecture speedups. These optimizations should be in specific parts of the code that can benefit from parallel implementation. Recent examples of hybrid architectures that can benefit from this offload work to a GPU using solutions such as NVidia CUDA (NVIDIA Corp., 2014). This leaves the inherently sequential part of the code to the CPU, which is better fitted for the job.

Other platforms that allow hybrid processing are the new System on a Chip (SoC) solutions that integrate dual-core ARM processors with an FPGA fabric, such as Xilinx Zynq-7000 (Xilinx Inc., 2014c) and Altera Arria V, Arria 10 and Stratix 10 SoC series (Altera Corp., 2014b). While less powerful than CPU/GPGPU processing, SoC solutions require less power, benefitting implementations for robot applications that require embedded processing in the robot itself.

When using a camera as a sensor for robots, the 3D world is projected into a 2D image plane. This process reduces information, and consequently depth perception is lost. One of the ways to later recover that information is by capturing different images simultaneously with two cameras, when their relative position is known. This is called stereo vision, and by doing it, perceived horizontal disparity between images is used as a means to retrieve depth information. Two major problems arise in stereo vision, the correspondence problem and 3D reconstruction. The first consists in matching the points of the two images which are the projection of the same point in the scene. When the correspondence is done, it is possible to reconstruct the structure of the scene, arriving in the second problem. The solution of both problems requires that intrinsic and extrinsic parameters about the cameras are known through calibration (Siegwart et al., 2011).

Current solutions for the stereo correspondence problem can be distinguished in two main categories, area-based and feature-based. In the former, a small patch (window) of the image is looked for in the other image in order to find the most similar one. In the latter, Points of Interest (POIs) are found in both images, and then matched according to local descriptors (Siegwart et al., 2011).

For area based approaches, visual spatial information used is limited within the chosen window. Feature based approaches on the other hand usually only consider the immediate surroundings of a POI for describing it. Mortensen, Deng, and Shapiro (2005) expanded the local SIFT descriptor (Lowe, 2004) using a global context vector similar to shape contexts, adding curvilinear shape information, what increased its robustness to local appearance ambiguity and non-rigid transformations.

This work is based on the same premises, that considering a larger neighborhood to the POI could yield positive results when finding its correspondence. It differs by exploring a different feature and descriptor and that the correspondence is found independently of any

other POI descriptors (including SIFT), so that the spatial information is used alone for finding the correspondences.

In the proposed solution, POIs can be detected using standard algorithms such as the Harris corner detector (Harris & Stephens, 1988), Scale Invariant Feature Transform (SIFT) (Lowe, 2004), Speeded Up Robust Features (SURF) (Bay, Tuytelaars, & Van Gool, 2006) and Features from Accelerated Segment Test (FAST) (Rosten & Drummond, 2006). The proposed feature investigated in this work is based on the fact that POIs have intrinsically a high repeatability property Therefore, POIs with 3D coordinates close to each other should also produce 2D coordinates on both image planes from the stereo cameras with a high probability of maintaining their relative positions within the small variation in the Point of View of the two cameras.

For evaluating the use of a hybrid processing architecture, a Xilinx Zynq-7000 SoC processor with a dual-core ARM processor and integrated FPGA was used. A hardware co-processor was implemented in VHDL and synthesized in the FPGA to accelerate the detection of POIs by using the Harris and Stephens corner detector (Harris & Stephens, 1988). The processes of image acquisition, description of the hardware-detected points and finding the correspondence between them are performed in software running in GNU/Linux on the ARM processor.

Figure 1 shows a simplified block diagram of the steps required in feature-based landmark acquisition from a stereo camera setup for Visual SLAM. As previously explained, only the POI detector step was implemented as a hardware co-processor, while all other tasks run in software at the main processor. The left and right images are processed sequentially on the co-processor. The POI descriptor step is implemented using the new proposed solution, and is also implemented as software.

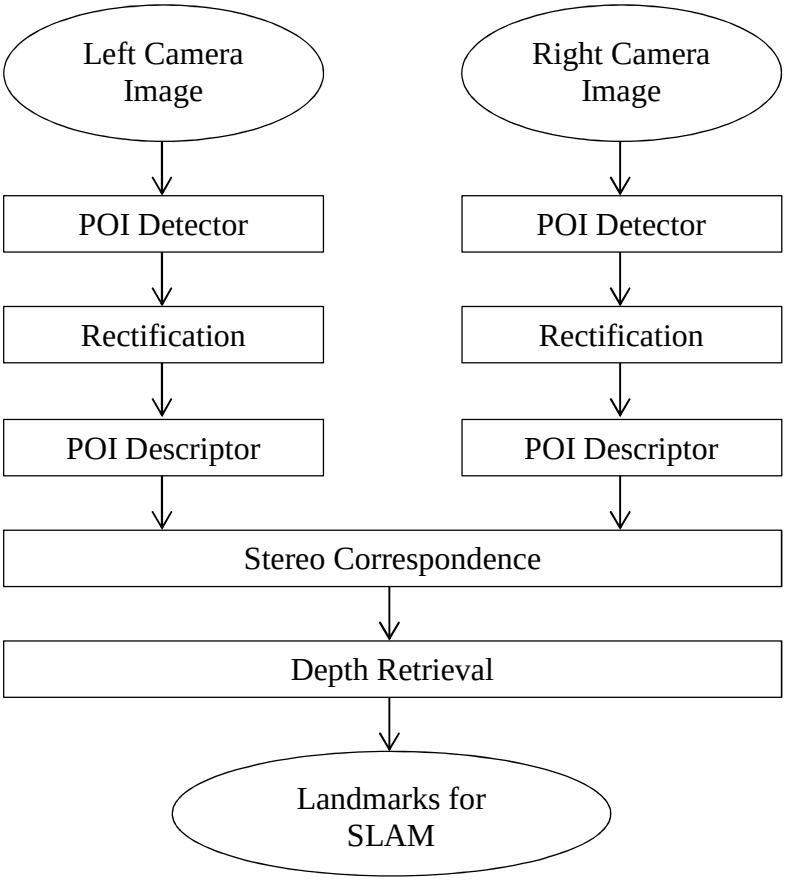


Figure 1: Landmark acquisition system for Visual SLAM with stereo feature-based correspondence. In this work, the POI detector step is implemented in the FPGA, while all other tasks are done in software on the ARM processor, resulting in hybrid architecture. (Author's Figure).

1.1 General Objectives

The objective of this work is to explore an innovative way to solve the correspondence problem between two images, in order to retrieve their 3D coordinates by using the visual spatial relationship between the desired point and its neighbors as a global context. These points are found using a standard POI detector to be used as landmarks for the map construction and localization for SLAM applications, relying on their properties of repeatability and distinctiveness.

Also, this work in concerned with evaluating a hybrid architecture System on a Chip solution that integrates an FPGA fabric with a modern embedded processor as an auxiliary unit for stereo camera image pre-processing and landmark extraction for Visual SLAM, and proposing a solution that runs effectively on these platforms.

1.2 Specific Objectives

In order to verify if visual spatial relationship between POIs can be used to effectively produce relevant information for describing those POIs with the stereo correspondence problem in mind, a prototype of a new descriptor needs to be created, and its speed, number of correct correspondences and error rate compared with existing solutions. If the prototype to be produces useful results, the technique can be incorporated in stereo correspondence systems alone or coupled with existing descriptors.

The proposed system is focused on balancing error reduction, i.e., false matching, with improvements on execution time in order to run the algorithm effectively and efficiently in embedded systems.

The hybrid architecture available on SoCs with ARM processors and integrated FPGA allows for a co-processor to be designed in order to optimize existing software in specific parts parallelization is possible. Thus, a section of the process of obtaining landmarks from POIs for SLAM that is relevant for such optimization needs to be selected for hardware implementation. Finally, the speed of execution and quality of the implementation can be compared with the original software alternative for evaluating the applicability of the optimizations.

A simple SLAM solution also needs to be implemented and tested to determine if the SoC test platform can be used to process the landmarks obtained by the analysis of stereo images in reasonable time.

1.3 Methodology

To ensure that the hardware co-processor designed in this work aids in solving a task that takes significant execution time, a simple system designed in software, comprising all of the building blocks shown in Figure 1, has been implemented in software that runs on the main processor of the SoC. The execution time of each individual block was measured and the results substantiate the choice of developing a hardware co-processor to aid in POI detection (Section 6.1).

After the co-processor synthesis, the execution time was again measured, to compare with the software-based results and determine the speedup gain (Section 6.3.1).

The proposed solution uses POIs initially as in a feature-based correspondence solution (instead of a small window as in area-based), but differs in the fact that spatial relationship between points is considered for finding correspondence. In a sense, it is a hybrid solution. When only the POIs are plotted white in an empty black image, starry night patterns emerge (Figure 2). These patterns can be used to develop a descriptor for matching POIs for a stereo pair of image, and techniques already in use for describing star arrangements in star identification problems (Ho, 2012; Spratling & Mortari, 2009) can then be considered for reapplication for the stereo matching problem.

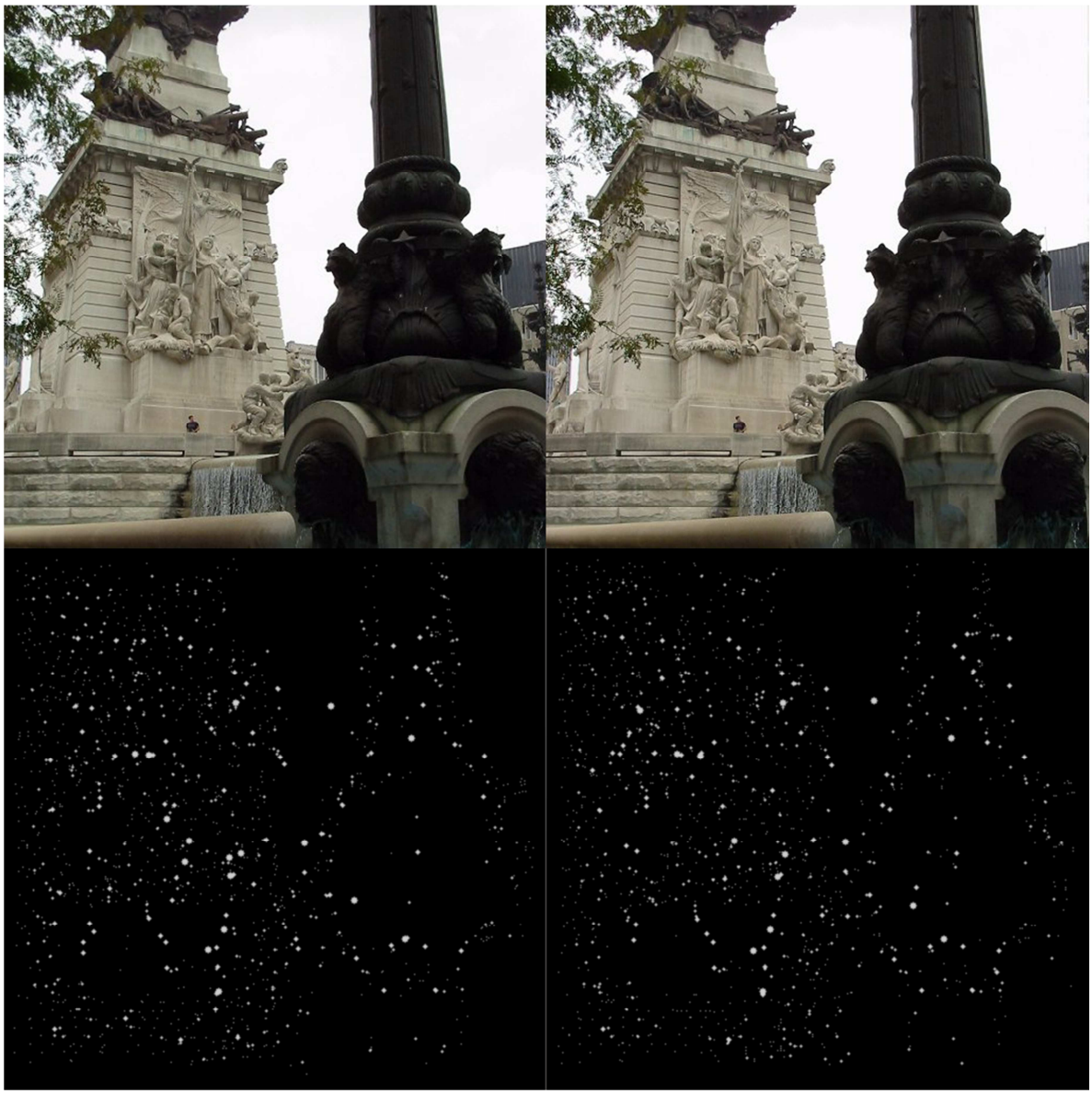


Figure 2: Starry night pattern. POIs detected with SURF are plotted for a pair of image (Callet, 2010) as circles, with radius proportional to the octave of the pyramid where they were detected. (Author's figure).

After the implementation of a descriptor using the proposed technique and its subsequent use for finding correspondence between POIs, its performance is be compared with other POI descriptors such as SIFT (Lowe, 2004), SURF (Bay et al., 2006), BRIEF (Calonder, Lepetit, Strecha, & Fua, 2010), BRISK (Leutenegger, Chli, & Siegwart, 2011), ORB (Rublee, Rabaud, Konolige, & Bradski, 2011) and FREAK (Alahi, Ortiz, & Vandergheynst, 2012) with respect to correct matches, error and speed in the context of SLAM.

2. SIMULTANEOUS LOCALIZATION AND MAPPING (SLAM)

This chapter contextualizes the target application for the work, presenting and defining the SLAM problem. To understand why SLAM solutions are based on probabilistic models, it is important to review the sensors used in SLAM applications because the nature of the noise present in them is the key to the choice of a probabilistic model for SLAM.

While a statistical model, based on the Bayes rule, is shared among the most common solutions (Kalman Filters, Particle Filters and Expectation Maximization) it is interesting to discuss in which point they differ from each other. The EKF-SLAM solution was chosen to be used in this work for two reasons, due to the extensive information available on it for being the most popular tool to solve SLAM and for its implementation being relatively simple when compared to particle filter solutions. EKF-SLAM is then explained in detail by the end of this chapter.

2.1 Definition

As seen in the Introduction, SLAM stands for Simultaneous Localization and Mapping. In SLAM, the robot's task is to incrementally create a map of its surroundings while localizing itself within it (Durrant-Whyte & Bailey, 2006). At any moment, the available information comes from the proprioceptive and exteroceptive sensors, which are always affected by noise in real environments. Proprioceptive sensors measure values internal to the system, while exteroceptive sensors acquire information about the robot's environment (Siegwart et al., 2011).

2.2 Sensors Employed in SLAM

There are many kinds of sensors used for mobile robots. The most typical are optical encoders, compasses, gyroscopes, accelerometers, global positioning system (GPS), ultrasonic sensors, laser rangefinders and vision sensors (CMOS and CCD cameras). Sensors are subject to errors, which can be systematic or randomic. Random errors cannot be

corrected with deterministic approaches, therefore a statistical model is used, and error propagation can be computed integrating data from different kinds of sensors and be used to quantify uncertainty on the robot system. The Gaussian distribution is usually employed for modeling sensors that have no specific models, as it performs well and is mathematically advantageous to other models. (Siegwart et al., 2011) (p. 101-148)

Exteroceptive sensors are the robot's way to get information about the environment, which comes from the landmarks present on it. In the context of mobile robotics and SLAM, landmarks can be defined as objects in the world that can be used as reference for retrieving their relative position to the robot when in its field of view (Siegwart et al., 2011) (p. 344-345). Interesting landmarks are ones that can be easily segmented from the surrounding environment, have unique features that can easily differentiate them from each other and can be perceived from different angles without confusion. They can be purposely placed in the environment (artificial) or objects already present on it (natural). Points of Interest can also be used as landmarks, since they have by definition the characteristics of being repeatable and distinct from each other (Todt, 2005).

Early SLAM works focused on the use of range sensors such as sonar rings and laser (Munguia, Castillo-Toledo, & Grau, 2013). Lasers, particularly, can be used to easily and quickly retrieve spatial information of whole rooms for indoor use, making the mapping task relatively easier. An example of laser reading can be seen in Figure 3.

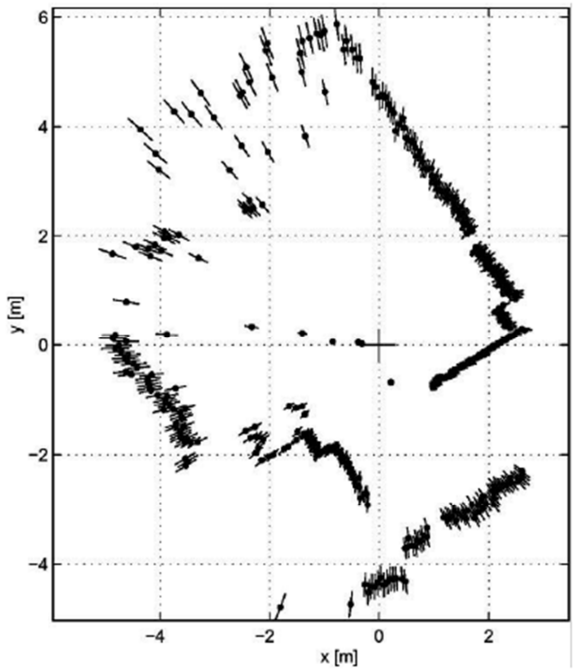


Figure 3 : Typical range images of 2D laser range sensor with a rotation mirror. (Siegwart et al., 2011) (p. 132)

Range sensor's main disadvantages are limited available information for data association, high cost, limited working range, and that some of them are limited to a 2D plane scan only. These issues led to an increased move on recent works towards the use of cameras as a primary sensor, trend that is also motivated due to the fall in prices of camera systems. Besides that, relative to lasers, cameras provide richer information, suitable for data association needed for landmark matching and loop closure on SLAM applications (Munguia et al., 2013). For embedded applications, modern cameras are attractive for being small, lightweight and having low power consumption. For these reasons, they can be easily embedded in mobile robots (Lee & Lee, 2013; Munguia et al., 2013).

These points led to the choice of using cameras as the main sensors for SLAM in this work. It's important to note that while the richer information provided by cameras is useful for the reasons cited above, it's processing require more computing power, which is particularly limited on embedded applications. This increases the importance on researching optimizations for tasks specifically related to this subject.

2.3 Statistic model

Measurement noise is one of the key challenges of robotic mapping and localization. Since noise is statistically dependent, repetitive measurements of the same landmarks are not enough to cancel its effect. Accommodating these systematic correlated errors is the key for a successful SLAM algorithm (Sebastian Thrun, 2002).

A solution for the SLAM problem therefore requires proper modelling of sensors, explicitly including noise. This explains why the three dominating techniques in SLAM (Kalman Filters, Particle Filters and Expectation Maximization) are all probabilistic in nature, derived from the recursive Bayes rule. Probabilistic algorithms explicitly model different sources of noise and their effects in measurements (Aulinas, Petillot, Salvi, & Llado, 2008). The three currently dominating techniques in SLAM are briefly explained further in Section 2.4.

In the context of mobile robotics, robot pose is defined as its position coordinates plus its orientation (angle) to some reference. In SLAM, when the same landmarks are seen from different robot poses (different perspectives), the uncertainty is reduced for the landmark

itself and for the whole map, thus making the probabilistic model converge to a correct estimation of the map, together with a more accurate estimation of the robot pose (Siegwart et al., 2011) (p. 348-351). This reduction in uncertainty is called loop closure, and is illustrated in Figure 4.

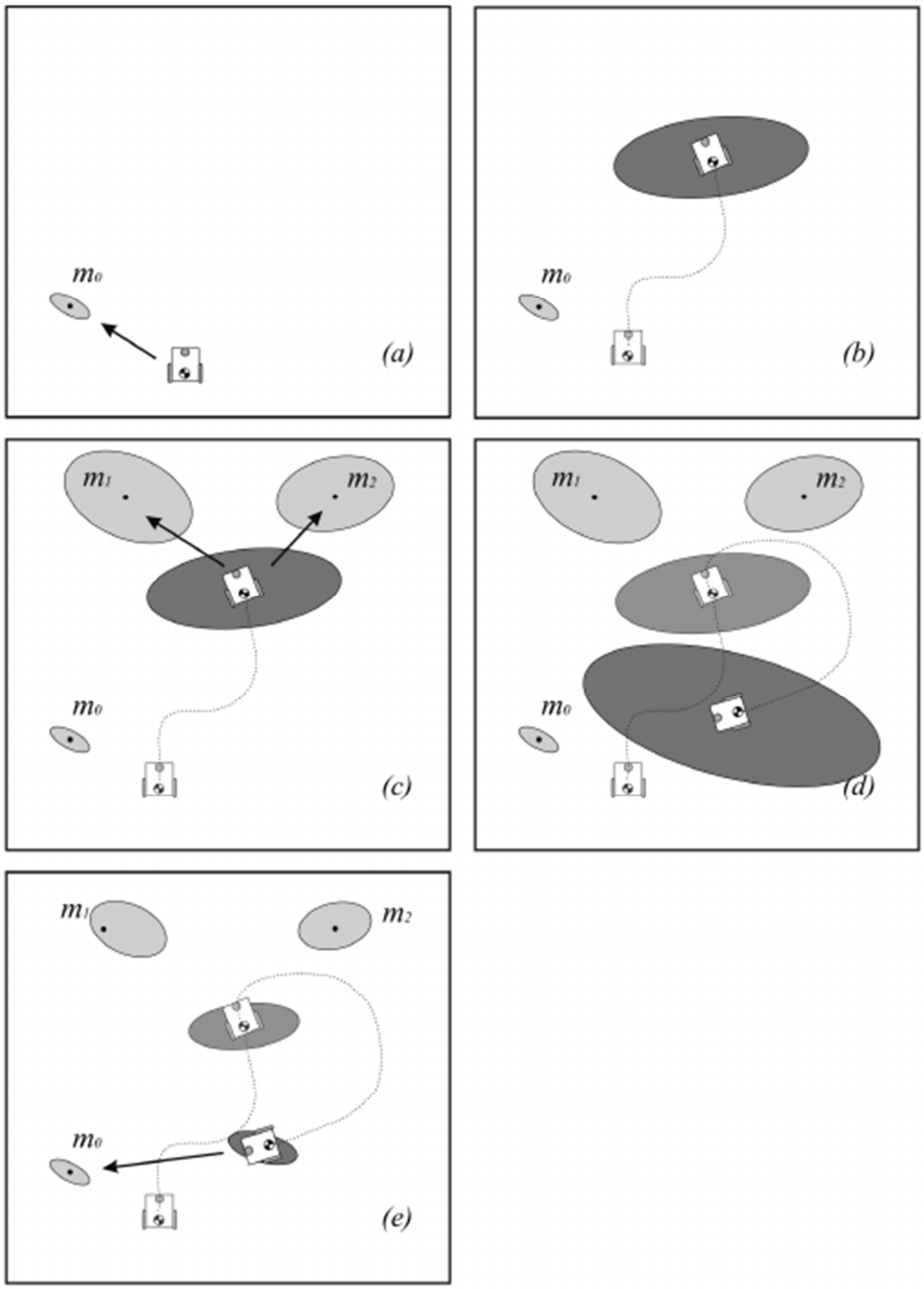


Figure 4 : Illustration of the SLAM problem (Siegwart et al., 2011) (p. 350). In (a), the robot observes the landmark m_0 ; in (b), the robot moves to a new position, and consequently the uncertainty on its position increases; in (c), the robot observes two new landmarks: m_1 and m_2; in (d), the robot moves again, and its position uncertainty increases again; in (e) the landmark m_0 is seen a second time. Its position uncertainty is reduced, and since the position of the landmarks and the robot are all correlated, the uncertainty of the whole map members is reduced. Uncertainties of position in landmarks and robot are represented by the ellipsis encircling them.

2.3.1 Probabilistic model for SLAM

The probabilistic SLAM model is presented in detail in (Durrant-Whyte & Bailey, 2006), and is summarized in this section.

In probabilistic form, the SLAM problem can be expressed with the probability distribution shown in eq. (2.1), which is the joint posterior density of the landmark locations.

$$P\left(R_t, \mathrm{M} \mid Y_{0:t}, U_{0:t}, R_0\right) \tag{2.1}$$

Where:

R_t : state vector describing the pose of the robot at time t

M : a vector with the position of the landmarks: $M = \begin{bmatrix} L_1 & L_2 & \cdots & L_n \end{bmatrix}^T$

L_n : the n landmark

T : transposed

$Y_{0:t}$: the set of all landmark observations from time 0 to time t

$U_{0:t}$: the history of control inputs applied to the robot

R_0 : the robot initial pose

The SLAM solution consists of computing the probability given in eq. (2.1) for all time instants since the beginning of the process until t. For this, a recursive solution is desirable, so that if the probability distribution of $t-1$ is known, the next probability t can be computed directly from it without recalculating for previous instants.

Starting from the previous estimation as in the probability distribution in eq. (2.2), a control u_t is applied to the robot, and an observation y_t is made.

$$P\left(R_{t\text{-}1}, M \mid Y_{0:t\text{-}1}, U_{0:t\text{-}1}\right) \tag{2.2}$$

The joint posterior as in eq. (2.1) is then computed using the Bayes' Theorem. For that, both a motion model and an observation model are needed. In probability notation, they are expressed as the probabilities distributions in eq. (2.3) and eq. (2.4).

$$P\left(R_t \mid R_{t\text{-}1}, u_t\right) \tag{2.3}$$

$$P\left(y_t \mid R_t, M\right) \tag{2.4}$$

The SLAM algorithm is then implemented in a two-step recursive prediction and correction form, as shown in eq. (2.5) and eq. (2.6) respectively, which are called the time-update and measurement update.

$$P\left(R_t, \mathrm{M} \mid Y_{0:t\text{-}1}, U_{0:t}, \mathrm{R}_0\right) = \int P\left(R_t \mid \mathrm{R}_{t\text{-}1}, u_t\right) \times P\left(R_{t\text{-}1}, M \mid Y_{0:t\text{-}1}, U_{0:k}, \mathrm{R}_0\right) dR_{t\text{-}1} \tag{2.5}$$

$$P\left(R_t, M \mid Y_{0:t}, U_{0:t}, \mathrm{R}_0\right) = \frac{P\left(y_t \mid \mathrm{R}_t, M\right) P\left(R_t, M \mid Y_{0:t\text{-}1}, U_{0:k}, \mathrm{R}_0\right)}{P\left(y_t \mid Y_{0:t\text{-}1}, U_{0:t}\right)} \tag{2.6}$$

2.4 Main Solutions to the SLAM Problem

Since the 90s, SLAM approaches are dominantly probabilistic in nature (Sebastian Thrun, 2002). They can be classified as derivatives from Kalman Filters (KF), Particle Filters (PF) and Expectation Maximization (EM). All of these are based on derivations of the recursive Bayes rule. Comprehensive descriptions and comparisons of these approaches can be found in (Aulinas et al., 2008; Sebastian Thrun, 2002) while a brief discussion of the main points, relevant to this work, are summarized in the following sections.

2.4.1 Kalman Filter and its Variants

Kalman filters are Bayes filters that represent posteriors using Gaussian distributions. There are two main variations of Kalman filters in SLAM: the Extended Kalman Filter (EKF) and the Information Filter (IF).

The standard Kalman Filter assumes that observations are linear functions of the state, and that the next state is a linear function of the previous state. Since state transitions and measurements are rarely linear in practice, a linearization through a first order Taylor expansion can be applied to the nonlinear functions, resulting in the EKF (Sebastian Thrun, Burgard, & Fox, 2005) (p. 54-60).

The dual representation of the EKF, the IF, represents belief also as a Gaussian. The main difference is in the way the Gaussians are represented, where instead of using their moment (mean and covariance), they are represented with an information matrix and information vector, which are obtained through a canonical parameterization of the multivariate Gaussian distribution (Sebastian Thrun et al., 2005) (p. 71). On the update step of the IF, an inversion of the information matrix is needed for recovering state, and further inversions are needed in the prediction step. This escalates poorly when the dimension of the map increases with the increase of landmarks, resulting in a poorer performance than EKF (Aulinas et al., 2008).

IFs can be further extended for non-linear state change functions similarly to KF being extended into EKF. EIFs were further improved by representing the information matrix sparsely (S. Thrun et al., 2004), leading to constant time use by the update equations, regardless of the number of landmarks, in what is called the Sparse Extended Information Filter (SEIF). Here, sparcity in the information matrix is enforced to a certain level, reducing the information kept by the filter with the intention of trading it for speed gains as a mean to guarantee the constant update time.

Another KF derivative, the Unscented Kalman Filter (UKF), addresses the possible errors that come from the linearization step applied on EKF. This is done by selecting a minimal set of carefully chosen sample points for the Gaussian Random Variable that when propagated through non-linear systems results in the posterior being captured with more accuracy (Aulinas et al., 2008). UKF performs slightly slower than EKF, but it has the same asymptotic complexity. While it can perform better in some cases where the EKF Taylor approximation leads to more errors in estimation, in many practical applications, the difference is negligible (Sebastian Thrun et al., 2005) (p. 70).

The main disadvantage of EKF is that the computational effort scales quadratically to the number of landmarks on the map on the update step, where both the state vector (representing the means of the pose of the robot and position of the landmarks) and the

covariance matrix (representing the covariance between the map constituents) need to be updated once for every seen landmark. This make long missions impractical because the update step can become unfeasible to be executed in real time when the landmarks become too numerous (Aulinas et al., 2008).

In spite of this, EKF is the most popular tool for state estimation in robotics. Its strength lies in its simplicity and computational efficiency, when comparing with other algorithms such as particle filters, which can require exponential time growth in computation with the size of the state vector (robot pose plus number of landmark positions) (Sebastian Thrun et al., 2005) (p. 61).

The dimensionality problem of EKF has been addressed by employing strategies such as limiting the size of the map to local areas, reducing computational cost to the square of the number of landmarks on this reduced area, and then transferring the information to the overall map in a single step at full SLAM computational cost (Guivant & Nebot, 2001). Another approach uses similar small metric maps taken on equally-spaced basis, but integrates them using a higher level topological graph-based map that includes transformation matrices and uncertainty information between lower-level maps (Schleicher, Bergasa, Ocana, Barea, & Lopez, 2010). Strategies like these can be employed to make EKF applicable to higher scale problems.

2.4.2 Particle Filter based methods

The Particle Filter method is a recursive Bayesian filter that is implemented in Monte Carlo simulations. The Bayesian posterior is represented as a series of samples drawn from the probability distribution, the so called particles. By representing the posterior in this form, the Particle Filter method handles better highly non-linear sensors and noise that cannot be successfully modeled by Gaussian distributions (Aulinas et al., 2008). The number of particles needed for the update step to guarantee convergence of the map can, in worst case scenarios, escalate exponentially with the size of the map (Michael Montemerlo, Thrun, Roller, & Wegbreit, 2003).

There are combinations of Particle Filter with other methods which can get around the dimensionality problem. Examples of these techniques are the FastSLAM (M. Montemerlo et

al., 2002) and it's enhanced version, the FastSLAM 2.0 algorithm (Michael Montemerlo et al., 2003). FastSLAM takes advantage of a characteristic of SLAM problems in that landmark estimates are conditionally independent given the robot's path (Michael Montemerlo et al., 2003). This is the key for the speed of the algorithm. Instead of a map being represented by a covariance matrix where the robot and all landmarks are all dependent on each other (as in EKF), in FastSLAM the map is represented as a set of independent Gaussians, with linear complexity (Durrant-Whyte & Bailey, 2006).

2.4.3 Expectation Maximization methods

Expectation Maximization (EM) is a family of algorithms that was developed in the context of Maximum Likelihood (ML) estimation with latent variables. It is based on the idea that if the robot's path is known (in expectation), determining a map is relatively simple. EM works in two steps: first the posterior for the robot pose is calculated for a given map (the expectation step), then EM calculates the most likely map given the pose expectation (the maximization step). The result is a series of increasingly accurate maps (Sebastian Thrun, 2002).

EM algorithms perform well for determining large scale cyclic maps, as they solve the correspondence problem for loop closure even when odometry errors accumulate over large loops. The downside of the algorithms is that they don't keep a full notion of uncertainty, thus they are not able to produce maps incrementally, and run offline after all sensor data is acquired and accumulated. Map generation can take hours to complete on low-end hardware (Sebastian Thrun, 2002).

2.5 EKF-SLAM

EKF-SLAM represents the motion eq. (2.3) and observation eq. (2.4) models with the equations given in eq. (2.7) and eq. (2.8). In eq. (2.7), the function f represents the model for the kinematics (depending solely on the previous state and the control vector) with added noise n_t, which is modeled by a zero mean, uncorrelated Gaussian distribution. Similarly, in the observation model (2.8), the function h describes the geometry of the

observation, again with added noise v_t modeled by a zero mean uncorrelated Gaussian distribution (Durrant-Whyte & Bailey, 2006).

$$R_t = f\left(R_{t-1}, u_t\right) + n_t \tag{2.7}$$

$$y_t = h\left(R_t, L_n\right) + v_t \tag{2.8}$$

In the motion model, the robot moves from position R_{t-1} to R_t following a control vector u_t. The function f describes this movement, while n_t is the noise intrinsic to the movement.

In the observation model, the robot takes a measurement y_t of the landmarks L when the robot is at pose R_t. The geometric transformation that happens from the global frame coordinates into the actual measurement as it is observed in the robot frame of reference as seen from the pose R_t is represented by the function h. The added sensor noise is represented by v_t.

The global frame of reference has its origin in an arbitrary position in the world, usually the starting pose of the robot, resulting from the map initialization process, and it is the stationary frame of reference used for building the map. In contrast, the robot frame is always relative to the robot pose. Since the built-in sensors of the robot move with it, naturally, the measurements are always in the robot frame. This brings the need for an inverse observation model that can provide the world frame coordinates of the landmarks when they need to be first stored in the map. The function g in eq. (2.9) is responsible for transforming from the local coordinates to the world coordinates.

$$L_n = g\left(R_t, y_t\right) \tag{2.9}$$

It is interesting to note that eq. (2.8) is not always invertible. This is true in the cases where not all the degrees of freedom of perceived landmarks are directly obtained from the transformation (Solà, 2013). A practical example is the case of monocular visual SLAM, where a camera projects the 3D coordinates in a 2D image plane, where it is impossible to reconstruct the 3D scene from the information available on one frame alone, and this is reflected mathematically as a non-invertible function.

2.5.1 Map definition in the context of EKF-SLAM

In EKF-SLAM, the map is comprised of a large column vector, stacking the mean of robot pose and landmark position states, modeled by a Gaussian variable. Accompanying the state vector is a covariance matrix, representing the uncertainty referent to the members of the map, and the correlation between them. The Extended Kalman Filter is responsible for predicting the next state, given the control input to the robot and correcting it using the information about perceived landmarks coming from the available sensors.

Thus, the internal map representation follows the form in eq. (2.10).

$$\bar{x} = \begin{bmatrix} \bar{R} \\ \bar{M} \end{bmatrix} = \begin{bmatrix} \bar{R} \\ \bar{L}_1 \\ \vdots \\ \bar{L}_n \end{bmatrix} \qquad P = \begin{bmatrix} P_{RR} & P_{RM} \\ P_{MR} & P_{MM} \end{bmatrix} = \begin{bmatrix} P_{RR} & P_{RL_1} & \cdots & P_{RL_n} \\ P_{L_1R} & P_{L_1L_1} & \cdots & P_{L_1L_n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{L_nR} & P_{L_nL_1} & \cdots & P_{L_nL_n} \end{bmatrix} \tag{2.10}$$

Where the vertical vector $\bar{x}$ stores the mean of the robot pose $\bar{R}$ and the mean of landmark positions $\bar{L}_1 \cdots \bar{L}_n$. The covariance matrix P stores the covariance between the members of the map, keeping the uncertainty of the system.

The goal of EKF-SLAM is to keep this map up to date at all times. To achieve this, there are two main steps: prediction and correction. The prediction step occurs whenever the robot moves, while the correction step happens when an observation is made. There are also auxiliary steps: map initialization and landmark initialization. The first is done at the beginning to set the initial values to the map. The latter is made when a new landmark needs to be added to the map. This gives the system the ability to expand its representation online, turning the EKF into a filter of state of dynamic size (Solà, 2013). There are also optional steps that add robustness to an EKF-SLAM system, such as removing landmarks that are no longer perceived, thus recycling the space available on memory to store the map.

2.5.2 Map initialization

In the beginning, there is usually no previous frame of reference for the map being created. In this case, it is usual procedure to consider the robot pose as the origin of the map. If this supposition is made, and before any movement of the robot, the certainty of its pose is absolute, thus the covariance matrix that represents uncertainty is initialized with zeroes, representing absolutely no uncertainty. Here the map consists of only the robot pose, because there are yet no perceived landmarks to populate it, as shown in eq. (2.11).

$$\bar{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.11}$$

2.5.3 Robot motion: the prediction step

Following the model of robot motion presented in eq. (2.7), the robot knows both the prior estimate of its pose and the control vector that was applied to it. The true noise that invariably exists in every movement is not known, but its distribution is internally modeled by a Gaussian. The prediction step thus updates the mean of the robot pose $\bar{R}$ by considering the noise's mean value as zero in eq. (2.12), and increases the uncertainty of the robot pose alone based on the noise model. Landmarks are not supposed to move when the robot does, so no update on their representation is required at this point. Mathematically, the increase in uncertainty is calculated through the error propagation law according to eq. (2.13) (Siegwart et al., 2011) (p. 113-115), with the uncertainty of the map modeled internally with the covariance matrix P.

$$\bar{R} \leftarrow f\left(\bar{R}, u_t\right) + 0 \tag{2.12}$$

$$P \leftarrow F_R P F_R^t + F_n N F_n^t \tag{2.13}$$

Where:

F_R : The Jacobian of the function f with respect to R: $F_R = \dfrac{\partial f\left(\bar{R}, u\right)}{\partial R}$

F_n : The Jacobian of the function f with respect to n : $F_n = \frac{\partial f\left(\overline{R}, u\right)}{\partial n}$

N : Covariance matrix of perturbation n .

The superscript t denotes a matrix transposition.

In many cases, it can also be said that the second Jacobian F_n is derived with respect to the control vector u , since the noise could be considered as a disturbance to the control (Solà, 2013).

2.5.4 Landmark observation: the update step

The update step is performed whenever an observation is made. Classically, it is represented by the set of five equations (2.14) to (2.18).

$$\overline{z} = y - h\left(\overline{R}, L_n\right) \tag{2.14}$$

$$Z = H_R P H_R^t + V \tag{2.15}$$

$$K = P H_R^t Z^{-1} \tag{2.16}$$

$$\overline{x} \leftarrow \overline{x} + K\overline{z} \tag{2.17}$$

$$P \leftarrow P - KZK^t \tag{2.18}$$

The eq. (2.14) represents what is called the innovation $\overline{z}$, with the associated uncertainty Z represented by eq. (2.15). The mean of the innovation is the difference between the noisy measurement y of a landmark and the prediction of what the measurement would be based on previous information about the landmark present in the map, following the observation model. The covariance is calculated by the equation (2.15), where V represents the covariance matrix of the measurement noise and H_R is the Jacobian of the function h with respect to the robot pose R .

The eq. (2.16) represents the Kalman Gain, K , which determines how much the innovation is going to be weighted in relation to the past information stored on the map. The

innovation and Kalman Gain calculations collect the information needed for the update step, which is performed for the mean value $\bar{x}$ in eq. (2.17), and its associated uncertainty, P, in eq. (2.18).

It is important to note that when a movement update is performed in eq. (2.12), the uncertainty about the robot pose increases, as shown in Eq. (2.13). However, in eq. (2.18) the minus signal represents a reduction in uncertainty for the whole map, meaning that the uncertainties of both the robot pose and of all the landmarks are reduced in the sequence of the algorithm. An update step is performed once for every landmark that is observed, but since in SLAM the error of measurement is correlated for all landmarks and also depends on the robot pose, the uncertainty of the whole map is reduced. This particular point is the reason why a map being constructed by EKF-SLAM can incrementally converge into a more correct representation with each iteration step of the algorithm, as previously shown in Figure 4.

The last EKF-SLAM, eq. (2.18), defines the complexity of $O(n^2)$ to the algorithm. Since this operation is performed once for every perceived landmark, the algorithm complexity can be written as $O(k \cdot n^2)$, where k is the number of landmarks observed simultaneously.

Eq. (2.14) also assumes that landmark correspondence is known, that is, an observed landmark has a specific map entry where its information was previously stored in the map, and this position is known for each landmark. When landmark correspondence is unknown, extra steps are needed before the innovation can be calculated to infer the correspondence.

2.5.5 Landmark Initialization

Landmark initialization happens when new landmarks are discovered by the robot, resulting in an increase on the size of the state vector $\bar{x}$ for every new landmark. The process is relatively easy to perform as long as the observation function h from eq. (2.8) is invertible generating the g function in eq. (2.9). The procedure works by calculating the landmark's position mean L_n and the Jacobians G with respect to the robot pose R and the measurement y_n as shown in eq. (2.19).

$$\begin{aligned} \overline{L}_n &= g\left(\overline{R}, y_n\right) \\ G_R &= \frac{\partial g\left(\overline{R}, y_n\right)}{\partial R} \\ G_{y_n} &= \frac{\partial g\left(\overline{R}, y_n\right)}{\partial y_n} \end{aligned} \tag{2.19}$$

Then, the landmark's covariance P_{LL} and cross-covariance with the rest of the map P_{Lx} is calculated as shown in eq. (2.20).

$$\begin{aligned} P_{LL} &= G_R P_{RR} G_R^T + G_{y_n} R G_{y_n}^T \\ P_{Lx} &= G_R P_{Rx} = G_R \left[P_{RR} P_{RM} \right] \end{aligned} \tag{2.20}$$

P_{RR} and P_{RM} are slices from the covariance matrix P previously defined on (2.10).

The results from eq. (2.20) are then appended to the state mean and covariance matrix as shown in eq. (2.21).

$$\begin{aligned} \overline{x} &\leftarrow \begin{bmatrix} \overline{x} \\ \overline{L}_n \end{bmatrix} \\ P &\leftarrow \begin{bmatrix} P & P_{Lx}^T \\ P_{Lx} & P_{LL} \end{bmatrix} \end{aligned} \tag{2.21}$$

2.6 Discussion

This is the generic form of the EKF-SLAM algorithm. By using the appropriate sensor models in the place of generic functions, it is possible to project a specific EKF-SLAM system with many kinds of input sensors in two or three dimensional maps. As presented on Section 2.5, for an EKF-SLAM system, the direct and reverse observation models are needed for the chosen sensors. These transformations will be performed in an embedded system, which is studied in Chapter 3. The models themselves are presented in Chapter 4, with their practical implementations shown in Chapter 5.

3. EMBEDDED SYSTEM

Departing from the general context of SLAM presented in Chapter 2, here the specific details for the hardware platform that was chosen to implement the proposed system are discussed. The SoC platform that integrates a multi-core processor with an FPGA fabric is introduced, including the development kit that was used.

The communication protocol standard between the processor and custom FPGA peripherals is presented. Finally, details about synthesizing a circuit written in hardware description language are shown, along with the specific memory architecture of Xilinx's FPGAs.

3.1 System on a Chip Platforms with Embedded FPGA

FPGA stands for Field-Programmable Gate Array. It consists of a 2D array of generic logic cells and programmable switches. Figure 5 depicts a conceptual structure of an FPGA, where a matrix of special cells are configured to perform simple tasks, and by combining them selectively with the aid of the switches, lead to a custom digital circuit design. These cells are called CLBs (Configurable Logic Blocks) by Xilinx, or LABs (Logic Array Blocks) by Altera (Chu, 2008b; Pedroni, 2010).

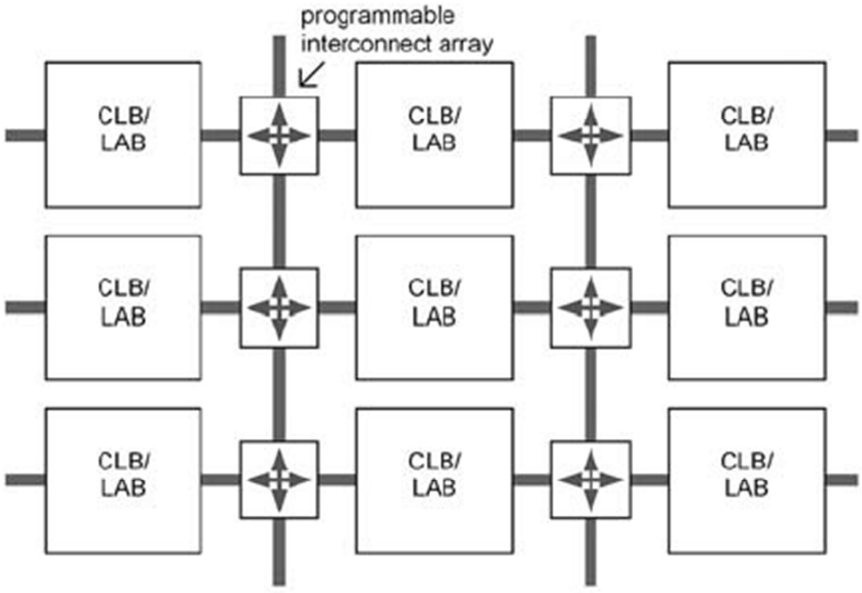


Figure 5: Conceptual structure of an FPGA device (Pedroni, 2010).

In the context of programmable devices, digital circuits designed to solve a particular problem or perform a particular task are called Intellectual Properties (IPs).

It is very common for circuit designs to feature embedded processors to aid in specific tasks, and part of FPGA logic fabric can be configured to work as such. There are commercial IP solutions to relieve the hardware engineer from designing the processors, adapted and optimized for each FPGA manufacturer. When a processor is implemented within the FPGA logic, they are called soft processor cores. Examples are Xilinx MicroBlaze processor core (Xilinx Inc., 2014a) and Altera Nios II embedded processor (Altera Corp., 2014a).

There are implementations in which the embedded microprocessor speed becomes the main factor in the speed of the whole system or there is intention to reallocate the logic cells consumed by the soft core processor for performing other functions, when using an external processor can aid. There is yet another solution, the use of a System on a Chip (SoC) platform that integrate the processors and FPGA fabric within a single chip. These processors are called hard processor cores.

SoC platforms can reduce Printed Circuit Board (PCB) space needed to integrate an otherwise external microprocessor with the FPGA. They can also benefit from high speed interfaces between the nearer processor and FPGA. Tighter integration also results in reduced power consumption.

Last generation SoCs from Xilinx and Altera provide high performance dual-core ARM processors integrated with FPGA fabric. Examples are Xilinx Zynq-7000 (Xilinx Inc., 2014c) and Altera Arria V, Arria 10 and Stratix 10 SoC series (Altera Corp., 2014b).

SoC solutions provide a platform for the project of highly specialized hardware co-processors on the FPGA fabric that exploit parallelization paradigms to provide hardware acceleration for computation of specific tasks, while highly sequential code can still run on the regular processors which are better fitted for these tasks. The possibility of running a full-featured GNU/Linux distribution on the main processor allows for programming in many different computer languages, while retaining the extensive and expansible hardware driver support on the Linux kernel.

By running the Linux kernel, the use of end-user consumer hardware peripherals are made easier with SoC platforms, being connected through standard personal computer I/O

ports instead of FPGA specific expansion peripheral modules. This can potentially reduce system cost by including external hardware that are produced in larger quantities.

3.2 ZedBoard

ZedBoard is a development kit for creating or evaluating designs for the last generation SoC solutions that integrate FPGA fabric with a dual-core ARM processor in a hybrid architecture. The board is manufactured by Avnet and Digilent. Its heart is the XC7Z020 version of the Zinq-7000 SoC, which features a dual-core ARM Cortex A9 processor running at 866MHz and an Artix-7 FPGA equivalent in the same chip. There is 512MB of DDR3 RAM available to the processor and logic, and storage can be done either in the integrated 256MB flash or through a SD card slot (Avnet Inc., 2014; Xilinx Inc., 2013b). The board is available in the VRI laboratory, and can be seen in Figure 6.

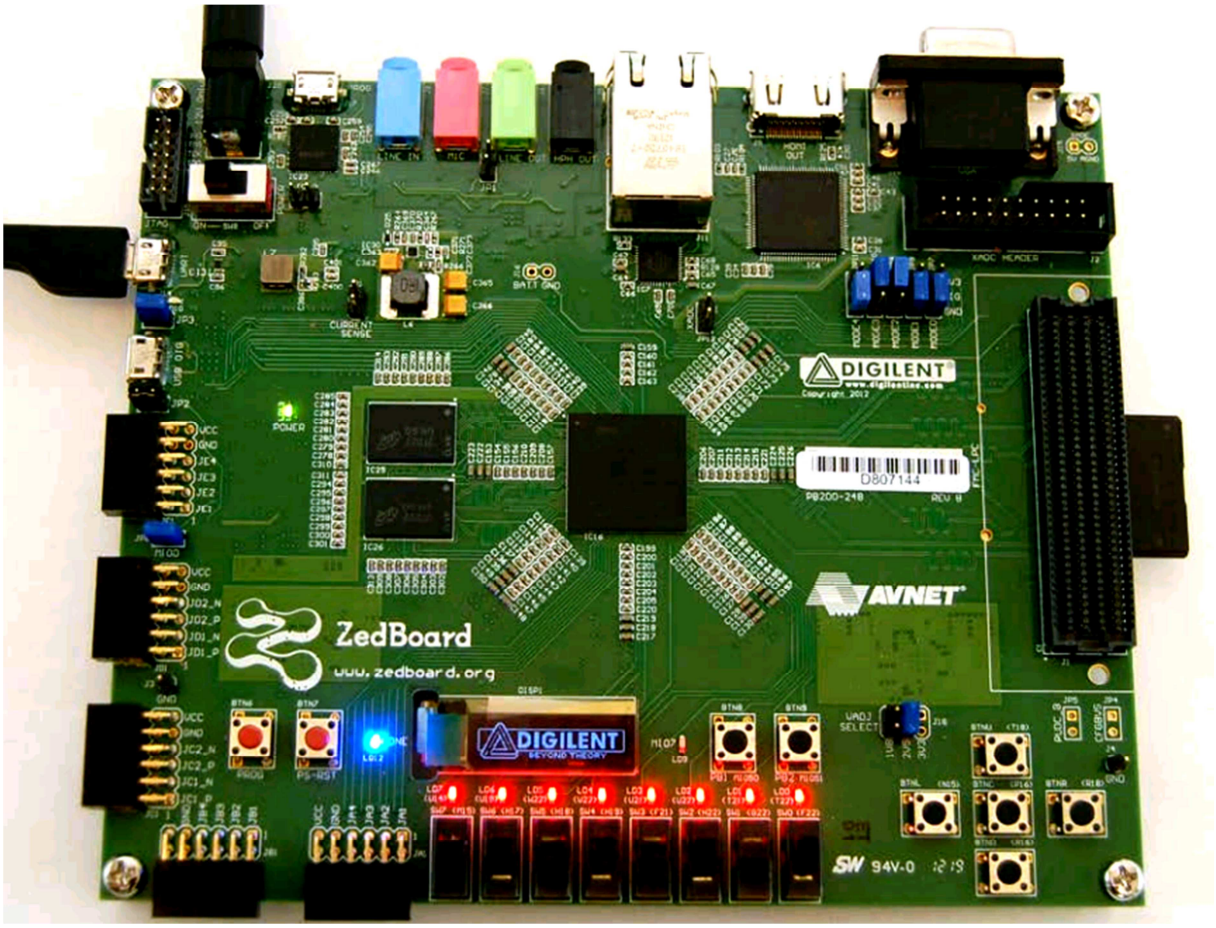


Figure 6: ZedBoard (Avnet Inc., 2014).

There are many input/output ports available on the board, such as gigabit Ethernet, USB-UART and USB-OTG 2.0. For video output, VGA and HDMI ports are present. (Avnet Inc., 2014). Audio I/O is done through the ADAU1761 audio codec with integrated DSP (Analog Devices Inc., 2010).

The FPGA part of the SoC is equivalent to a standalone Xilinx Artix-7 FPGA. XC7Z020 has 6,650 Configurable Logic Blocks (CLB), each consisting of two slices, totaling 13,300 slices. Each slice is contains 8 Flip-Flops (FF) and 4 6-input Look-Up Tables (LUT), plus multiplexers and arithmetic carry logic (Xilinx Inc., 2013a, 2013b).

The common metric for comparing Xilinx's FPGAs is the number of logic cells. Classically, a logic cell is comprised of a 4-input LUT and a flip-flop. Xilinx series 7 FPGAs (including Zynq-7000) have LUTs with more inputs, abundant FFs and latches, additional carry logic and about a third of its slices can be configured to create distributed RAM or shift registers. For this reason, the ratio between 6-input LUTs and classic logic cells is calculated to be 1.6:1 (Xilinx Inc., 2013a). Since the XC7Z020 has 13,300 slices, with 4 6-input Look-up tables each, the total of 6-input Look-up tables is 53,200. Adjusting this number by the 1.6:1 ratio, it can be said to have the equivalent of 85,120 classic logic cells, as advertised (Xilinx Inc., 2013b).

3.3 Intellectual Property (IP) Design

Hardware Description Languages (HDL) associated to FPGA devices are technologies that allow for quick development and simulation of sophisticated digital circuits. There are two widely used languages for projecting digital circuits: VHDL (IEEE, 2007) and Verilog (IEEE, 2006). Despite drastic differences between them, both are IEEE standards and equally capable when used for hardware synthesis and simulation (Chu, 2008b).

HDL code is used for describing the behavior or structure of a digital circuit, that can be later simulated or synthesized into a compliant circuit in CPLDs (Complex Programmable Logic Device), FPGAs or mask generation for ASIC (Application-Specific Integrated Circuit) (Pedroni, 2010).

HDL languages can be used for purposes other than synthesis. Many HDL constructs are meant for circuit modeling, and if used naively for circuit synthesis, can cause unintended

and/or unnecessarily complex hardware implementations. In extreme cases, some constructions can even result in non-synthesizable hardware (Chu, 2008b). Therefore, the use of HDL languages must be taken with caution when the objective is not modeling and simulation of circuits, but synthesizing actual working circuit for practical FPGA or ASIC applications.

In order to ensure that HDL code is correctly interpreted and infers the appropriate hardware when being synthesized by the Xilinx Synthesis Technology (XST), a user guide is provided by Xilinx with examples that are guaranteed to produce the desired optimized hardware they depict (Xilinx Inc., 2009). These guidelines are used whenever possible in this work as the construct units for the proposed IP.

XST constructs are expanded in (Chu, 2008a, 2008b), providing examples of both VHDL and Verilog constructs specialized for the Spartan-3 platform. While ZedBoard uses a Zynq-7000 SoC that belongs to the latest series 7 of Xilinx's FPGAs and Spartan-3 belongs to the third series, these circuits can be adapted and ported for newer FPGAs. Xilinx also provides a XST user guide with guidelines exclusive to Virtex-6, Startan-6 and 7 series devices (Xilinx Inc., 2012). A comprehensive textbook of VHDL constructs which explicitly differentiate between code intended for synthesis and simulation was written by Pedroni (Pedroni, 2010), and serves as an adequate language reference. It discusses the synthesized hardware from VHDL example constructs, clarifying limitations in what can produce reliable hardware for FPGAs while keeping code in industry standard form.

Strategies for designing parallel co-processors for image processing in FPGAs are discussed in (Bailey, 2011), which further expands on the viability of solving image processing problems within FPGA hardware platforms, considering hardware limitations with respect to the dimensionality of what can fit into a current FPGA chip.

Since both VHDL and Verilog can be used for circuit synthesis in this work, language choice is merely a matter of preference. VHDL was chosen for describing designed circuits since the language is widely used in the institution and by laboratory colleagues.

3.4 AXI - Advanced eXtensible Interface

The dual-core ARM microprocessor in Zynq-7000 communicates with the FPGA fabric through the Advanced eXtensible Interface (AXI) protocol. AXI is part of the ARM AMBA (Advanced Microcontroller Bus Architecture), a family of microcontroller buses. AMBA v.4.0 specifies the second version of AXI, the AXI4 protocol (Xilinx Inc., 2011).

For proper communication between the ARM processor and the designed IP, the later should be designed as a slave AXI4 IP device, while the processor acts as the master in communication (Digilent Inc., 2013). The slave device can be considered as a hardware peripheral to the processor. Figure 7 shows a schematic of the Zynq-7000 with AXI connections between the processor and programmable logic, through the central interconnect (Red and blue connections for 32 and 64-bits, respectively).

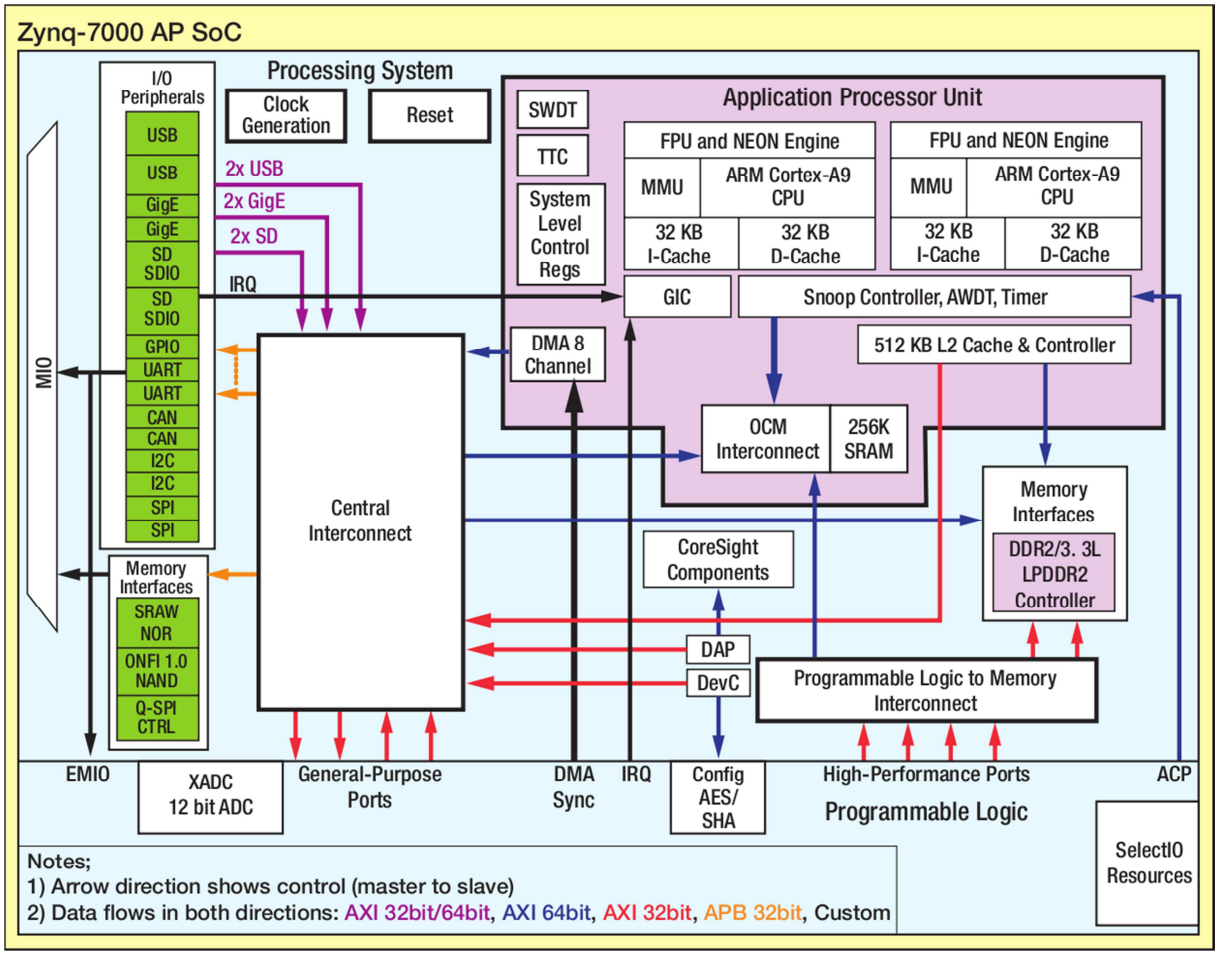


Figure 7: The Xilinx Zynq-7000 Extensible Processing Platform. AXI connections can be seen between the Application Processor Unit and the Central Interconnect, which is then connected to the Programmable Logic (through General-Purpose Ports) (Taylor, 2015).

There are three kinds of AXI4 interfaces specified in the standard: AXI4, AXI4-lite and AXI4-stream. The first is the full-featured high performance, memory mapped interface. Its second version has a simplified communication structure, with constructs acting analogous to registers, which are also memory mapped. The third version is designed for high speed streaming data communication, where data is continuously sent and received in a serial stream (Xilinx Inc., 2011).

ZedBoard's manufacturer, Digilent, provides a detailed step by step implementation of both a hardware template for AXI4-lite IP and a correspondent Linux kernel driver. A software example that uses the driver to communicate with the example hardware is also provided, which is intended for the GNU/Linux distribution running in the dual-core ARM processor (Digilent Inc., 2013).

The availability of documentation and simplicity of designing AXI4-lite hardware were the reasons of choosing the provided template as the initial foundation for IP design in this work.

3.5 FPGA Memory Architecture

There are two types of internal memory in Xilinx FPGAs: distributed RAM and block RAM. Distributed RAM is nothing more than FPGA logic cells being configured to function as a memory. This uses the logic cells' look-up tables (LUTs), configured as a synchronous RAM module. Multiple LUTs can be cascaded to form a wider or deeper memory module. The obvious downside of using distributed RAM is that it competes for resources with normal logic, so its use should be restricted to tasks that require relatively small storage (Chu, 2008b).

Since processing tasks require memory to a certain degree, special memory modules are embedded in Xilinx FPGAs, separately from logic cells, called block RAM or BRAM in short. These memory modules can be configured with varied dimensions, thus being easily adapted to circuit requirements (Chu, 2008b).

The Zynq-7000 SoC model XC7Z020 that is used in ZedBoard has 140 blocks of RAM distributed along the FPGA. Each block can store 36Kb of data, with a total of 560KB of data. This assumes that an extra bit of parity is used for every 8-bit of data. Each block

memory can be read from two independent output ports. For applications that need extra memory, external RAM is needed. ZedBoard has 512MB of external DDR3 memory that is shared between the ARM processor and the FPGA (Avnet Inc., 2014; Xilinx Inc., 2013b).

Block memory can be instantiated in a variety of forms. Each 36Kb block can be divided in two independent blocks or used as a single entity. Supported configurations can then take the forms described in Table 1.

Table 1: Memory layout configuration, adapted from (Xilinx Inc., 2014b)

36Kb block	2x18Kb block
32K x 1	16K x 1
16K x 2	8K x 2
8K x 4	4K x 4
4K x 9	2K x 9
2K x 18	1K x 18
1K x 36	512 x 36
512 x 72	-

A block can also be cascaded with an adjacent one, forming a special 64x1 construct (Xilinx Inc., 2014b).

There are three ways to incorporate block RAM modules into HDL design: by instantiation, through Xilinx's Core Generator program and using the Behavioral HDL inference template presented in the XST User Guide (Xilinx Inc., 2009).

Instantiation is done by using HDL templates featuring specific entity names. These templates lack an architecture body, since a real RAM block is being instantiated instead of being synthesized. They are available directly in the Xilinx ISE (Integrated Software Environment) tool (Chu, 2008b).

Xilinx's Core Generator program automates the instantiation process through a graphical user interface. Since the program creates its own files that lie outside of the HDL framework, compatibility problems are to be expected when using 3rd party tools that are no part of Xilinx developing environment (Chu, 2008b).

The closest solution to a portable solution is using behavioral templates as suggested in (Xilinx Inc., 2009). Behavioral templates provide both an entity and architecture body in VHDL. When using instantiation templates, on the other hand, only the entity body is needed. The extra architecture body present in behavioral templates describes a memory that is analogous in function to the real block RAMs, and using them depends on the ability of Xilinx Synthesis Tools for recognizing the hardware engineer's intention of using real BRAMs instead of wanting to construct a memory out of FGPA's logic cells. The clear advantage is that these templates, having a fully functional architecture description in VHDL, can be used for simulation purposes even outside Xilinx's environment tools. This is desirable for preserving the freedom to use simulation tools outside of Xilinx's environment. Since there is only assurance of correct recognition of templates within XST, porting the code to FPGAs from different manufacturers still requires work (Chu, 2008b).

3.6 Chapter Discussion

Considering the memory architecture of the FPGA and the guidelines for correct implementation in HDL, it is possible to write a custom intellectual property peripheral intended to accelerate the execution of algorithms. This peripheral is connected through the AXI interface to the ARM processor, which can execute high level code supported by GNU/Linux. In essence, this allows the project of dedicated logic to speed up specific parts of the algorithm by implementing them in circuit level on the FPGA fabric. Thus, it is not necessary to write whole algorithms in low level HDL language, along with drivers to interface with external hardware which are already supported by the Linux kernel and can be executed on the embedded processor, while keeping everything in a single chip in this hybrid solution.

4. THE VISUAL SLAM SYSTEM

This chapter takes a step forward on the concepts presented in Chapter 2, introducing the specific challenges of using cameras as the main sensors for SLAM. Initially, the standard EKF-SLAM system is briefly reintroduced, then its observation model and reverse observation model are rethought taking the camera sensor particularities into account. This leads to the construction of a camera model, then finding the camera parameters through calibration.

A stereo camera configuration is needed in order to recover the depth perception intrinsically lost when the image is projected into two dimensions on the sensor plane. Retrieving the depth information requires that the correspondence between the two image's contents are known, leading to a discussion on the main approaches for solving the correspondence problem in order to provide subsidies to the new proposed solution presented in Chapter 5.

4.1 Particularities of Visual SLAM Systems

All EKF-SLAM systems, as described in Chapter 2, share some essential steps: map initialization, robot motion update, sensor observation update and landmark initialization. Figure 8 illustrates how these general steps work together in an EKF-SLAM system.

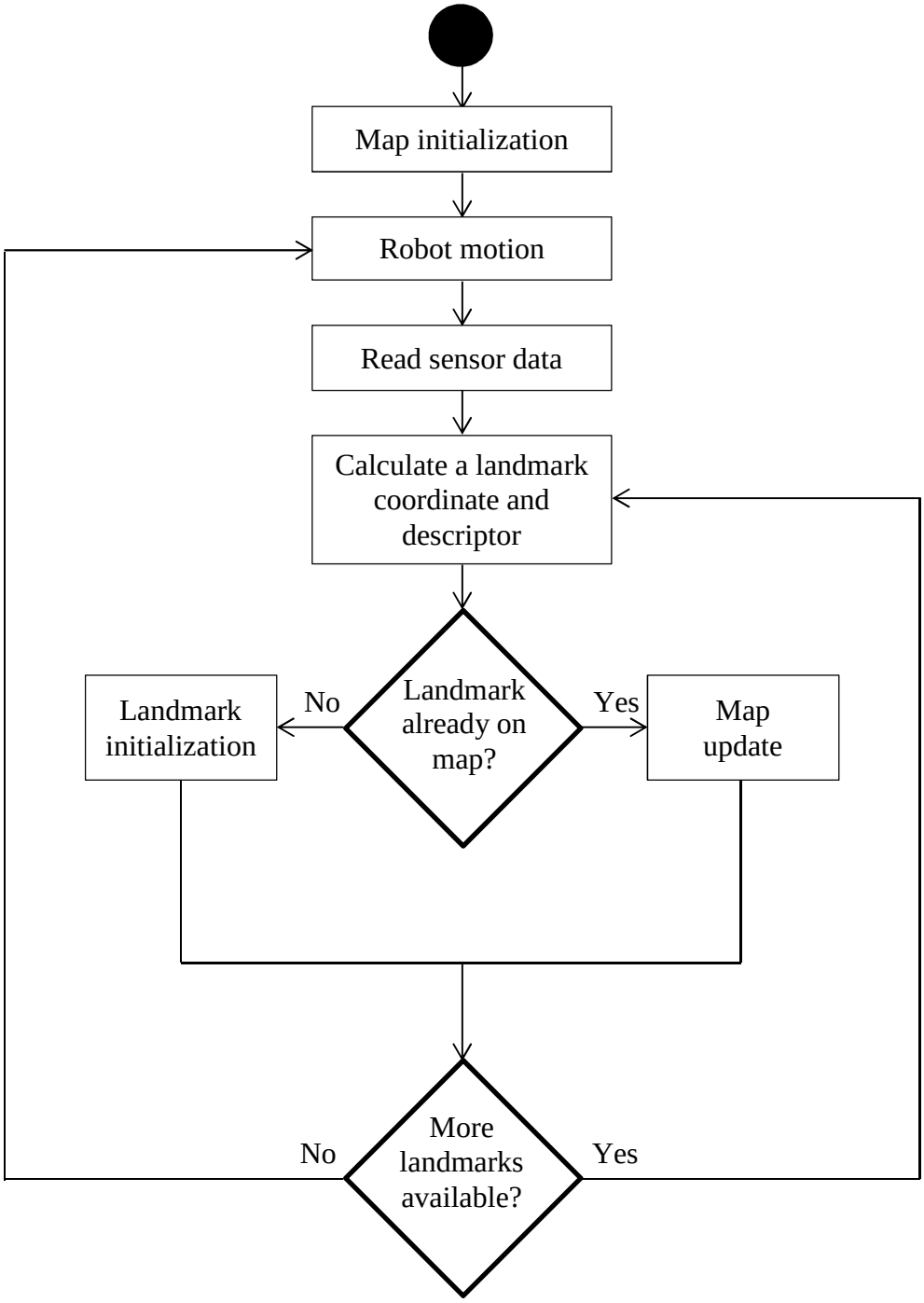


Figure 8: Typical EKF-SLAM system, as described previously on section 2.5. (Author's figure).

Using a stereo camera setup for acquiring landmarks for SLAM requires some extra processing to recover the 3D depth information from the image pair that was captured simultaneously at a given time. When a point is captured from a camera, its 3D coordinates are projected into 2D pixel coordinates in the image.

When a landmark is present in the map, it is necessary to emulate the transformations that happen in the image captured by the camera sensor in order to predict where a landmark

will be seen by the robot before applying the update steps. When initializing new landmarks on the map it is required to do the opposite: recover the 3D information that was lost when projecting the image into the camera sensor, to position it properly on the map.

These requirements imply the use of both a camera model that emulates the 3D to 2D transformations that happen when using cameras, and a reverse camera model to undo these transformations.

4.2 The Camera Model

The general idea behind a camera model is to predict where a point being seen by the camera in the world in some arbitrary 3D reference coordinate system appears in the image plane in 2D discrete pixel coordinates.

In contrast, a reverse camera model would do the opposite: recover the 3D coordinates in the real world given an image with 2D pixel coordinates. Unfortunately this becomes impossible when using a single image, due to the nature of scene projection in a 2D plane, where information is lost.

Getting an invertible observation model then requires more information than that provided by a single image. An invertible function can be obtained by using more than one image taken at different poses, either at different times or simultaneously by means of using multiple cameras. A stereo camera setup is one particular case that explores this ability. Two calibrated cameras with known relative poses can be used to obtain enough information for inverting the camera observation function.

The extra information comes from the difference between captured images, called disparity. Depth information can be recovered up to a certain distance, proportional to the distance between the cameras. In general, this is enough for mapping the robot surroundings, and presents no problem on performing obstacle avoidance, as relevant obstacles are always close to the robot (Siegwart et al., 2011).

For this work, the camera model used in the Open Source Computer Vision (OpenCV) library is adopted (OpenCV, 2014). OpenCV camera model is based on the

pinhole camera model (Hartley & Zisserman, 2003) (p. 154). Figure 9 shows the pinhole camera geometry that is used as the basis to the model.

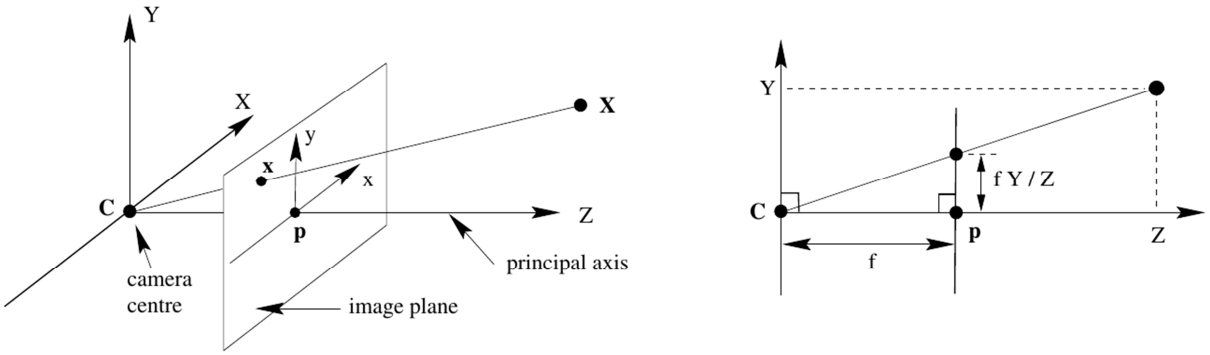


Figure 9: Pinhole camera geometry (Hartley & Zisserman, 2003) (p. 154). C is the camera center, p is the principal point (image center) and f is the focus distance from C to the image plane.

The pinhole camera model is shown in Equation (4.1), and is expanded to incorporate cameras that use lenses to compensate for lens distortion, and that is done separately by Equation (4.2) and (4.3), respectively, for radial distortions and tangential distortions. The parameters are defined in the following paragraphs.

$$\begin{bmatrix} w\cdot x \\ w\cdot y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{4.1}$$

$$\begin{aligned} x_{corrected} &= x\left(1+k_1r^2+k_2r^4+k_3r^6\right) \\ y_{corrected} &= y\left(1+k_1r^2+k_2r^4+k_3r^6\right) \end{aligned} \tag{4.2}$$

$$\begin{aligned} x_{corrected} &= x+\left[2p_1xy+p_2\left(r^2+2x^2\right)\right] \\ y_{corrected} &= y+\left[p_1\left(r^2+2y^2\right)+2p_2xy\right] \end{aligned} \tag{4.3}$$

This model assumes that objects in the world are given coordinates relative to the central point in the image plane, while image coordinates start at the top left corner in the image (Figure 10). The points in the real world proportionally change their size in the projection plane relative to the focus distance f_x and f_y, and then are translated into this new coordinate system by adding the center point coordinates c_x and c_y from Equation (4.1).

Pixel coordinates are obtained by normalizing the first two terms (wx and wy) of the column vector from the left side of the Equation (4.1) by the third term w (here equal to the Z coordinate, resulting in far objects appearing smaller in the projected image). f_x and f_y are the same if the camera sensor aspect ratio is 1:1 (being geometrically a square). Rectangular sensors will result in different values of effective focus in each direction.

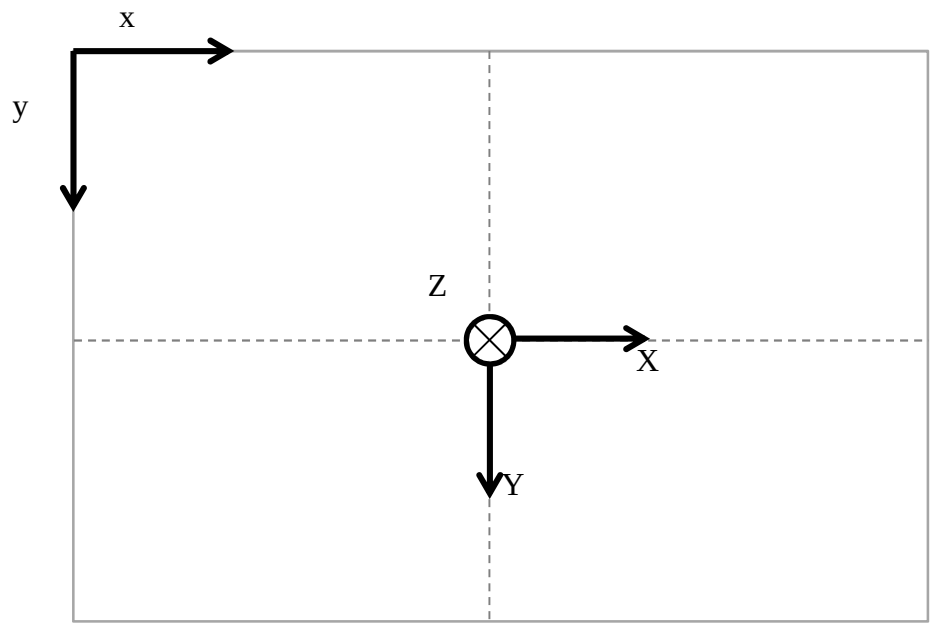


Figure 10: World and image plane coordinates.

Radial distortion, corrected by Equation (4.2), is a type of lens distortion that is proportional to the radius distance r from the image center. In practice, they appear as barrel distortion (also known as fish eye distortion) or pincushion distortion (Siegwart et al., 2011) (p. 157), represented in Figure 11. k_1, k_2 and k_3 are constants, determined during calibration.

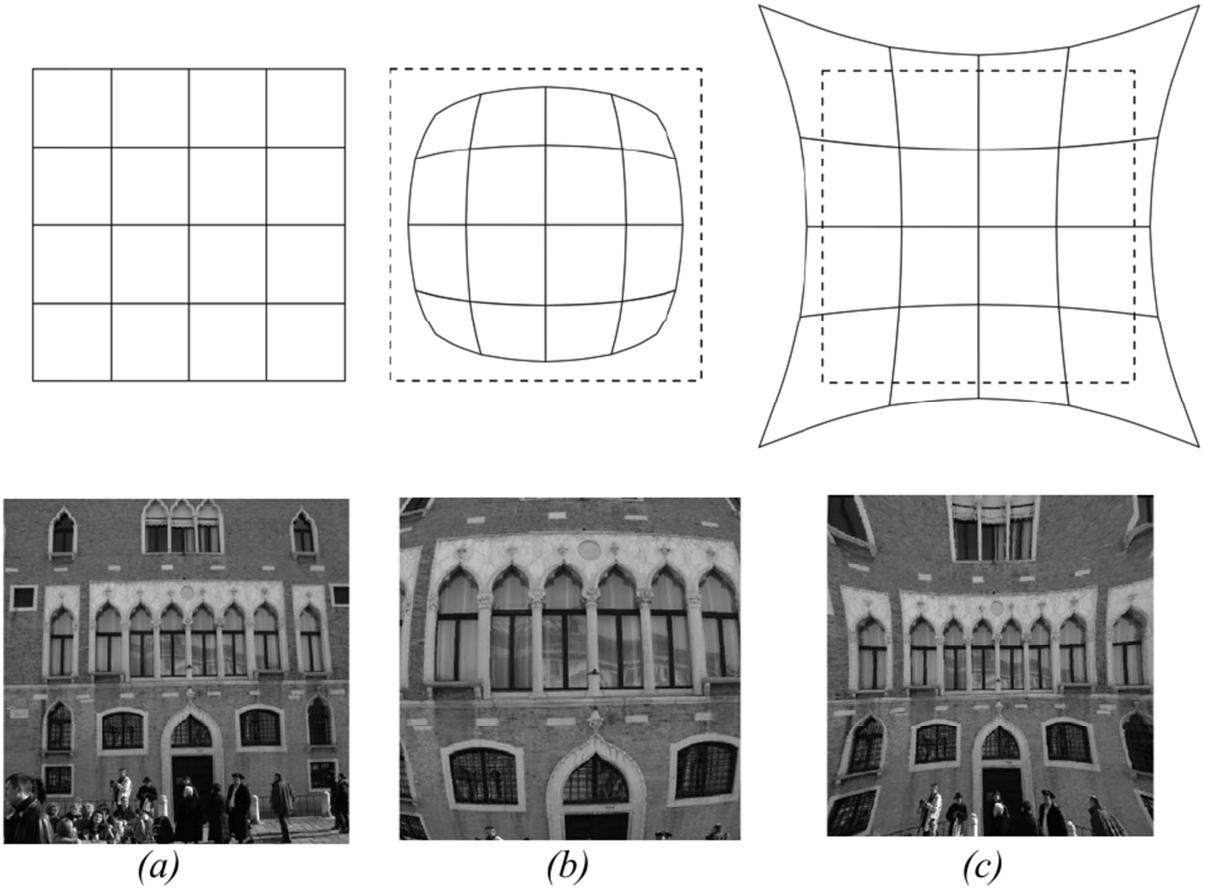


Figure 11: Examples of radial lens distortion: (a) no distortion, (b) barrel distortion, (c) pincushion (Siegwart et al., 2011) (p. 157).

Tangential distortion occurs when the camera sensor (CCD or CMOS) is not perfectly aligned with the lenses when the camera is build. Its effect is less expressive than radial distortion. In many cases the constant parameters p_1 and p_2 obtained through software camera calibration are null. Correction is performed using Equation (4.3).

When the reference coordinate system is arbitrary, Equation (4.1) can be adapted to include a rotation and translation matrix in order to convert to the previous used coordinate reference frame, resulting in Equation (4.4). A simplified form of writing it is shown in Equation (4.5).

$$\begin{bmatrix} w \cdot x \\ w \cdot y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4.4}$$

$$w \tilde{p} = A[R \mid t] \tilde{P}_{world} \tag{4.5}$$

The A matrix from Equation (4.5) is said to contain the camera intrinsic parameters (focus and center parameters). The rotation/translation matrix $[R|t]$ elements are called the extrinsic camera parameters. In a stereo configuration, $[R|t]$ matrices with extrinsic parameters are used in many situations to encode information for changing coordinates system between the reference camera and the secondary one.

As a convention, the left camera used in this work is considered the reference camera (camera 1), while the right camera is considered the secondary (camera 2). Therefore, a $[R|t]$ matrix is used to bring the coordinates from the right camera frame to the left camera reference frame. In this case, extrinsic parameters represent the relative spatial separation between the cameras and the difference in their relative alignments.

4.3 Camera Calibration

The process of determining the intrinsic and extrinsic parameters (Matrix A and $[R|t]$ from Equation (4.5)), and distortion correction coefficients (used in Equations (4.2) and (4.3)) is called camera calibration.

The camera calibration techniques mostly in use today derive from the work of Tsai (Tsai, 1987), which was further modified by Zhang (Zhang, 2000) to enable quick calibration through the use of a planar pattern in the place of using a 3D calibration object, as in the original approach. Both the Caltech MATLAB toolbox (Bouguet, 2013) and the OpenCV library (OpenCV, 2014) are complete implementations of camera calibration, allowing for calibration of a single camera or a stereo pair, following an approach very similar to Zhang's proposed solution (Siegwart et al., 2011).

Figure 12 shows a chessboard-like pattern used in this work as a planar pattern for camera calibration.

Figure 12: Chessboard pattern being used for calibration aided by the OpenCV library. Overlaid is a pattern showing detected inner corners being used as reference for calibration. (Author's figure).

In stereo camera calibration, in addition to determining the intrinsic parameters and distortion correction coefficients for each camera individually, the $[R|t]$ parameters for converting between the coordinates of the secondary camera to the primary are also determined. In addition, the Fundamental and Essential matrices are also provided, and can be used as long as the pose of the cameras don't change in relation to each other. The Fundamental Matrix is the algebraic representation of epipolar geometry, while the Essential Matrix is the specialization of the Fundamental Matrix in which the assumption of calibrated cameras is removed (Hartley & Zisserman, 2003).

Both the Essential and Fundamental matrices represent algebraically the epipolar geometry, which is an interesting property existing between the two stereo image that allows for reducing the area of search for correspondence for a point in one image to a line in another. Figure 13 shows a visual representation of the epipolar geometry, where a plane π is traced between the chosen point x and both camera centers C and C' in (a). When the depth of the x point is not known, search for the correspondent point x' becomes restricted to the epipolar line in (b); (Hartley & Zisserman, 2003). The Essential matrix applies to calibrated images while the Fundamental matrix incorporates the calibration information on it, so it can be applied to uncalibrated images directly.

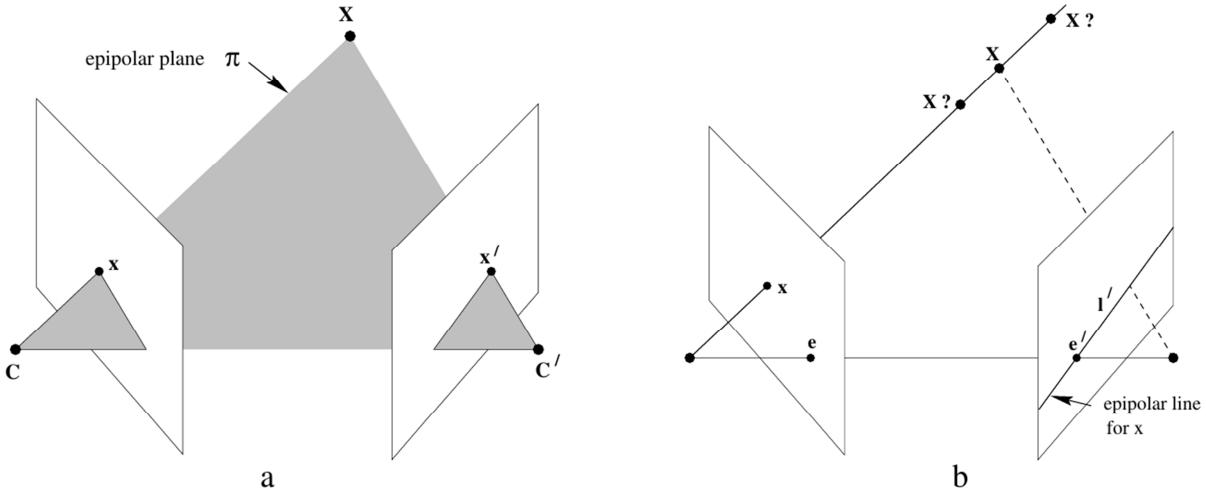


Figure 13: Visual representation of epipolar geometry (Hartley & Zisserman, 2003) (p. 240).

To facilitate the process of finding correspondence and recovering depth information, a process can be applied to the distortion corrected images (either as a whole or to selected points) to transform them so that epipolar lines become horizontal. Thus the search becomes limited to the x coordinate only. This is known as the stereo rectification process (in OpenCV done through the stereoRectify method), and during this process the coordinates of the stereo camera setup converted to a single reference..

After the rectification step, the disparity $u_l - u_r$ can be converted in the depth Z by using Eq. (4.6), with u_l being the x coordinate of the left image, u_r the x coordinate of the right image and b the camera separation. Figure 14 shows the geometric representation of the disparity to depth conversion.

$$Z = b\frac{f}{u_l - u_r} \tag{4.6}$$

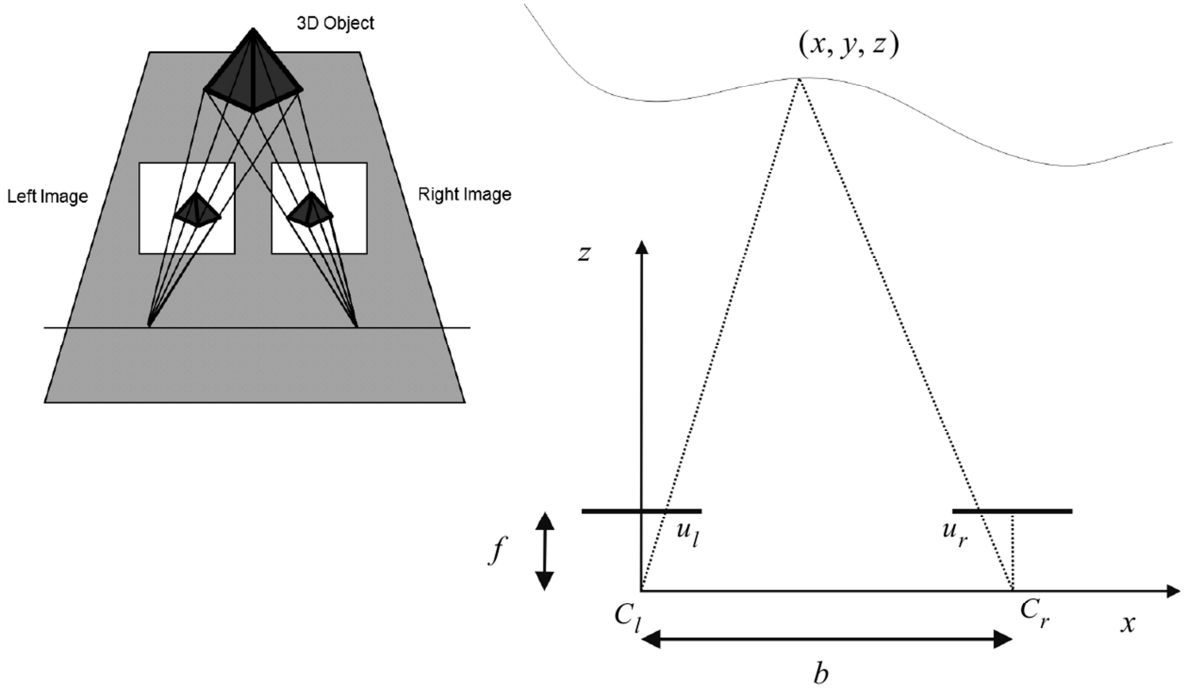


Figure 14: Disparity do depth geometric representation (Siegwart et al., 2011) (p. 172).

OpenCV rectification process provides a disparity-to-depth matrix (Q), which can convert disparity between correspondent points into depth (the Z coordinate) information, which is analogous to equation (4.6). The matrix Q works independently of which image is chosen as a reference, so differently from the equation (4.6) there is no need to label left and right.

Figure 16 shows a pair of original images with the detected corners on the chess pattern and the subsequent rectified images using the disparity-to-depth Q matrix provided by OpenCV. The nonparallel camera configuration used is shown in Figure 15.

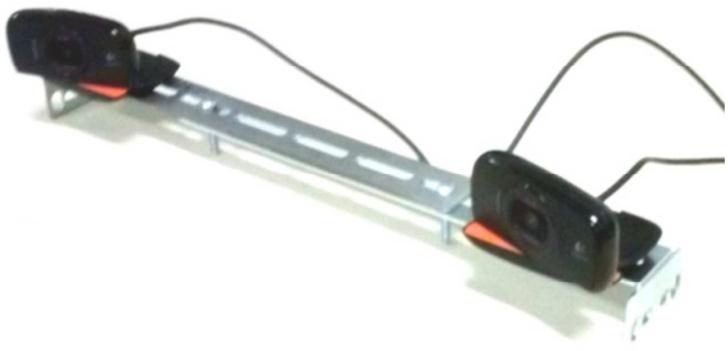


Figure 15: Nonparallel stereo configuration using two Logitech C525 cameras (Logitech, 2014) constructed in the VRI lab. The distance between cameras is 25 cm. (Author's figure).

Figure 16: Original images with detected corners on the chessboard pattern (above), and rectified images with horizontal epipolar lines (below). (Author's figure).

To recover depth, the disparity-to-depth Q matrix is used as depicted in Equation (4.7), where x and y are the point pixel coordinate in the reference camera rectified image, and the disparity $d = x' - x$; with x' being the x coordinate of the secondary camera rectified image. The equation is shown in reduced form in (4.8). The 3D coordinates X , Y and Z are obtained by normalizing the left column vector $\tilde{P}_{world}$ by w.

$$\begin{bmatrix} w \cdot X \\ w \cdot Y \\ w \cdot Z \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Q_{0,3} \\ 0 & 1 & 0 & Q_{1,3} \\ 0 & 0 & 0 & Q_{2,3} \\ 0 & 0 & Q_{3,2} & Q_{3,3} \end{bmatrix} \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} \tag{4.7}$$

$$w \tilde{P}_{world} = Q \tilde{p} \tag{4.8}$$

Thus, for proper using the disparity-to-depth Q matrix in order to construct a reverse camera model, it is necessary to have the system calibrated and that the correspondence between two given points in both pictures is known.

While the task of finding correspondent points may seem easy, it is not trivial for a machine. When considering the epipolar constraints for reducing the correspondence search to

a line, many false positives can be avoided or filtered when finding correspondence, but ambiguity will still exist within the boundaries of the epipolar line.

Increasing the robustness of correspondence algorithms can potentially increase the number of correct matches and reduce the incidence of false positives. This work proposes the use of extra information for finding correspondence between images when matching Points of Interest detected in them.

Traditional algorithms to find the correspondence in stereo images can be classified in two large groups: area based and feature based solutions (Siegwart et al., 2011).

4.4 Area Based Algorithms

A region (or area) of an image that belongs to a stereo pair can be chosen and a correspondent region be searched on the other image. A correspondence is found by applying techniques that select the most similar candidate for finding the correspondence. The most widely used techniques for finding similar image patches are Sum of Absolute Differences, Normalized Cross-correlation, Sum of Squared Differences and Census Transform (Siegwart et al., 2011).

As mentioned before, the search can be confined to an area surrounding a single line, the epipolar line, as a mean to reduce the dimension of the search.

4.5 Feature-Based Algorithms

In contrast to area based algorithms, these solutions extract invariable features from the image, for example corners, edges, line segment or blobs. Attributes associated to these features are then used to perform the matches. The features do not necessarily have a defined geometric entity which they correspond to (Siegwart et al., 2011).

In general, these algorithms are faster and more robust than area-based ones. However, they provide only disperse maps, which need to be interpolated to reconstruct the whole scene depth map when this is required, a problem that doesn't exist on area based approaches (Siegwart et al., 2011).

Some Visual SLAM works uses a hybrid technique, by first searching for corners using the Harris corner detector (as in feature based solutions), then using a patch of image surrounding the chosen corners to describe them, and then comparing these patches of image using the tools for area based solutions (Paz, Pinies, Tardos, & Neira, 2008).

After Lowe's SIFT algorithm appeared (Lowe, 2004), this tendency evolved to the use of a pure feature based solution, which can both detect and describe points of interest. SIFT's detection portion of the algorithm was sometimes not used at first due to being slow compared to corner detectors such as Harris', which were used to detect the points before being fed to the SIFT descriptor. The Multi-scale Harris corner (MSHC) variant of the Harris corner detector has invariance to rotation, scale, affine and illumination changes, and in many cases is more repeatable than SIFT, and is therefore chosen in some implementations (S. H. Ahn, Choi, Doh, & Chung, 2008; S. Ahn, Lee, Chung, & Oh, 2007; Choi, Lee, Ahn, Choi, & Chung, 2006).

Later on, a trend started to improve the speed and/or accuracy of the SIFT algorithm, leading to a publication of a myriad of new algorithms, such as: SURF (Bay et al., 2006), FAST (Rosten & Drummond, 2006), ORB (Rublee et al., 2011), BRIEF (Calonder et al., 2010), BRISK (Leutenegger et al., 2011) and FREAK (Alahi et al., 2012). Getting to know what algorithm is optimal to each particular application is an open problem. An effort to compare these detectors/descriptors for Visual SLAM applications can be found in (Hartmann, Klussendorff, & Maehle, 2013).

In general, feature based solutions work by first selecting points in each image that are both distinctive and have repeatability. Subsequently, the points are assigned an unique identifier (a descriptor) that ideally would be invariant to viewpoint changes (such as camera rotation or zoom) and changes in illumination of the scene (Siegwart et al., 2011). After a list of descriptors is obtained for each image, they are compared and the descriptors that are closer to each other on feature space are considered to be matches. A visual representation of the detected points and the attempt to find the correspondence between them can be seen in Figure 17.

(a)

(b)

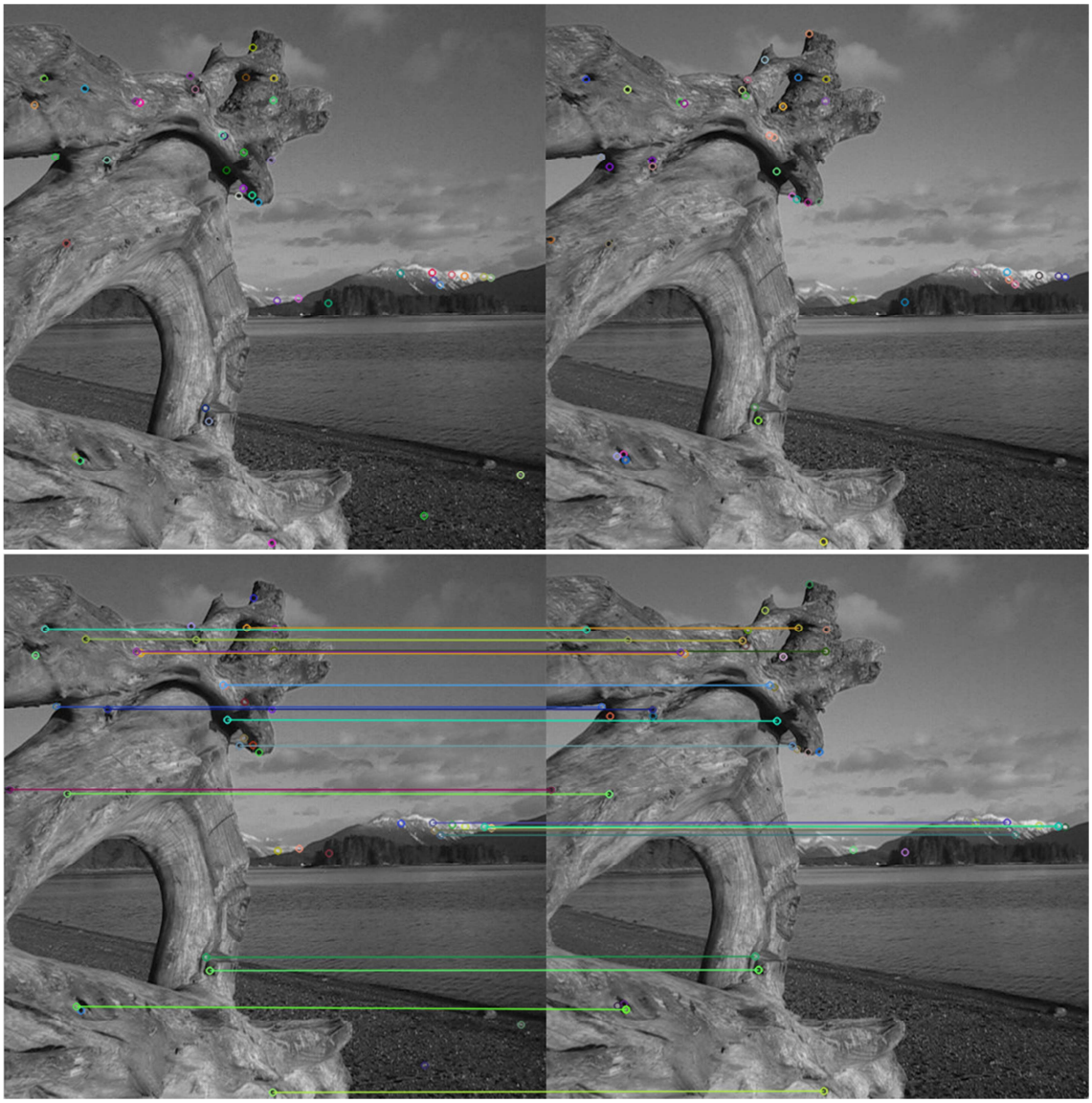


Figure 17: Feature-based correspondence: (a) POIs being detected in the image; (b) matched points by their descriptors. Input images belong to the database available on (Callet, 2010). (Author's figure).

To reduce false positives, the pairs of matched points that don't conform to the epipolar geometry constrains are discarded. When matching is performed, the ratio between the first and second nearest neighbors (found using the k nearest neighbor algorithm with k=2) is considered as a parameter to exclude false positives. Lowe recommends that all matches with a ratio larger than 0.8 be discarded (Lowe, 2004). SURF tests were performed eliminating ratios above 0.7 in the same fashion (Bay et al., 2006). Calculating cross matches, that is, matching the elements from the first image with the second, and then matching the elements from the second with the first can also be used to eliminate pairs that are not correspondent in both ways (Laganière, 2011).

When POIs are matched using k-NN algorithms, each point is considered individually, ignoring the potential spatial relationship information that exists between points. There are techniques that improve matching based on grouping near points and matching them together: (Jung & Lacroix, 2001) and (Ascani, Frontoni, Mancini, & Zingaretti, 2008). Differently, extra information on the descriptor is added by considering curvilinear shape information from a larger neighborhood as a global context, reducing ambiguities (Mortensen et al., 2005).

The proposed solution also intends to consider extra spatial information for matching POIs, but diverge from these two later techniques in that the formation of clusters for group matching is not needed. It considers a global context like in the solution proposed in (Mortensen et al., 2005), but the information used is simpler to extract and use. After detection of POI by using any technique that can provide high repeatability, a global context is considered by treating the points as a starry night pattern, and describing each point as a star belonging to a constellation. This proposal is detailed in the next chapter.

4.6 Discussion

This chapter attempts to elucidate all the steps required from having the raw images to providing the 3D coordinated of POIs to be used as landmarks for SLAM. In Chapter 5, the generic system described in Chapter 2 is converted into a real system for obtaining landmarks for SLAM, adapted to run in an embedded SoC processor with a co-processor in FPGA logic, with the hardware architecture previously shown in Chapter 3.

5. IMPLEMENTATION

In Chapter 1, a generic landmark acquisition system for Visual Slam was presented in Figure 1. The designed system, based on the outlines of the figure, is comprised of software that runs on the Zynq-7000 SoC ARM processor and hardware implemented on the SOC's FPGA.

The implementation of the proposed system is presented in this chapter. Section 5.1 describes the building blocks for the software stack that runs on the dual-core ARM processor. Following it, Section 5.2 describes the hardware co-processor designed in VHDL and synthesized on the FPGA. The innovative descriptor proposed in this work is explained in detail on Section 5.2.12. Finally, Section 5.4 describes the simulations done with EKF-SLAM algorithm running on the embedded platform.

5.1 The Software Stack

The software part of the implemented landmark acquisition system is shown on Figure 18. It runs on top of the Linaro GNU/Linux distribution (version linaro-trusty-alip-20141024-684) (Linaro, 2014), which is a port of the Ubuntu GNU/Linux distribution to the ARM architecture.

Each building block from Figure 18 is explained in detail in what follows. The arrows in the figure depart from each segment of software to their pre-requisites.

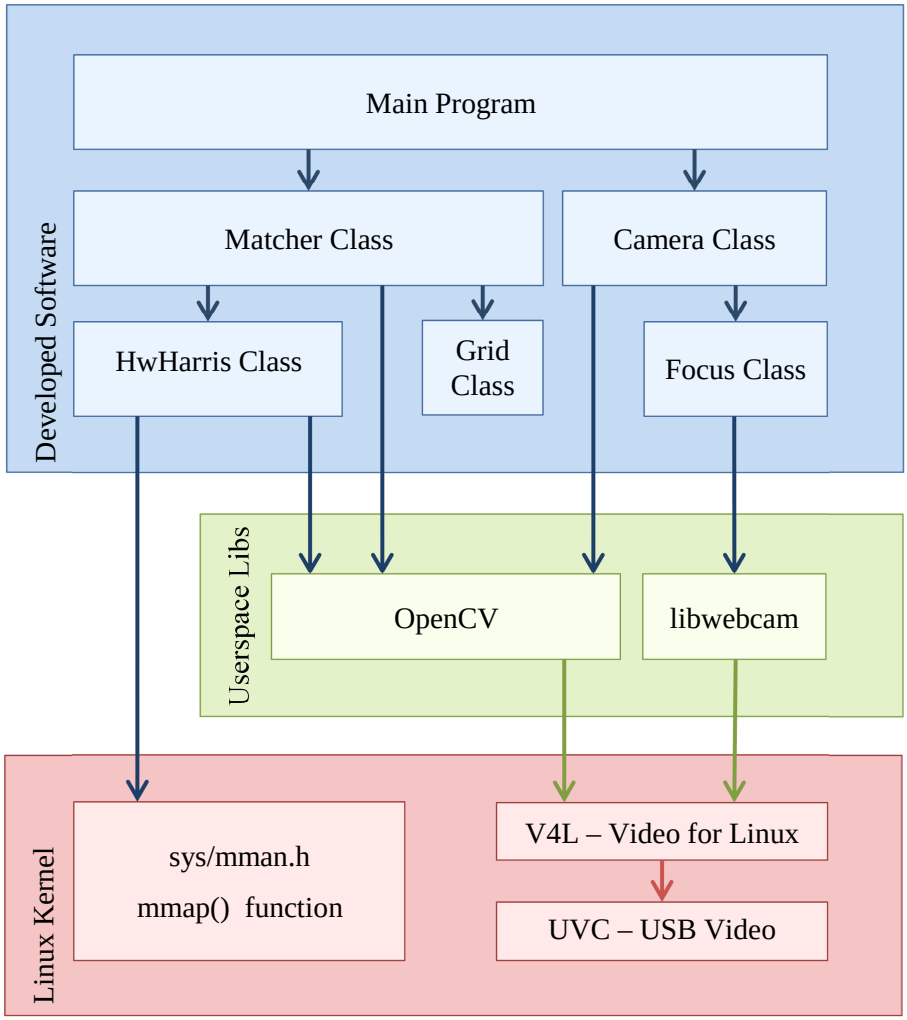


Figure 18: Software stack for the complete landmark acquisition system for Visual SLAM. (Author's figure).

5.1.1 UVC – USB Video Class

The USB Video Class (UVC) is contained inside the Linux kernel, and is a driver for common consumer grade webcams, that supports the two Logitech C525 cameras (Logitech, 2014) used in this work.

5.1.2 V4L – Video for Linux

The Video for Linux (V4L) comprises of a device API for video capture and output and a driver framework for the Linux Kernel. It is a part of the Linux kernel, and can be compiled as a module when building it.

5.1.3 The mmap function

The mmap function, available in the sys/mman.h header (Kerrisk, 2015b), can be used to map the Linux's (real) memory device (*/dev/mem*) file into the virtual address space of the current process. Given that the created co-processor hardware address is set before synthesis, it is known and can then be accessed with a pointer inside a C or C++ program.

5.1.4 Libwebcam

The libwebcam library (libwebcam, 2014), initially developed by Logitech, now a community driven project, exposes controls for some webcams, such as focus distance, aperture and exposition time. It is used as a support library in this work, as the OpenCV library doesn't support controlling these parameters directly from its methods.

5.1.5 OpenCV

The OpenCV library, Open Source Computer Vision, is a library initially developed by Intel and now maintained by Willow Garage (OpenCV, 2014). It is the main library used in this work, providing various algorithms and interfaces used for computer vision applications.

In this work, it is used to capture images from the two webcams, abstracting the interface provided by V4L (Section 5.1.2). This is done through the VideoCapture C++ class, which can also be used to capture images from files.

The Common Interface of Feature Detectors class (FeatureDetector class) provides wrappers that facilitate the switching between different algorithms that can be used for solving the POI detection step, which natively supports the detectors used in this work.

Simmilarly, the Common Interface of Descriptor Extractor (DescriptorExtractor class) also provides wrappers for easily switching between different POI descriptors used for comparison within this work.

To undistort the POIs position, the camera calibration data (Section 4.3) is used with the cv::undistortPoints method.

The BFMatcher class is used for matching the descriptors using a k-NN approach.

5.1.6 HwHarris class

The HwHarris class is responsible for sending the images with the structures compatible with OpenCV to the Harris co-processor. This is accomplished by using the mmap function (Section 5.1.3) to remap the memory address where the AXI4-lite register lies to the virtual address space within the class. It then copies the image pixels to this address (represented in C++ as a pointer) in the correct order. This process is explained within the context of hardware in Section 5.2.2.

The HwHarris class can be seen in Appendix C.

5.1.7 Grid Class

The Grid class includes the code of the proposed algorithm for POI description based on a Star-ID technique from the Grid Algorithm (Padgett & KreutzDelgado, 1997). It is explained in detail on Section 5.2.12.

The Grid class can be seen in Appendix E.

5.1.8 Focus Class

The Focus class is a simple wrapper for abstracting the libwebcam camera controls in a friendlier way for controlling the focus position and exposure time that were relevant to the work.

The focus distance needs to be constant for proper camera calibration using the method presented on Section 4.3. By default, the Logitech C525 webcams change the focus automatically to focus the object in the center of the image. Since this would be impractical

for the use in this work, it is set to a constant value before calibration and kept constant when in use.

The exposure time is by default set to be automatically determined by the image intensities. Since it is calculated independently for each image from the stereo setup webcams, this can result in very different values due to variations on the visible scene, which in practice degrades performance by resulting in different intensity and color values for objects in the pictures to be compared. To solve this problem a constant value is set when calibrating and using the cameras.

The Focus class can be seen in Appendix G.

5.1.9 Camera Class

The Camera class abstracts the initialization of both cameras for capturing the images. It turns them on by using the OpenCV VideoCapture class and sets the focus distance and exposure time settings using the Focus class (Section 5.1.8).

5.1.10 Matcher Class

The Matcher class encapsulates the OpenCV funcions mentioned in Section 5.1.5, providing POI detection, description, correspondence and distortion correction through camera calibration. It adds the ability to use the Harris co-processor by uniting it with the POI detectors already available in OpenCV by interfacing with the HwHarris class (Section 5.1.6), and the modified Grid descriptor by interfacing with the Grid class (Section 5.1.7).

Also, it provides its own methods to perform ratio, symmetry and epipolar constraint tests, which can be seen in Appendix F.

The ratio test takes the 2 nearest neighbors using a 2-NN matcher and compares them, eliminating the point above a given ratio. When the distances are similar, the ratio approaches one. For SIFT, the author recommends a ratio of 0.8. Points that exceed this ratio are eliminated (Lowe, 2004). This test helps eliminating ambiguity.

The symmetry test verifies if the closest neighbor from the left to the right image are the same when performing the test from right to left. This also helps to eliminate ambiguity.

The epipolar test checks if the rectified points (Section 4.3) from camera calibration are lying on the same line (within a certain number of vertical pixels due to noise).

5.2 Design of the Harris co-processor

5.2.1 Overview

The task of detecting POIs is, time wise, significantly taxing in the context of landmark detection for Visual SLAM. It was found to be the most time demanding task in early analysis using software only in the main processor, as described in Section 6.1.

The Harris and Stephens algorithm for corner detection (Harris & Stephens, 1988) was chosen to perform the POI detection in this work, due to the high repeatability of detection, when compared with other widely used POI detectors (Siegwart et al., 2011) (p.233). It is also simpler to implement than other POI detectors in hardware, since most of its components require only local information for data dependency.

The FPGA part of Zynq-7000, available on the chosen board, has 36Kb of Block RAM. Since the addressing considers a byte to be composed as 9 bits, 8 bits plus an extra bit for parity, 4KB of memory are available. A gray-scale pixel is typically represented as a 0-255 decimal value representing its intensity, which is represented as a single byte. This limits the pixels that can be stored in the board to 4,000, leaving a square 64x64 image as the upper limit resolution to be processed if the whole image is stored into the FPGA block memory.

In contrast, the images acquired from the two Logitech C525 cameras (Logitech, 2014) have an upper limit of 1280x720. This is limited in practice due to simultaneously streaming in the same USB 2.0 controller to 640x360 pixels in stereo configuration at 30 fps in ZedBoard, where only one of the two USB controllers is accessible to the external USB ports.

Since the image sizes are very large compared to what can be stored locally for processing in the FPGA, the proposed architecture was designed to avoid using Block RAM

and to rely in the minimum data dependency required for calculating if a certain pixel is considered a corner or not.

Due to the filters applied in different steps to the image, the minimum square region required was determined to be a 7x7 window. This assumes that the non-maximum suppression is searched in the smallest area possible of 3x3, and considers all de data dependencies of the filters that are components of the Harris and Stephens algorithm. This conclusion was independently determined in the article of (Amaricai, Gavriliu, & Boncalo, 2014), that also explores an architecture that tries to minimize the use of Block RAM on a different FPGA. In the architecture used, sliding window, the 7x7 window is processed then moved to the right until it reaches the end of the line, when it starts at the next line and so on until the end of the image. An exemplification can be seen in Figure 19.

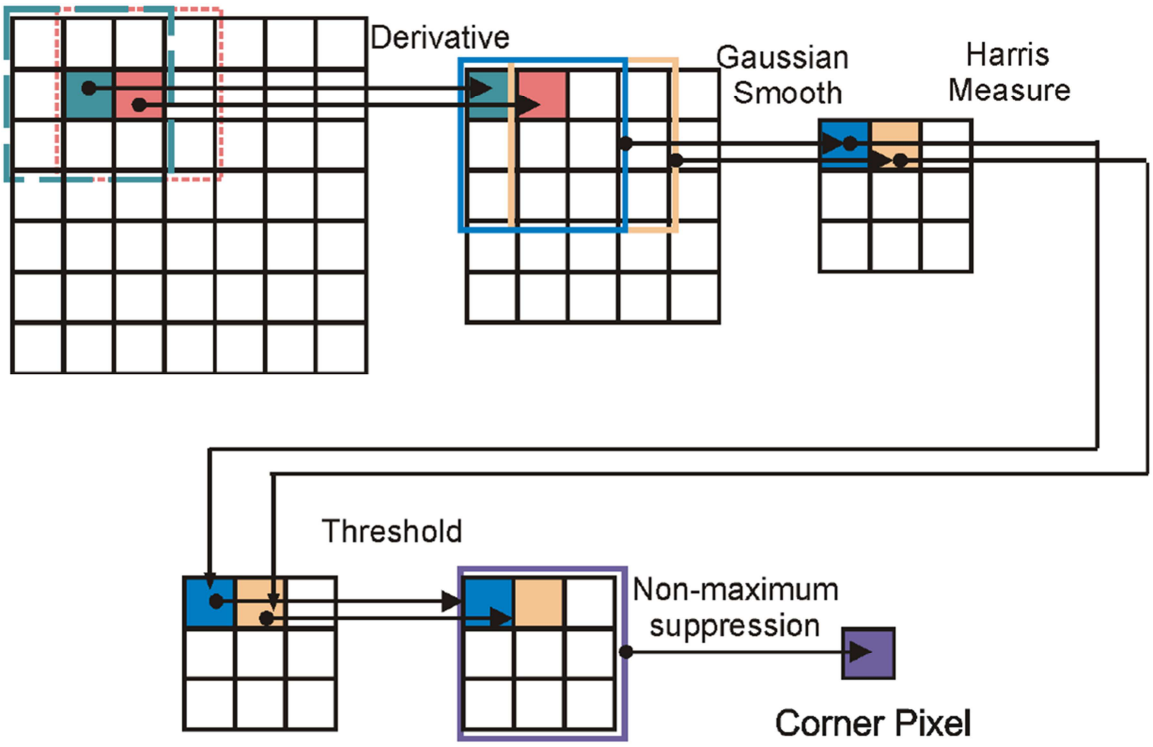


Figure 19: Processing Sliding Window for Harris Corner Detection (Amaricai et al., 2014).

The Harris and Stephens Corner detector algorithm used in OpenCV version 2.4.9 was chosen as a reference for the proposed design of this work (OpenCV, 2014). Its structure can be seen in Figure 20. Six sequential steps are performed in the six blocks within the gray area, which correspond to the Harris algorithm. For the outside blocks, the first one represents the main processor, while the second and last are auxiliary blocks that convert the input and output information for optimal throughput.

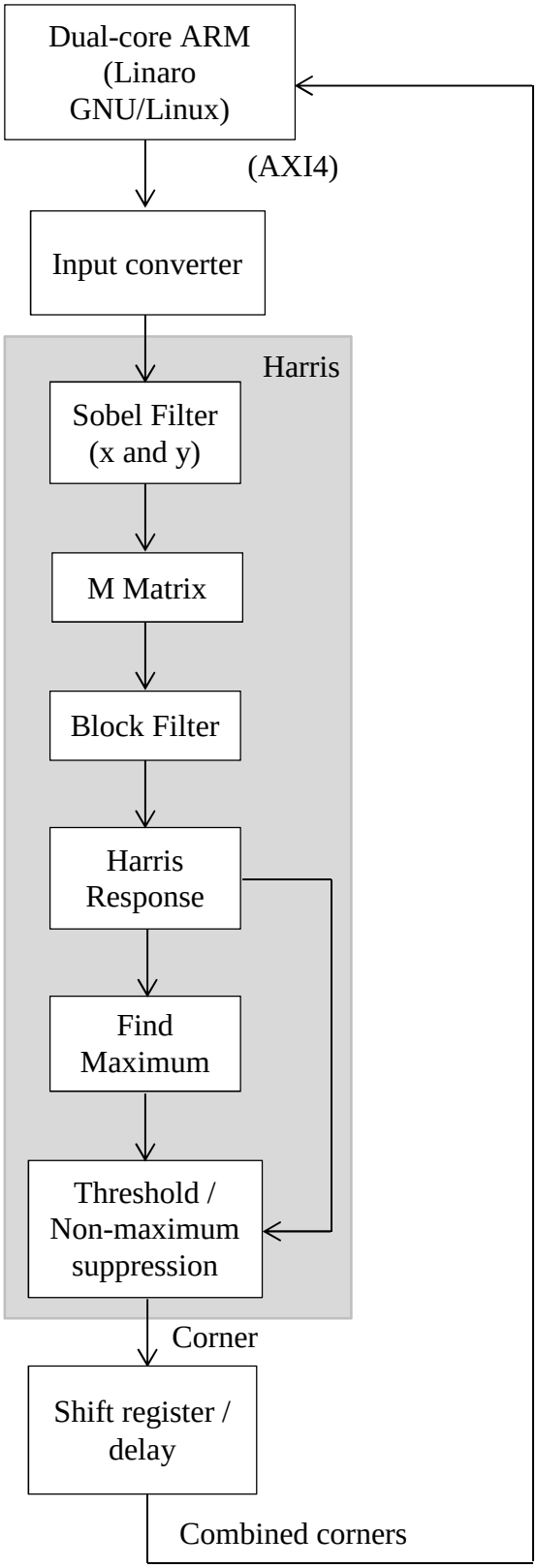


Figure 20: Simplified block diagram for the proposed hardware architecture for the Harris Algorithm on FPGA. The GNU/Linux Linaro distribution runs on the dual-core processor (first block), which connects to the remaining blocks (implemented on FPGA logic) through the AXI4 interface. The blocks inside the grey area belong to the Harris algorithm, which is the corner detector. The corners that are determined by the algorithm are combined in the shift register, and then read again through the AXI interface by the main processor. (Author's figure).

The block architecture shown on Figure 20 and used in this work relates to the elements shown in Figure 19 as follows: the derivative corresponds to the Sobel and M Matrix blocks, the Gaussian Smooth was replaced by the Block Filter, the Harris Measure corresponds to the Harris Response calculation, the Threshold is replaced by an Adaptive Threshold, which requires an extra block to keep the maximum values and the Non-maximum suppression remains unchanged.

The FPGA in Zynq-7000 operates at 100MHz, with the clock period being 10ns. Due to propagation delay (gate delay), the whole Harris processing cannot be performed in a single FPGA clock period. In FPGA hardware synthesis using the Xilinx platform, implementing designs that have operations where the propagation delay exceeds the clock period result in timing constraints errors.

The initial solution to overcome timing constraint problems was to divide the algorithm in 6 consecutive steps, each one performed in a single clock period, that correspond to the 6 main steps in Harris (Figure 20). This approach reduced timing constraints problems significantly, and only localized problems in the Harris Response step remained due the cascaded operations performed in this step (shown in detail in Section 5.2.7). The further division of the Harris Response in two periods instead of one solved the remaining problems with timing constraints, with the algorithm being performed in a total of 7 clock cycles.

For this sliding window architecture, 7 new pixels are required to slide the 7x7 window for each calculation, taking 7 clock steps for input. The similarity with the 7 steps required for performing the Harris algorithm was exploited to synchronize the active blocks with the input data, which is explained later in Section 5.2.3.

The following sections explain each block of Figure 20 in detail.

5.2.2 Dual Core ARM

The first block in Figure 20 corresponds to the physical main processor. The ARM processor runs the Linaro GNU/Linux distribution (version linaro-trusty-alip-20141024-684)

(Linaro, 2014), which is a port of the Ubuntu GNU/Linux distribution to the ARM architecture. The software that runs on the distribution is detailed in Section 5.1.

Every block below the processor in Figure 20 is implemented with FPGA logic as an Intellectual Property (IP) and connected to the main processor through the AXI4-lite interface.

Two approaches can be used to transfer data between software running on the processor with a GNU/Linux distribution and the FPGA through the AXI4 interface: using a device driver and direct mapping the memory.

For the first approach, a kernel driver template is provided by Digilent in (Digilent Inc., 2013), which exposes the AXI4-lite registers as a file in the file system. Writing a 32-bit value to this file is equivalent of writing the hardware register, and reading it is correspondent to reading the register itself. Using this driver, it is possible to interface with the hardware using any programming language that is supported on Linux on the ARM platform, as long as it can read and write to files.

Alternatively, the second approach can be used to map the Linux's (real) memory device (*/dev/mem*) file into the virtual address space of the current process. Given that the created peripheral hardware address is known, it can be accessed with a pointer inside a C or C++ program. This is accomplished by using the *mmap* function available in the *sys/mman.h* Linux's library (Kerrisk, 2015b).

A C++ program was written using the second approach as a C++ library to send an image to the hardware in the order that the Multiplexed Input Converter block expects, and receive back the corner information (See Section 5.1.6). The images sent to hardware can either be read from the file system for test purposes or acquired from the two Logitech HD C525 webcams (Logitech, 2014) used for stereo corner detection in this work (See Section 5.1.5). This webcam has controllable focus distance and exposure time, and these controls are supported on Linux through the libwebcam C library (libwebcam, 2014), which interfaces with the USB Video Class (UVC) Linux Driver, available in the mainline Linux kernel (Explained on Section 5.1.4 and 5.1.8).

5.2.3 Multiplexed Input Converter

Internally, the multiplexed input converter is composed of four identical blocks (Input converters). Each one receives a single pixel at a time serially and builds the 7x7 window required to determine if a pixel is or isn't a corner.

The initial 7x7 matrix is initialized with zeroes at reset. A counter that goes from 1 to the number of rows (7) is incremented after each received pixel. The row corresponding to the counter is shifted to the left, while simultaneously the received pixel is added to the rightmost column at the row corresponding to the counter. When the counter reaches the last value, it returns to 1. To exemplify the order in which the matrix is constructed, an equivalent 3x3 matrix showing the entering pixel order is presented in Figure 21, which works similarly to what a 7x7 matrix would.

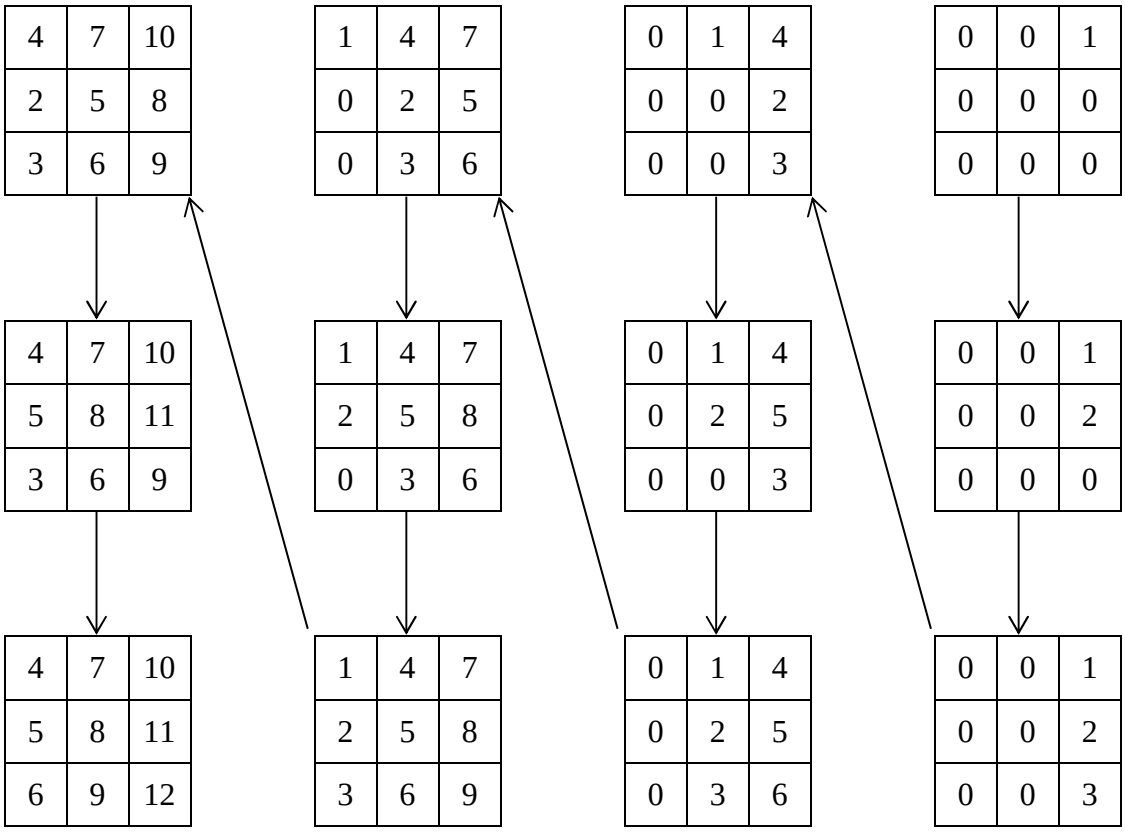


Figure 21: Order of entering pixels for a 3x3 window. (Author's figure).

For calculating if the first pixel is a border, the whole matrix must be filled. Once the matrix is full, in order to compute the next pixel in the same row, just the next column needs to be received, and the remaining values are shifted to the right. Therefore, for the first

calculation, 7x7=49 pixels are written into the matrix, but for the next calculation, only 7 more are needed. With this order, the window effectively slides to the right.

While the matrix is filled for the first time, all blocks that follow the input converter remain still. After that, when each of the 7 next pixels of the same row are individually added, one of the 7 steps of the calculation are executed synchronously.

The AXI4 lite interface works with 32-bit input and output registers. In the proposed platform, a gray scale pixel uses only 8 bits, representing the intensity in decimal value between 0 and 255. By concatenating 4 pixels from different regions of the same image together, 4 different 7x7 windows can be constructed simultaneously. The 4 different window matrices are processed serially in the remaining blocks that work as a pipeline. A multiplexer selects each matrix when the matrix is needed by the next step (Figure 21). The output of the multiplexer is connected to the input of the next stage, the Sobel x and y blocks.

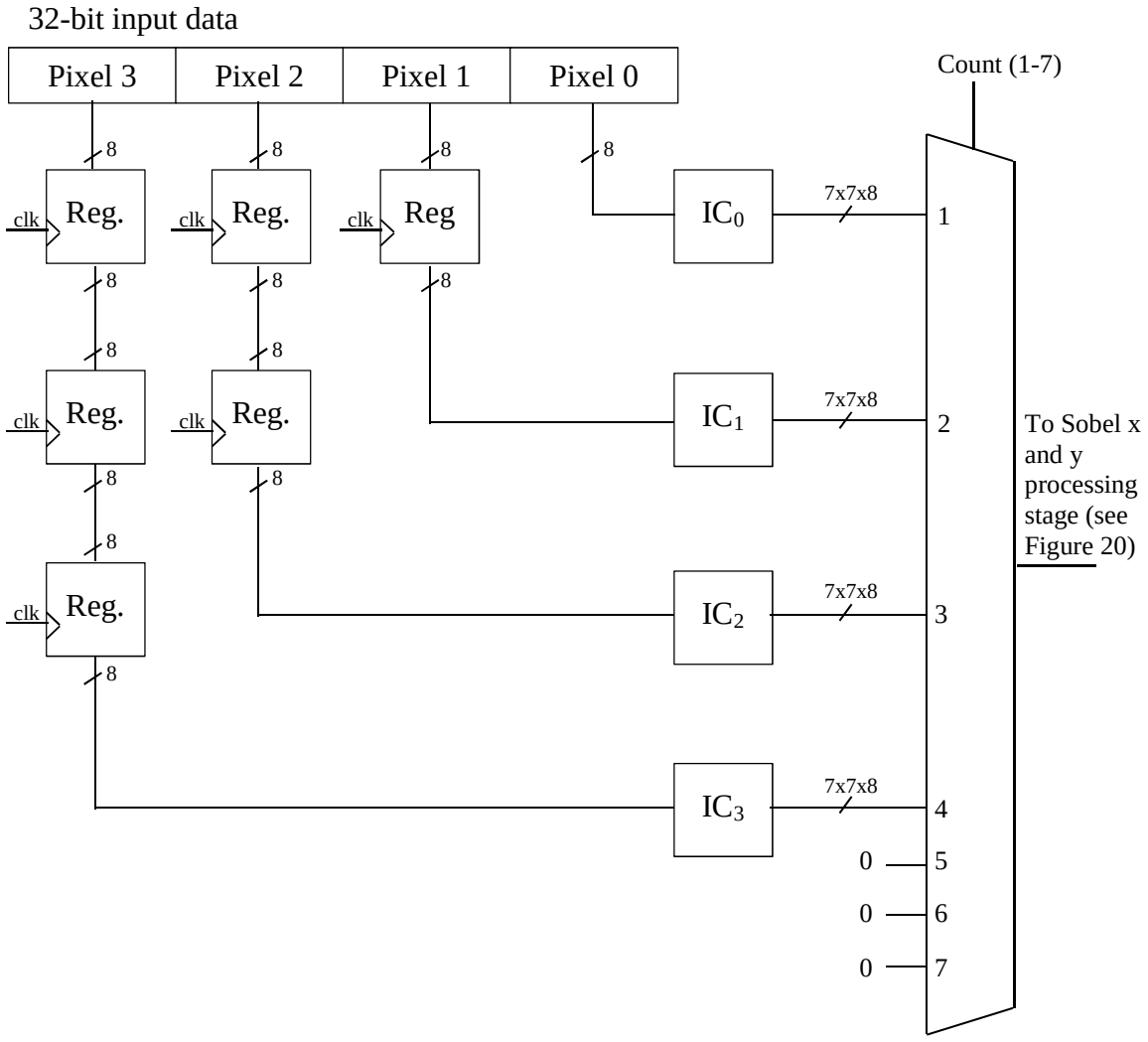


Figure 22: Multiplexing four 7x7 window input converters (IC_0 to IC_3). (Author's figure).

Using this structure, it is possible to increase the input data being processed by a factor of four without the need to change the blocks that come after the multiplexed input converter, except for doing a similar concatenation of the output. This is further discussed in Section 7.2.

By delaying the input being fed to the individual input converters, the matrices are correctly filled exactly when they are ready to be loaded into the pipeline by the multiplexer. In contrast to delaying with registers the whole output matrix of the converters, this saves logic elements in the FPGA.

The multiplexed input converter code can be seen in Section (e) from Appendix D.

5.2.4 Sobel x and y

Two blocks approximate the gradient functions of the image intensity function (I_x and I_y) by convoluting the Sobel operator (mask) in the x and y directions with the image intensity (I), as shown in (5.1).

$$I_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$
$$I_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I \quad (5.1)$$

A matrix of 5x5 of the Sobel x block and another of 5x5 of the Sobel y block are formed using the generate statement from VHDL. The input of both blocks is connected to the 7x7 8-bit unsigned values matrix from the Multiplexed Input Converter. The output is two 5x5 11-bit signed matrices, one for each direction. The calculations that determine the least amount of bits needed for the output (11) were done in a Wolfram Mathematica script and can be seen in Appendix A. Both x and y directions are calculated concurrently in a single clock period.

The Sobel code can be seen in Section (h) and (i) from Appendix D, for the x and y directions, respectively.

5.2.5 M Matrix Coefficients

Harris and Stephens define the M matrix, shown in eq. (5.2), as composed by three coefficients, A, B and C, which are calculated from the gradient values I_x and I_y as shown in eq. (5.3) (Harris & Stephens, 1988).

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix} \tag{5.2}$$

$$\begin{aligned} A &= I_x{}^2 \\ B &= I_y{}^2 \\ C &= I_x \cdot I_y \end{aligned} \tag{5.3}$$

The gradient values are provided in two 5x5 matrices, one for each of the orthogonal directions, and processed to calculate the A, B and C values in a 5x5 structure of blocks that do the processing in parallel. The output is three 5x5 matrices with 16-bit signed values. Internally, 21 bit signed values are used for the calculation, which is the minimum required (see Appendix A). To reduce the flip-flop usage count, the 21-bit signed values have their 5 least significant bits truncated to 16-bit signed values. All the calculations are done concurrently in a single clock step.

The VHDL code can be seen in Section (j) from Appendix D.

5.2.6 Block Filter

Harris and Stephens suggest the use of a Gaussian filter to reduce noise in the A, B and C coefficients calculated for the autocorrelation M Matrix (Harris & Stephens, 1988). The OpenCV implementation of the corner detector uses a simplified mask that averages the 3x3 neighborhood around the selected pixel, which is called a Block Filter (Bf), and shown in eq. (5.4).

$$Bf = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{5.4}$$

Convoluting the three previous 5x5 16-bit signed matrices (A, B and C) with the Bf matrix, while ignoring the boundary values, results in three 3x3 matrix with 20-bit signed filtered values (Appendix A). All the steps needed for the convolution are done in parallel. The output values are again truncated to 16-bit by removing the 4 least significant bits.

The Block Filter code can be seen in Appendix D, Section (k).

5.2.7 Harris Response

The Harris Response (R) is calculated using the determinant (Det) and trace (Tr) of the M matrix, where k has a typical value between 0.04 and 0.06 (Aydogdu, Demirci, & Kasnakoglu, 2013), as seen in eq. (5.5). $\overline{A}$, $\overline{B}$ and $\overline{C}$ are the M matrix components filtered by the block filter.

$$\begin{aligned} R &= Det - k \cdot Tr^2 \\ Det(M) &= \overline{A} \cdot \overline{B} - \overline{C}^2 \\ Tr(M) &= \overline{\mathrm{A}} + \overline{\mathrm{B}} \end{aligned} \tag{5.5}$$

OpenCV Harris implementation uses $k = 0.04$ as the default value, which was approximated as $k = 1/2^5 + 1/2^7 \cong 0.0391$ in this implementation. This approximation can use two bit shifts followed by an adder to avoid the need of a multiplication.

Making the substitutions in eq. (5.5) so that it can be written in a single equation results in Equation (5.6).

$$R = \overline{A} \cdot \overline{B} - \overline{C}^2 - k \cdot \left(\overline{A} + \overline{B}\right)^2 \tag{5.6}$$

Because the rightmost term of eq. (5.6) is synthesized in hardware as an adder followed by a multiplier, this operation can't be performed in a single clock period in the FPGA. This required the calculation of the Harris Response to be made in two steps, increasing the needed steps for the Harris calculation from 6 to 7.

The resulting calculations can be expressed as a single 3x3 32-bit signed integer matrix, as seen in Appendix A. The corresponding VHDL code for the Harris Response can be seen in Section (l) from Appendix D.

5.2.8 Find Maximum

This step is concerned with keeping in memory the maximum value from the Harris Response found within the whole image, so that it is available to use in the next frame as a reference for the adaptive threshold step that equals to zero all values below the threshold value, which is proportional to the maximum value (explained in Section 5.2.9). It reads the center value $(R_{2,2})$ from the 3x3 32-bit signed matrix calculated previously from the Harris Response step. It is not necessary to look at the non-center values because of the sliding window behavior they eventually reappear at the center position. The code for this step is shown in Appendix D, Section (m).

5.2.9 Adaptive Threshold

Although most Harris implementations in hardware use a fixed threshold value, this approach doesn't give efficient results when illumination changes in a large range (Birem & Berry, 2012).

The approach followed in OpenCV uses by default an adaptive threshold of 0.01 times the maximum value of the response found within the current image. This would require the entire response being calculated within the whole image before the threshold could be applied to the image, so an approach like the proposed here where just a 7x7 image window is sent to the FPGA can't benefit from an adaptive threshold like the one used in OpenCV.

To overcome this limitation, given that the application of the Harris Algorithm is a sequence of frames in which the difference in illumination is small between consecutive frames, instead of using the actual maximum value for calculating the threshold value, the maximum value from the previous frame is used. This allows the adaptive threshold to be calculated with only the 7x7 region being available from the second frame onwards. The use of the previous image information to compute the threshold for the next image was already applied for space applications with success by (Di Carlo et al., 2013).

Again, avoiding using a floating point multiplier, the constant used for calculating the adaptive threshold is approximated as $value = 1/2^7 + 1/2^9 \cong 0.00977$, using two bit shifts followed by an adder. All values below the threshold multiplied by the maximum from the

previous frame are changed to zero. Adjusting this constant effectively changes how high a response needs for an image pixel to be to be considered a corner.

The Adaptive Threshold code can be seen in Section (n) of Appendix D.

There is no change between the input and output bit length (32 bits), and the matrix is kept at size 3x3.

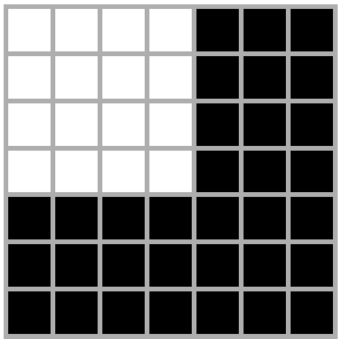
5.2.10 Non-maximum Suppression

From the output of the previous adaptive threshold stage, the center response value is considered a corner if it is the maximum value when compared to its 8 neighbors. The output from this stage is a single bit signaling if a corner was found. This step is taken in the same clock period as the adaptive threshold step, thus is represented as a single block in Figure 20. The VHDL code, however, is kept separate and is shown in Section (o) from Appendix D.

5.2.11 Shift register and delay

This last step concatenates the 4 bits that signal if a corner was detected in one of the 4 windows processed by the pipeline. Due to the timing requirements, the output will be ready after 12 clock steps. To simplify the design of the software that controls the I/O in the main processor, a delay is induced so the data will be ready within 14 clock steps. This allows the corner status to be read after 7 write operations are done in the hardware. Thus, when the 7 pixels are written to apply the algorithm for each of the 4 windows that are introduced simultaneously, the corner status of the center pixel of these windows will be ready after two more windows are written to the hardware and so on. The code is shown in Appendix D, Section (p).

To illustrate this delay in the context of the signals involved, a simulation of the complete system was run using the ISIM simulator from the Xilinx tools, and a time diagram was obtained (Figure 24), where the delay between data input and output is shown. The first two signals correspond to *clock* and *reset* (negated). The third signal, *load*, indicates when the input matrix is being filled. In this simulation, a corner image patch is loaded into the four input converters (Figure 23).



$$IM = \begin{bmatrix} 255 & 255 & 255 & 255 & 0 & 0 & 0 \\ 255 & 255 & 255 & 255 & 0 & 0 & 0 \\ 255 & 255 & 255 & 255 & 0 & 0 & 0 \\ 255 & 255 & 255 & 255 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 23: Harris image patch designed to cause a corner to be detected. (Author's Figure).

On Figure 24, the *control signals*, from lower up, indicate when each block of the Harris Algorithm (Figure 20) is active, with 2 signals controlling the two parts of the Harris Response block. Each block is activated for 4 clock periods, when they are processing the four input matrices independently in the pipeline. The output bit that indicates a detected corner (*crn*) appears after 8 to 11 clock pulses since the input matrix is available. They serve as input to the shift register (*shr*), and the output is delayed so it is available in the 14^{th} pulse, immediately after the new matrix is loaded into the circuit. This helps synchronizing the I/O controller in software.

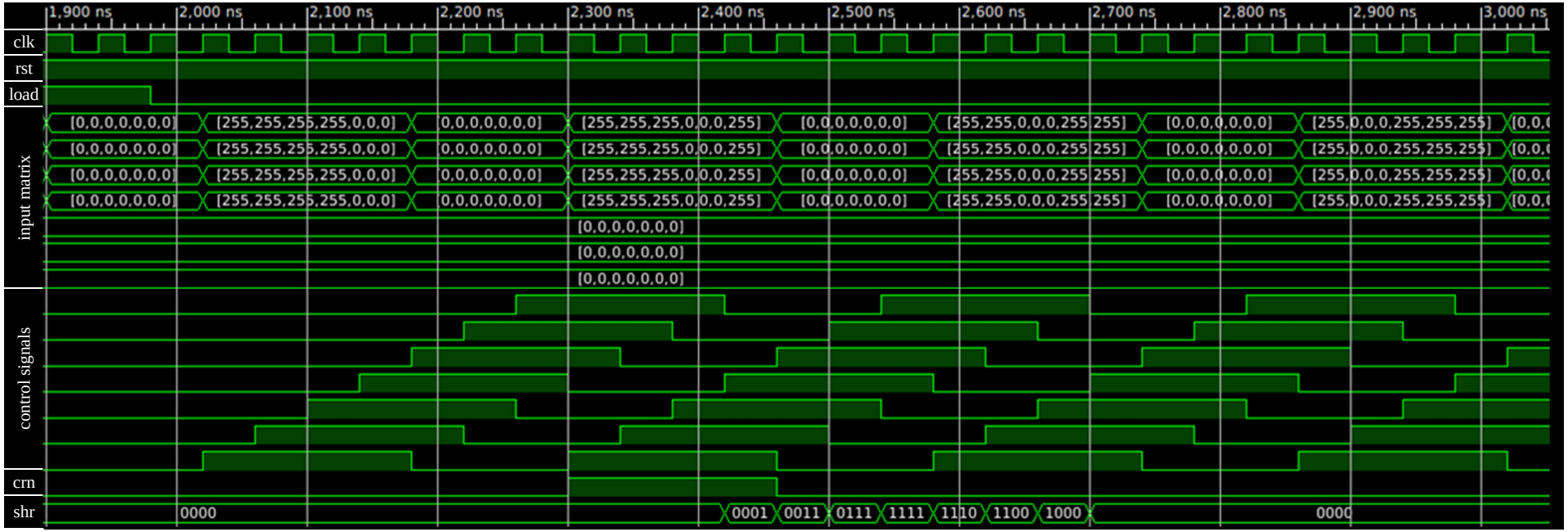


Figure 24: Digital timing diagram for input, control signals and output. Signal *clk* is the clock, *rst* the reset, *load* indicates when the input matrix is complete and the circuit can start processing the data, *crn* indicates if a corner was found and *shr* is the shift register output of the corners with delay. After the matrices are loaded, indicated by the *load* signal going from one to zero, the *control signals* from bottom up indicate when each composing stage of the Harris co-processor is active (each one is active on 4 periods). Each *control signal* is bound to a single stage, with the exception of the Harris response stage that is divided in two parts and receives 2 signals. The output bit that indicates a detected corner (*crn*) appears after 8 to 11 clock pulses since the input matrix is available. It serves as the input of the shift register (*shr*), and the output is delayed so that it is available in the 14th pulse, immediately after the new matrix is loaded into the circuit. (Author's Figure).

5.2.12 Synthesis

With the architecture and configuration presented in the previous sections of this chapter, the design could be synthesized in the FPGA, occupying 71% of the available slices (The complete synthesis log can be seen in Appendix B).

The synthesis is done within the Xilinx EDK program, following the steps shown in the Digilent documentation (Digilent Inc., 2013). During the creation of the custom IP, it is necessary to keep note of the memory mapping address of the AXI4-lite register, which is used in I/O communication in the HwHarris class (Section 5.1.6). The default custom IP user_logic.vhd file should be overwritten with the VHDL code shown in Appendix D, Section (a).

5.3 Star-ID based Descriptor

The descriptor here proposed treats the pattern formed by detected POIs analogous to the star pattern used for matching in the problem of autonomous star identification (Star-ID). The problem of star identification consists in extracting stars from an image acquired by a CCD or CMOS sensor and, by matching the measured stars with a catalog, identify what stars are in the field of view of the sensor (Na & Jia, 2006).

There are many descriptors used in solutions for the Star-ID problem. The extracted characteristics vary significantly between algorithms. Surveys that summarize the field's scientific research are available in (Ho, 2012; Na & Jia, 2006; Spratling & Mortari, 2009). The following two Sections explain the most widely used characteristic, angular distance, and the characteristic used in the prototype developed in this work, the grid pattern.

5.3.1 Angular Distance

In Star-ID, the most widely used characteristic for describing stars uniquely is the angular distance between them, shown in Figure 25, which was first described by (Liebe, 1993). In astronomy, the angular distance corresponds to the angular separation between the

two stars originating from the same observer, and it is visually seen as the linear distance between them.

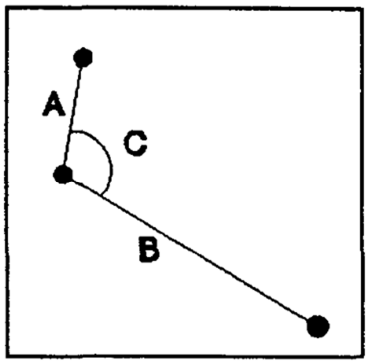


Figure 25: The triangular feature. A: the angular distance to the first neighboring Star; B: the angular distance to the second neighboring star; C: the angle between the neighboring stars (Liebe, 1993).

5.3.2 Grid Algorithm

A different way of extracting features from the available scene is to use a star pattern, a technique pioneered in the Grid Algorithm (Padgett & KreutzDelgado, 1997). The Grid Algorithm uses a loose grid to describe the observed stars. A visual representation of the extraction of the descriptor is shown in Figure 26.

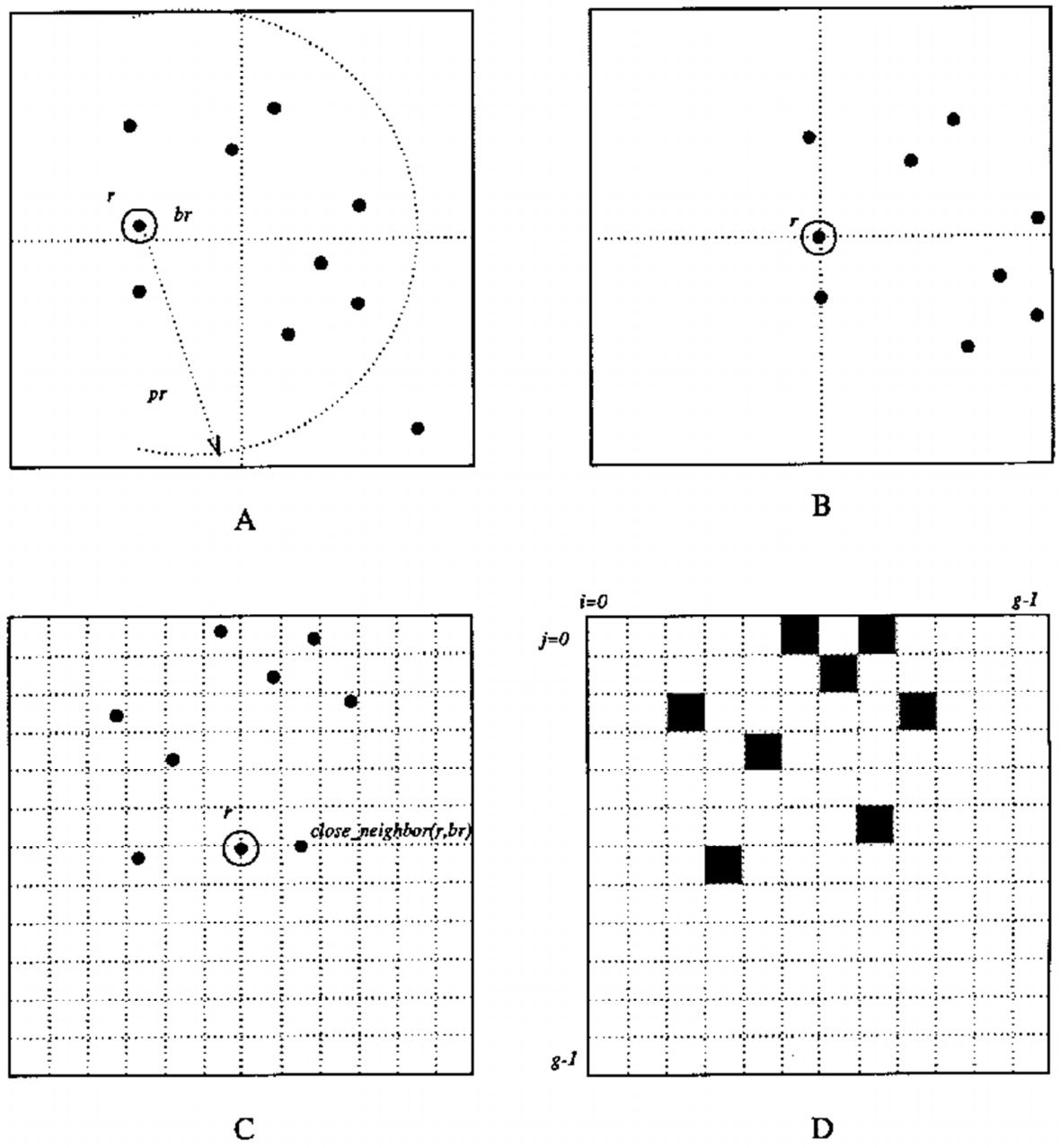


Figure 26: Feature extraction using the Grid Algorithm (Padgett & KreutzDelgado, 1997).

The descriptor is extracted as follow, with relation to the subfigures from Figure 26:

a) A star r is selected as the reference to create the pattern;
b) The r star and a part of the surrounding sky with radius pr is translated to the center.
c) A loose square grid of side g is placed, with the pattern rotated so that the closest neighbor star will lie in the x coordinate axis (achieving rotation invariance).

d) A g^2 length bit vector is derived from the grid pattern. The presence of a star in a cell is represent linearly in the vector so that the bit $k \times g + i$ is 1, while its absence is represented as the bit 0 .

For example, considering the coordinates [0,0] as the least significant bit (LSB), the hexadecimal descriptor for the pattern on Figure 26 would be:

00.00.00.00.00.00.00.00.08.08.00.00.01.01.04.04.00.a0

A simplification of Padgett and Kreutz-Delgado's algorithm was implemented in this work as the initial step for validating the applicability of Star-ID algorithms to the correspondence problem in stereo images. The loose grid pattern is extracted in both rectified images. Since rectification was performed and epipolar lines are horizontal, the correspondent features will not be rotated between the images, so the *(c)* step for achieving rotation invariance was skipped, and the descriptors are created without any kind of rotation on the input images. Instead of limiting the radius to pr , the pattern is limited in size only by the grid square side g . The resolution of the grid is reduced to achieve the loose effect present in the grid (as on Figure 26) by a factor of 2^h, where $h = \{0,1,2,3,4,...\}$. Increasing the h parameter results in a lower resolution of the grid.

5.4 EKF-SLAM implementation

To evaluate whether a simple SLAM implementation could also run alongside the pre-processing of landmarks on the embedded platform, a C++ implementation of EKF-SLAM using a 2D top-view map approach was ported from the MATLAB implementation available on (Solà, 2013). The Eigen C++ library for linear algebra was used to facilitate matrix and vector operations (Jacob & Guennebaud, 2014).

Since the original implementation included a simulator, which was responsible for presenting sensor data to the system with added Gaussian noise, testing the system would require that the simulator was either ported to C++ or left as MATLAB code. The second approach was chosen, and the MATLAB C/C++ Engine was used to transport data structures between MATLAB and C++, and to call MATLAB routines from the C++ code (The Mathworks, 2014). To run the C++ code in the ARM processor of ZedBoard, data was

interchanged between the simulator running on MATLAB on a standard PC and the C++ code running in the ARM processor on the Linaro GNU/Linux distribution through Ethernet using the Qt 4.8 library QtNetwork module (Digia plc, 2014). Figure 27 shows the structure of the test system.

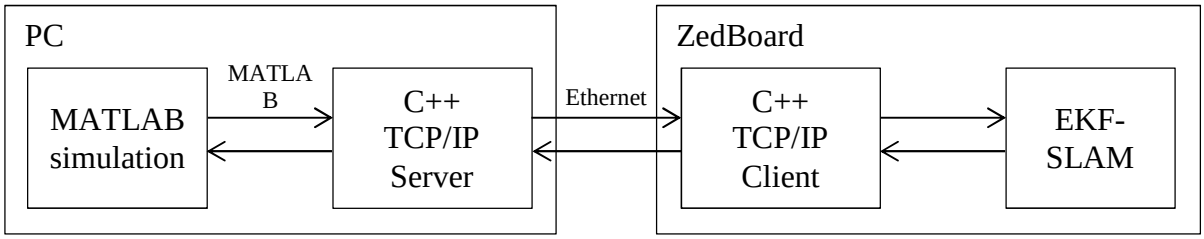


Figure 27: Construction to test the EKF-SLAM C++ port with MATLAB simulator. (Author's figure).

The simulator populates a 2D world with a number of landmarks and adds random noise when each landmark is measured by the sensor. The EKF-SLAM algorithm processes these readings and creates a map. All landmarks are updated to test for the worst-case scenario, where the main EKF-SLAM loop (Figure 8) is profiled and its execution time measured.

The EKF-SLAM system worked in the simulation environment, but as predicted by the complexity of the algorithm (Section 2.5.4), increasing the number of landmark quickly renders the process too slow to be performed in real time due to the dimensionality problem. The results can be seen in Section 6.2.

6. TESTS AND RESULTS

This chapter presents the tests and results performed during this work. On Section 6.1, a generic system for finding landmarks for SLAM from stereo cameras is profiled, and the results showed that the POI detector is the most costly step involved. This lead to the implementation of a hardware module to accelerate the detection, with the corresponding tests being shown on Section 6.3. Section 6.2 shows the profile of the EKF-SLAM implementation running on the embedded system. Section 6.4 evaluates the optimal parameters for the simplified grid algorithm implementation, used for describing POIs, and compares the results when running with these parameters with other descriptors. Finally, on Section 6.5 the power requirement of the running system is measured.

6.1 System Profile

A simple system for finding the correspondences for SLAM was implemented using OpenCV (OpenCV, 2014) to determine which portion of a feature-based stereo correspondence system (Section 4.5) would benefit more from optimizations.

Four elapsed time measurements were taken: image acquisition from both cameras, POI detection, POI description and the comparison between the descriptors for finding correspondences.

The Harris and Stephens corner detector (Harris & Stephens, 1988) was chosen as the POI detector because it has been used as a replacement for SIFT in numerous works due to speed reasons (see Section 4.5), which is relevant for embedded applications.

The BRIEF descriptor (Calonder et al., 2010) was used due to empirical tests showing it as a good performing descriptor for embedded systems (Section 6.4.2).

The correspondence between descriptors is found using the k nearest neighbor (k-NN) algorithm (Fix & Hodges Jr, 1951). K-NN was executed in both sides, with non-symmetrical matches discarded. After that, the two closest matches (2-NN) are compared, and if the ratio between them is less than 1 they are also removed. This eliminates cases where there is ambiguity in the correspondences. Table 2 shows the results for the system profiling.

The images were acquired using the two Logitech C525 cameras (Logitech, 2014), on the VRI laboratory, using a resolution of 640x360. All measurements were made using GNU/Linux's <time.h> library, with the clock_gettime function (Kerrisk, 2015a), using the CLOCK_MONOTONIC source, in the Zedboard's ARM Cortex A9 processor running at the clock of 866MHz. The clock time was sampled before and after each portion of code shown in Table 2.

Table 2: Generic system profile.

Task	Time (ms)
Stereo camera image pair acquisition	53.32
POI Detector	295.33
POI Descriptor	40.01
Correspondence (k-NN)	17.79

Analyzing the data from Table 2, it is clear that the most costly step in a feature-based system for stereo correspondence is the POI detector. For this reason, this step was chosen to be optimized with a hardware co-processor.

6.2 EKF-SLAM Profile on Embedded Hardware

A simple EKF-SLAM implementation was ported to C++, which was shown in Section 5.4, to test if a simple SLAM solution could run in reasonable time on the same hardware as the stereo vision landmark acquisition system.

Table 3 shows the execution time of the SLAM landmark update step, in worst case scenario where the whole map is updated for each landmark. Measurements were made in the Zedboard's ARM Cortex A9 processor running at the clock of 866MHz. They were taken using GNU/Linux's <*time.h*> library, with the *clock_gettime* function (Kerrisk, 2015a), using the *CLOCK_MONOTONIC* source. The clock time was sampled before and after the function, with the results shown.

The Eigen C++ library, used for vector and matrix operations, supports multiple threads using OpenMP. Since the hardware has a dual-core ARM processor, two threads could be run in parallel to speed up EKF-SLAM. The results of this configuration are shown on the third column of Table 3.

Table 3: Increase in execution time of EKF-SLAM update step due to map size, running in the dual-core ARM Cortex A9 processor in ZedBoard.

Landmarks	Time (ms)	OpenMP time(ms)
50	22	20
100	170	130
150	550	500
200	1500	1300

6.3 Harris Hardware Implementation

In Section 6.1, it was determined that the best candidate step for optimization in execution time of a landmark acquisition system that uses a feature-based stereo correspondence approach is the POI detection phase. For this reason, a Harris co-processor was designed in the FPGA portion of the Zynq-7000 SoC, as detailed in Section 5.2. The measurements of the effective speedup in real hardware are shown here in Section 6.3.1.

Section 6.3.2 is focused on ensuring that the quality of the optimizations remains close to the reference implementation in OpenCV.

6.3.1 Execution Time Comparison with OpenCV

The Harris co-processor was designed to process four sections of the image in sequence inside the pipeline (as shown in Section 5.2.3). At each write operation, four pixels are sent to the co-processor simultaneously. At every seventh write operation, a read operation is performed, and four bits are retrieved, signaling if a corner was detected in each of the four sections of the image sent 14 clock periods earlier (as explained in Section 5.2.11).

This architecture was designed to ensure the optimal throughput of the hardware, since the AXI4-lite interface accepts up to 32 bits simultaneously for input and output, which is ensured by sending four pixels simultaneously through it. Since all the operations inside the co-processor are synchronized with the write operations performed by the CPU, the number of write operations has the bigger impact in execution time.

Measurements were made in the Zedboard's ARM Cortex A9 processor running at the clock of 866MHz. They were taken using GNU/Linux's <*time.h*> library, with the

clock_gettime function (Kerrisk, 2015a), using the *CLOCK_MONOTONIC* source. The clock time is sampled before and after running the Harris detector function in OpenCV and in hardware. Table 4 shows the results for the execution time. GNU/Linux caches are dropped before each test. For comparison, the initial hardware version that performed the operation using only a single pixel at a time, with no pipeline gain, was also evaluated (discussed on Section 7.2). The image used on the tests is taken from (Wikipedia, 2015), and the visual output of the corners can be seen in Figure 28.

Table 4: Execution time comparison between the reference (OpenCV) and the hardware co-processor (FPGA). The speedup reached when using the hardware implementation compared to the reference is also shown. Tests were done both for the initial version of the co-processor that didn't use a pipeline (1 pixel at a time), and the final version of the co-processor (4 pixels at a time).

	OpenCV time	FPGA time	Speedup
1 pixel at a time	198ms	343ms	0.58
4 pixels at a time	195ms	94ms	2.07

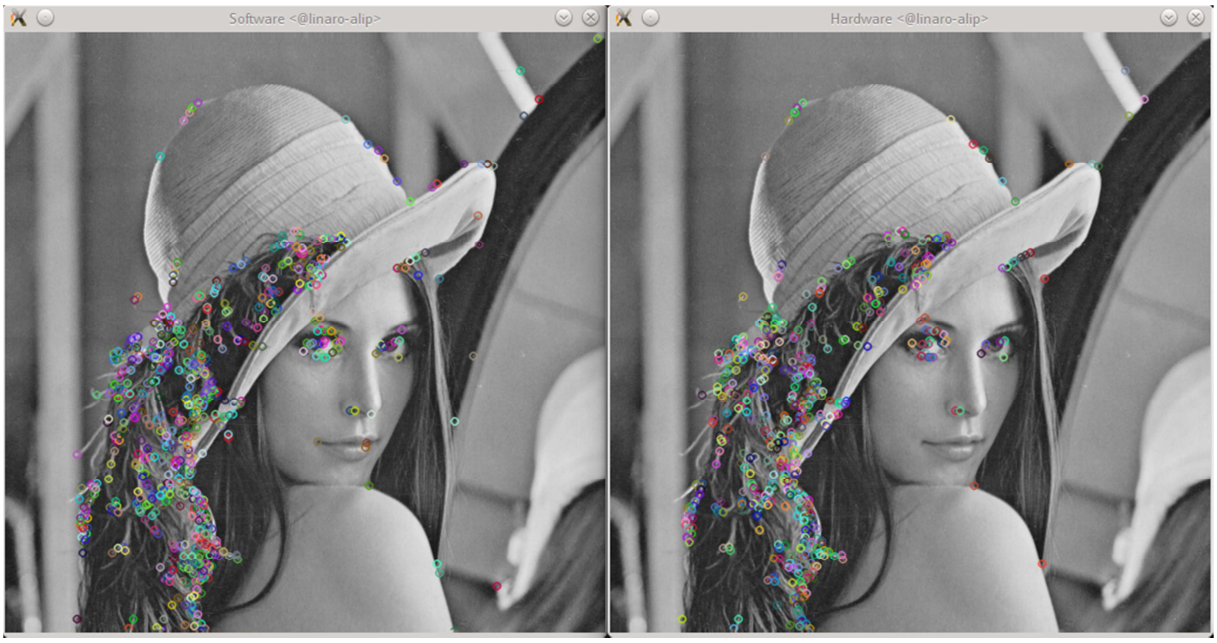


Figure 28: Visual output of corners detected in OpenCV (software) and in the FPGA (hardware) on the test image while measuring the execution time (left and right, respectively). (Author's figure).

The measurements shown in Table 4 include the transfer of the images from the main processor's DDR3 memory to the co-processor, the data processing and the transfer of the results back from the co-processor to the main processor.

Section 6.3.2 shows along with the detection quality comparison the hardware and software execution time for an image sequence.

6.3.2 Quality Comparison with OpenCV

In order to assess how well the hardware implementation of the Harris Algorithm performed in terms of quality, the OpenCV version of the algorithm was taken as a reference (OpenCV, 2014). The KITTI Vision Benchmark Suite dataset (Fritsch, Kuehnl, & Geiger, 2013; Geiger, 2015; Geiger, Lenz, Stiller, & Urtasun, 2013; Geiger, Lenz, & Urtasun, 2012) was selected for providing real world stereo sequential images from a typical SLAM problem. Both the standard OpenCV Harris corner detector solution and the designed FPGA co-processor were used to detect the corners in the image, and the results between them were compared for the whole dataset. Kitty is comprised of 22 image sequences, labeled from 00 to 21. Only sequence number 00 was used from the KITTI dataset. On the dataset, each image in the stereo pair has the resolution of 1241x376 pixels.

In the FPGA implementation, the threshold value is calculated from the previous image pair, as described on Section 5.2.9. The first image pair processed by the co-processor is used for initializing the threshold value, thus the results from corners detected from them are discarded. Results from corners from one image are returned with a 14 clock periods delay due to the time taken to process them in hardware (Section 5.2.11). Thus, the last corners detected from an image are returned to the processors when pixels from the next pair are being loaded into the FPGA. For this reason, the results from the last image pair are incomplete, because there is no next image pair to advance the processing by 14 clock periods, and are also discarded.

All pixels from the image pair are classified in the hardware implementation as true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) in relation with the results from OpenCV, and summed up for the whole image sequence. The metrics recall (Rc), specificity (Sc) and precision (Pc) were obtained from these values according to (6.1).

$$Rc = \frac{TP}{(TP + FN)}$$
$$Sp = \frac{TN}{(FP + TN)} \qquad (6.1)$$
$$Pc = \frac{TP}{(TP + FP)}$$

Additionally, the amount of time spent in software by the OpenCV reference running in the embedded ARM and the amount spent in hardware by the FPGA co-processor were measured for each image pair, and their mean execution time is shown, along with the speedup of FPGA processing relative to the software processing in embedded ARM. This can be compared to the speedup measured in Section 6.3.1. Execution time for the simplified Grid descriptor algorithm (running with $h = 3$ and $g = 16$) was also measured, and can be compared with further tests shown in Section 6.4.2. All measurements were taken with the same procedure described in Section 6.3.1. The results are shown in Table 5.

Table 5: Quality comparison between the results obtained from the designed Harris corner detector co-processor (hardware) and the reference in OpenCV. Additionally, a time of execution comparison is also provided to extend the results from the previous tests in Section 6.3.1 and descriptor execution time for Section 6.4.2.

True Positives	10,437,007
True Negatives	4,224,750,170
False Positives	372,423
False Negatives	380,448
Total Points	4,235,940,048
Recall	0.964830
Specificity	0.999912
Precision	0.965546
OpenCV Time	588ms
Hardware Time	332ms
Speedup	1.77
Descriptor Time	1304ms

6.4 Simplified Grid Algorithm

In this section, the optimal parameters for the simplified Grid algorithm are determined (6.4.1), and then compared with other commonly used descriptors (6.4.2).

6.4.1 Optimal parameters

The first test was performed with simplified Grid Algorithm (Section 5.3.2) with the intention of determining the optimal parameters that could be used for matching the POIs. These parameters consist of the grid size g , which defines the future length of the generated descriptor, and the factor of reduction of the resolution 2^h (as described on last paragraph of Section 5.3.2), which defines how loose is the grid used to create the pattern seen on Figure 26.

The input POIs can be obtained by using many different feature detectors available on OpenCV common interface of feature detectors (OpenCV, 2015), such as SIFT and Harris. The simplified Grid Algorithm then creates a descriptor for each point using only the information of the star pattern of its surrounding POIs.

For this initial test, two databases consisting of different stereo scenes were used to avoid training the selected parameters biased to a specific database.

The *IRCCyN IVC Quality Assessment Of Stereoscopic Images database* (Callet, 2010) is comprised of six stereo image pairs, each with 15 derived image pairs with added noise. Only the six undistorted image pairs were used on this test. All images have the size of 512x512 pixels. Both the OpenCV SIFT and Harris detectors were used to find POIs on the image. Specifically to the SIFT detector, two tests were run: one with the points limited to the 512 best points according to the response parameter, and another without such limitation. The points are then described through the simplified Grid Algorithm descriptor. POI pairs that don't pass the k-NN cross-check and 2-NN ratio test for ratio<1 are discarded in order to remove POIs that have ambiguous correspondences in the other image. The resulting points are then checked to respect the epipolar constraint, and those that comply are considered to be **correct matches**. The results for detected matches and correct matches are summed for all 6 pairs of images, to avoid the descriptor to be partial to any specific image pair on the set. The results can be seen in Figure 29,

Figure 30 and Figure 31. The SIFT descriptor (not to be confused with the SIFT detector) is used here as a reference, and the detected matches are also discarded when not passing the k-NN cross-check and the 2-NN ratio test of 0.8, as recommended by the author in (Lowe, 2004).

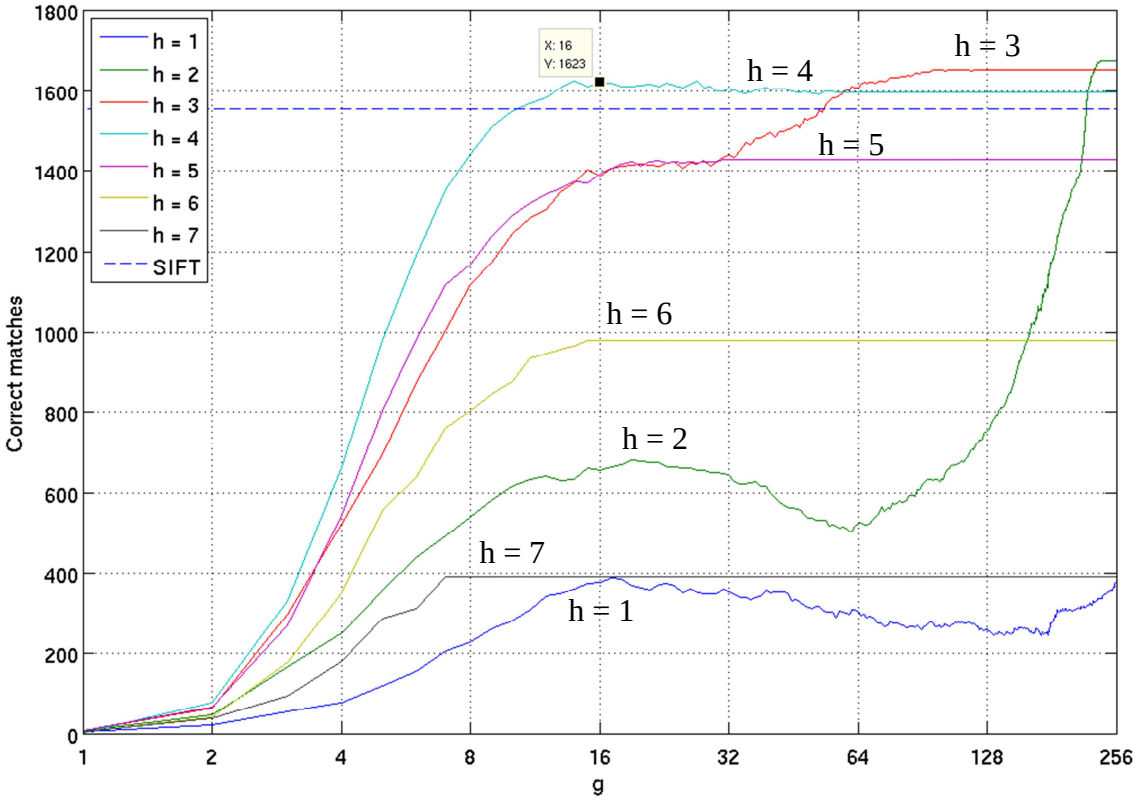


Figure 29: Correct matches for the simplified Grid Algorithm descriptor for the IRCCyN IVC Quality Assessment Of Stereoscopic Images database (Callet, 2010), using the SIFT detector limited to 512 points. The dashed horizontal line represents the correct matches found with SIFT as a reference for comparison. (Author's figure).

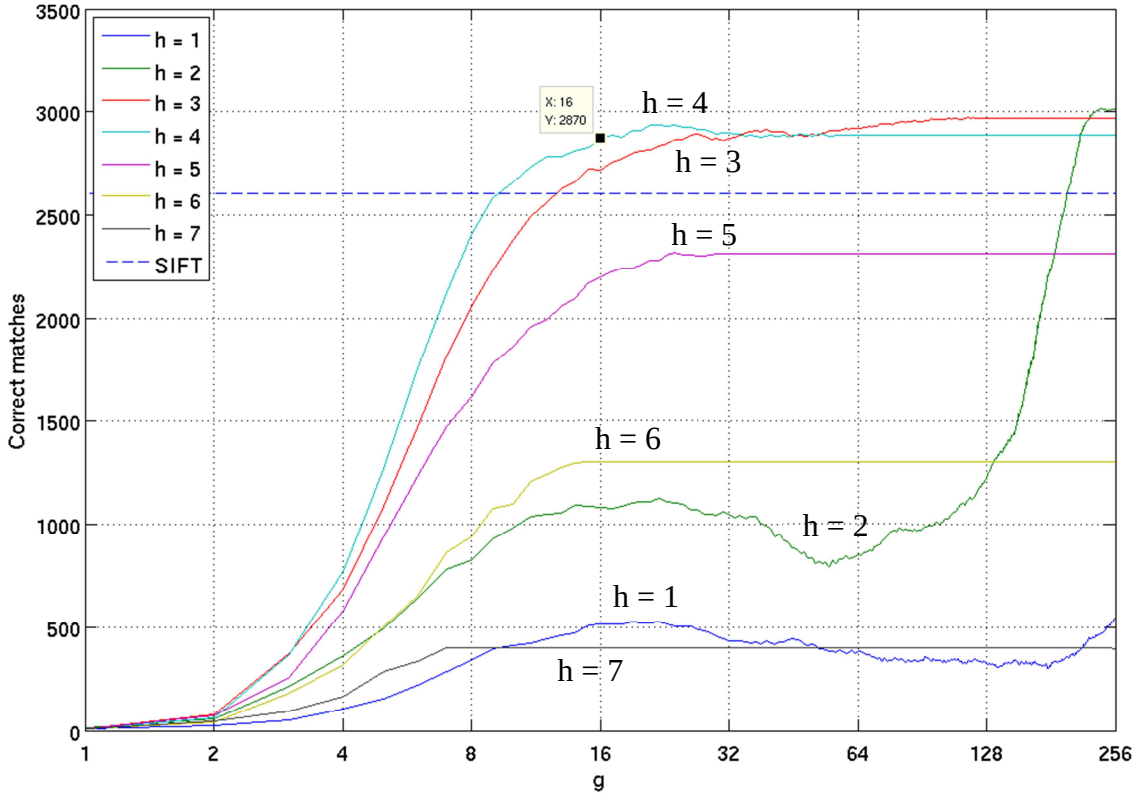


Figure 30: Correct matches for the simplified Grid Algorithm descriptor for the IRCCyN IVC Quality Assessment Of Stereoscopic Images database (Callet, 2010), using the SIFT detector (no limits). The horizontal line represents the correct matches found with SIFT as a reference for comparison. (Author's figure).

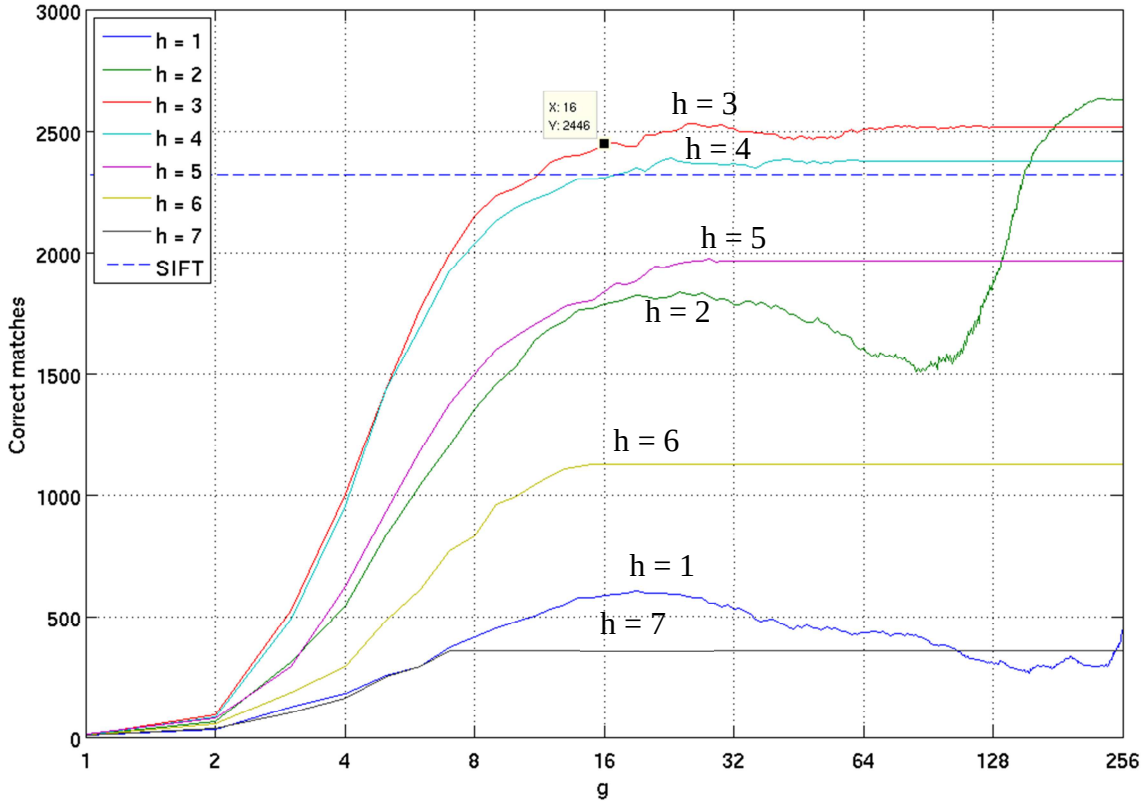


Figure 31: Correct matches for the simplified Grid Algorithm descriptor for the IRCCyN IVC Quality Assessment Of Stereoscopic Images database (Callet, 2010), using the Harris detector. The horizontal line represents the correct matches found with SIFT as a reference for comparison. (Author's figure).

The descriptor size in bits is g^2 (the grid is a $g \times g$ square), therefore low values of g are desirable. For the 512x512 images with the SIFT detector, an apparent good candidate would be to choose $h = 4$ and $g = 16$. For g values higher than the selected, the size of the descriptor increases significantly comparing to the benefit gained in the number of correct matches. The h value was simply chosen for having the highest number of correct matches when $g = 16$.

Similarly, for the Harris detector, a good candidate would be $h = 3$ and $g = 16$. In both cases this would present a descriptor 16 times smaller than SIFT while achieving similar correct matches (Table 6). Here h represents how lower is the resolution of the grid used in relation to the image resolution, which is reduced by a factor of 2^h.

Table 6: Comparison between the simplified Grid Algorithm and SIFT for the IRCCyN IVC Quality Assessment Of Stereoscopic Images database (Callet, 2010) .

Descriptor	Detector	Correct Matches	All Matches	Ratio	Size
Grid (g=16, h=4)	SIFT, 512 pts.	1623	1624	0.999	32 bytes
Grid (g=16, h=4)	SIFT	2870	2870	1.000	32 bytes
Grid (g=16, h=3)	Harris	2446	2450	0.998	32 bytes
SIFT	SIFT, 512 pts.	1555	1564	0.994	512 bytes
SIFT	SIFT	2605	2622	0.993	512 bytes
SIFT	Harris	2322	2325	0.999	512 bytes

The same steps were applied to the Middlebury 2006 Stereo Dataset (Scharstein, 2006), comprised of 21 pairs of stereo images, where images range between the sizes of 413x370 and 465x370. Results can be seen in Figure 32, Figure 33 and Figure 34.

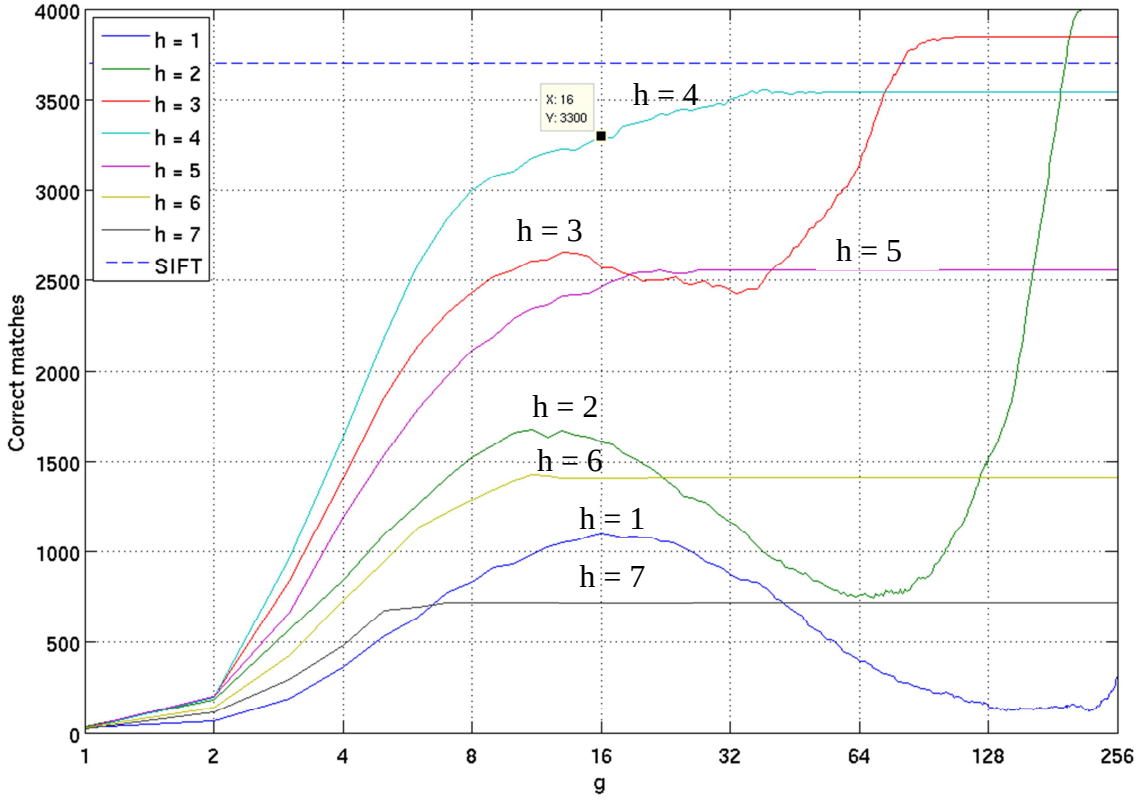


Figure 32: Correct matches for the simplified Grid Algorithm descriptor for the Middlebury 2006 Stereo Dataset (Scharstein, 2006), using the SIFT detector limited to 512 points. (Author's figure).

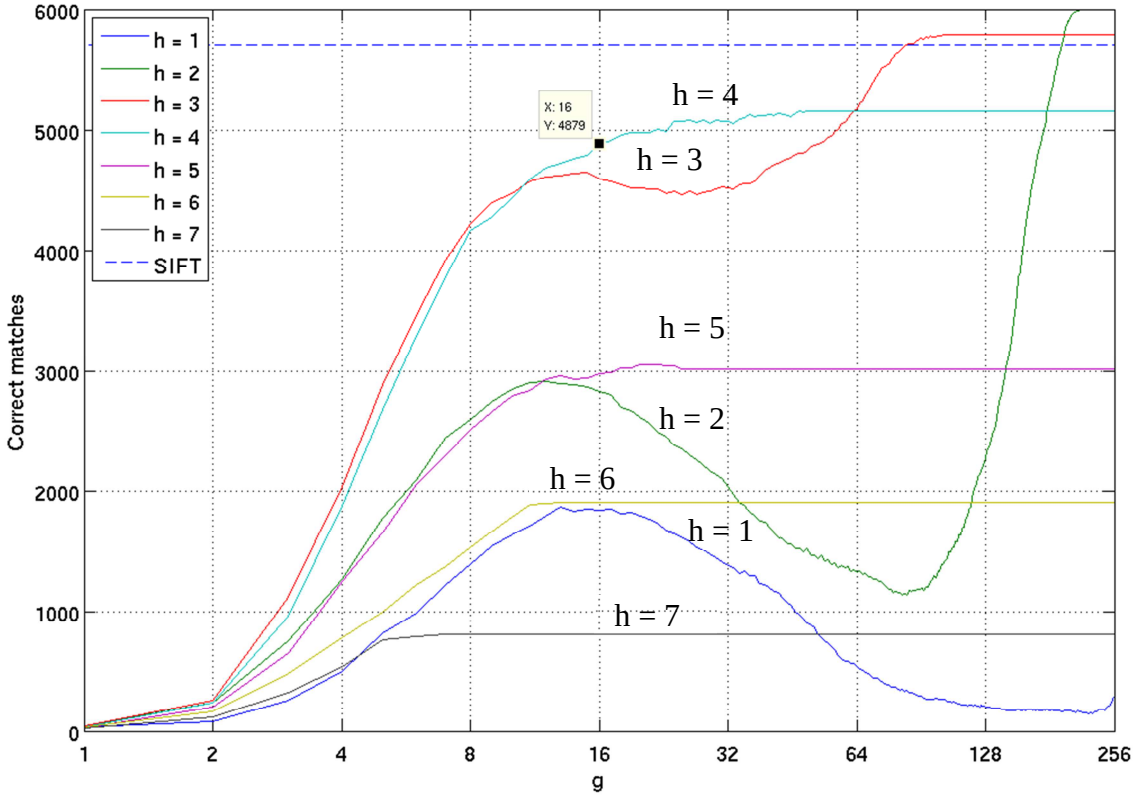


Figure 33: Correct matches for the simplified Grid Algorithm descriptor for the Middlebury 2006 Stereo Dataset (Scharstein, 2006), using the SIFT detector. (Author's figure).

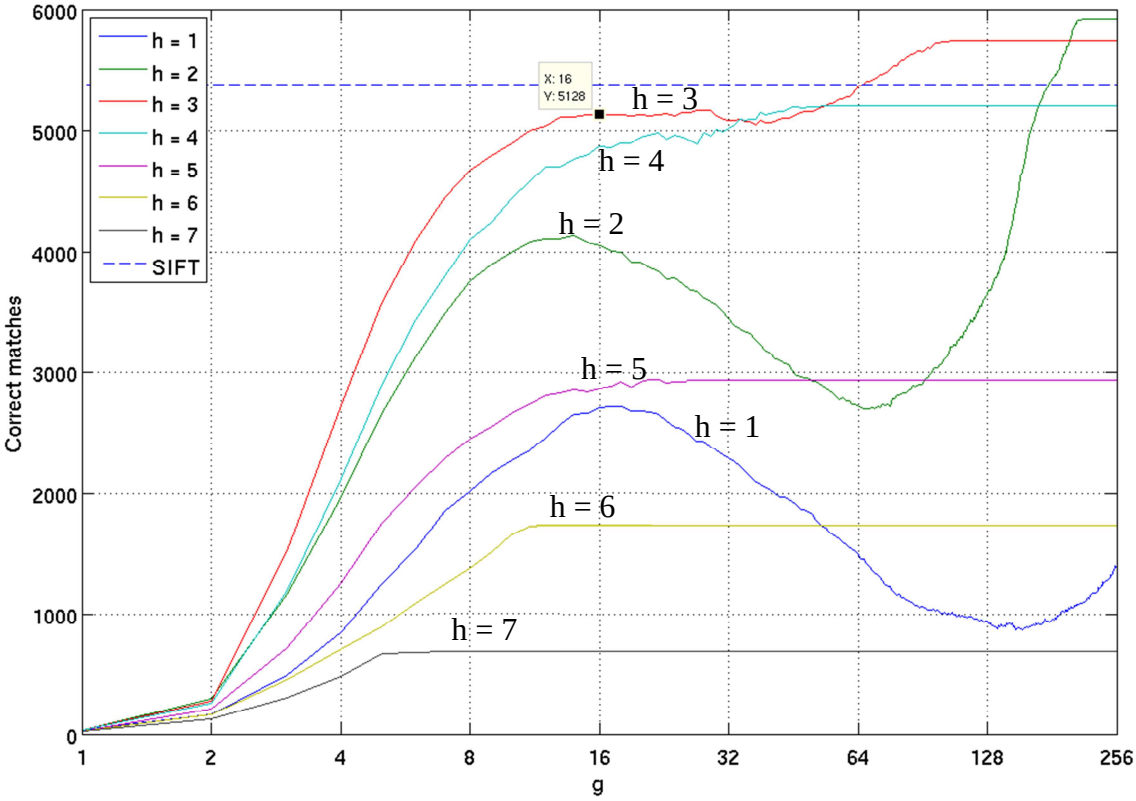


Figure 34: Correct matches for the simplified Grid Algorithm descriptor for the Middlebury 2006 Stereo Dataset (Scharstein, 2006), using the Harris detector. (Author's figure).

The same optimal parameters found in the previous test can be used with this database. The performance of the descriptor for the Middlebury 2006 Stereo Dataset is shown in Table 7.

Table 7: Comparison between the simplified Grid Algorithm and SIFT for the Middlebury 2006 Stereo Dataset (Scharstein, 2006).

Descriptor	Detector	Correct Matches	All Matches	Ratio	Size
Grid (g=16, h=4)	SIFT, 512 pts.	3300	3353	0.984	32 bytes
Grid (g=16, h=4)	SIFT	4879	4964	0.983	32 bytes
Grid (g=16, h=3)	Harris	5128	5177	0.991	32 bytes
SIFT	SIFT, 512 pts.	3701	3750	0.987	512 bytes
SIFT	SIFT	5702	5773	0.988	512 bytes
SIFT	Harris	5379	5398	0.996	512 bytes

Analyzing the graphs from Figure 29 to Figure 34, we can see that the curves for low values of $h = 1$, $h = 2$ and, to some extent, $h = 3$ have a sweet spot around $g = 16$. Up to this point there is an increase in the number of correct matches, that only get high again on much larger values of the grid size g. So, when reducing slightly the resolution of the grid (by a factor of 2^h), increasing the area of the descriptor gets more correct matches only up to a certain point. This means that points close to the chosen reference have a higher probability of having low disparity when comparing to the ones further from it. This effect diminishes for higher values of h, because the looseness of the grid allows points with larger disparities to still fall within the same cell in the descriptor. On the other hand, increasing too much the h value reduces the useful information available on the descriptor beyond the needed for finding proper correspondences. The difference in the shapes of the curves when comparing the two datasets is related to the changes in image size and in the content of the images. Also, the number of images on the IRCCyN dataset is relatively small when compared to the Middlebury dataset, leading to outliers being more pronounced on the accumulated results.

Table 8 reorganizes the values from Table 6 and Table 7, showing the difference in performance between the proposed descriptor and SIFT. For each database, the POIs were found using three detectors (SIFT limited to the 512 points, SIFT with all points and Harris). The correct matches for the simplified Grid Algorithm and SIFT are shown in 3rd and 4th columns with the ratio between them shown on last.

Table 8: Performance ratio between the simplified Grid Algorithm and SIFT for the IRCCyN IVC Quality Assessment Of Stereoscopic Images database (Callet, 2010) and the Middlebury 2006 Stereo Dataset (Scharstein, 2006), here database 1 and 2, respectively.

	Detector	Grid descriptor	SIFT descriptor (ref.)	Ratio
Database 1	SIFT, 512 pts.	1623	1555	1.044
	SIFT	2870	2605	1.102
	Harris	2446	2322	1.053
Database 2	SIFT, 512 pts.	3300	3701	0.892
	SIFT	4879	5702	0.856
	Harris	5128	5379	0.953

There was no significant difference on pairing the Harris detector or the SIFT detector with the simplified Grid descriptor on both databases. The smaller complexity of the Harris algorithm is more interesting for embedded systems, weighting for its choice within the application of this work. When using Harris as the detector, the SIFT descriptor performs better in both databases.

Since the resolution of the images acquired from the webcams used is 640x360, due to limitations in the USB 2.0 bandwidth in ZedBoard (see section 5.2.1), and the images from both databases are relatively close to this resolution, the parameters found in these tests could be considered as valid for this implementation. However, it is not clear which parameters should be used with images of very different resolutions. To answer this question, a dataset from Disney Research (Gross, 2012) that provides images with a much higher resolution, 21 megapixel, was used.

This dataset comprises 5 different scenes with 101 or 151 pictures taken with different separations. Only the extreme images were selected, and resized using the *convert* utility of the *ImageMagick* program version 6.5.4-7 (ImageMagick Studio LLC, 2015), without preserving the aspect ratio, to the resolutions of 320x200, 640x400, 960x600, 1280x800, 1600x1000, 1920x1200, 2240x1400 and 2560x1600. On each pair of images, the POI were found using the Harris detector from OpenCV, then described by the simplified Grid Algorithm with different parameters for the grid size g, and the factor of reduction of the resolution 2^h. Again, POI pairs that didn't pass the k-NN cross-check and 2-NN ratio test for ratio<1 were discarded. The resulting points were then checked with respect to the epipolar constraint, and those that comply are considered to be correct. The results for detected matches and correct matches are summed for all 5 pairs of images from the same

resolution, to avoid the descriptor to be partial to any specific image pair on the set. The results can be seen from Figure 35 to Figure 42.

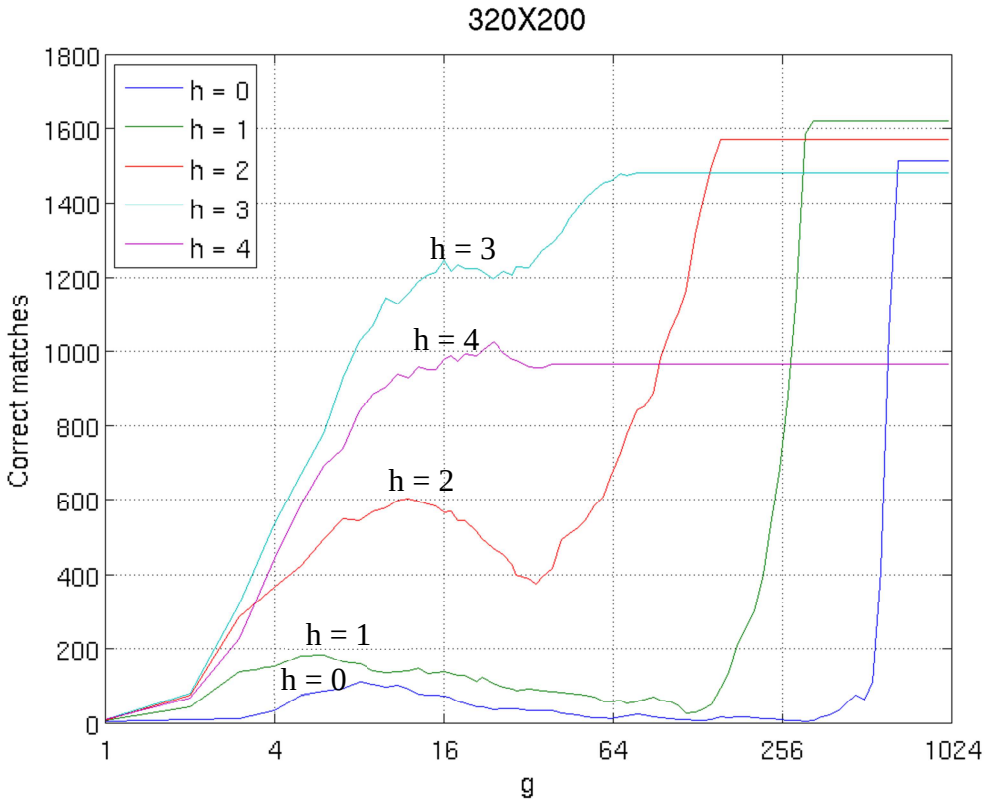


Figure 35: Correct matches for each used parameters for the 320x200 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

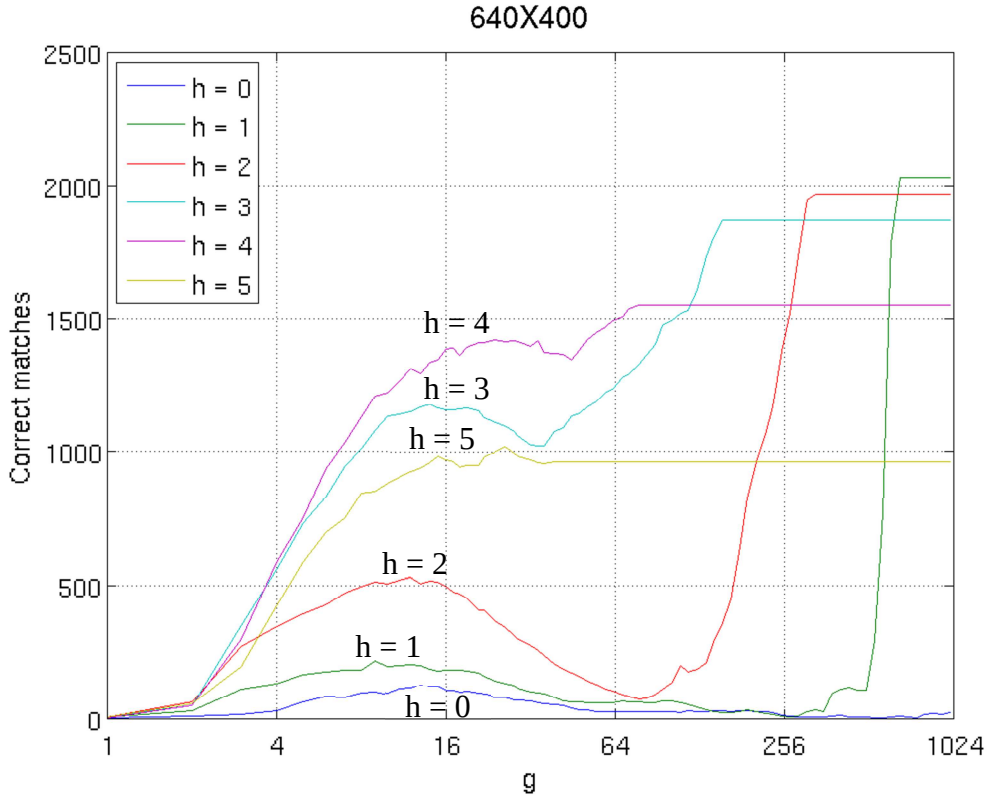


Figure 36: Correct matches for each used parameters for the 640x400 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

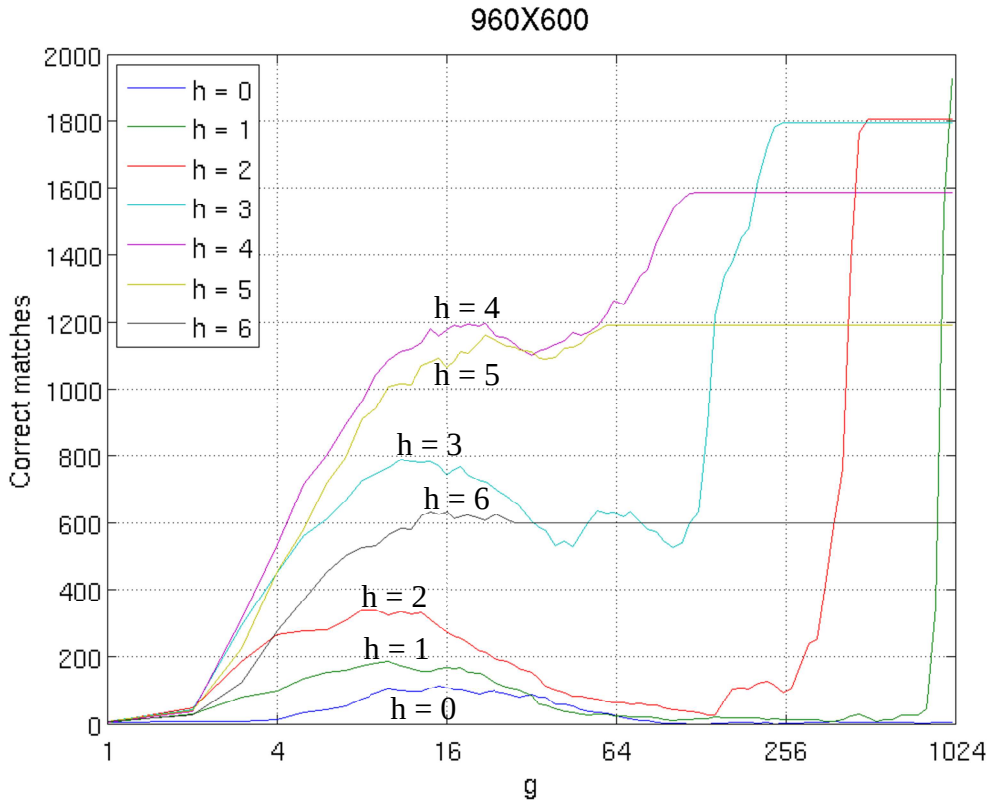


Figure 37: Correct matches for each used parameters for the 960x600 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

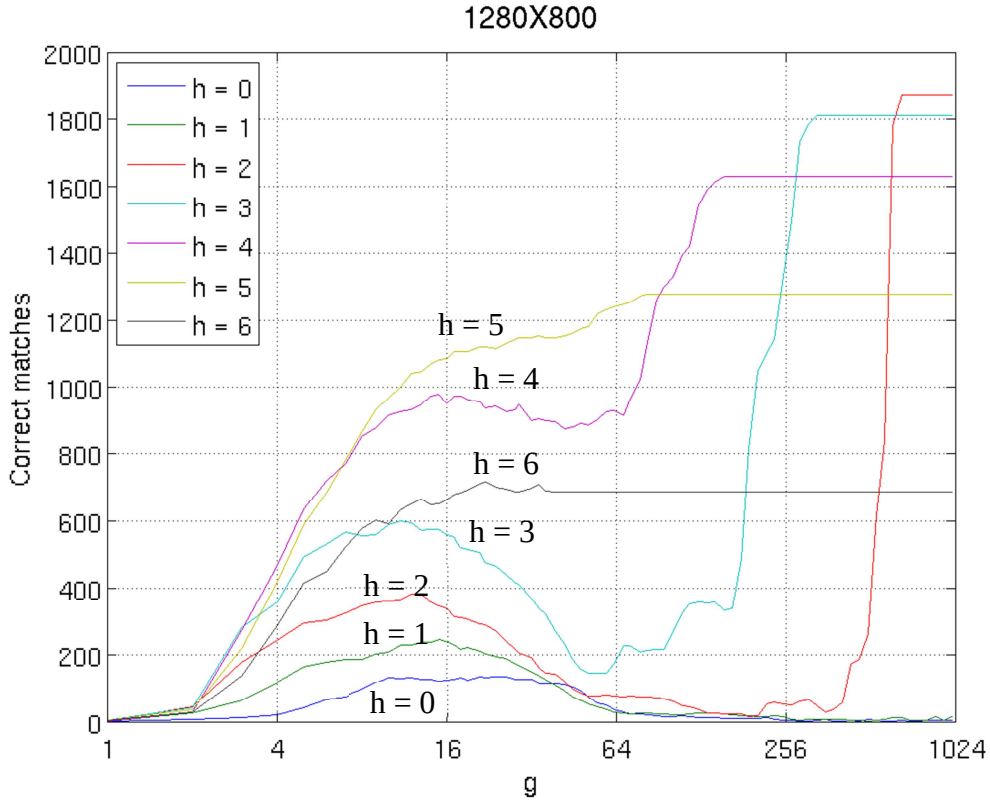


Figure 38: Correct matches for each used parameters for the 1280x800 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

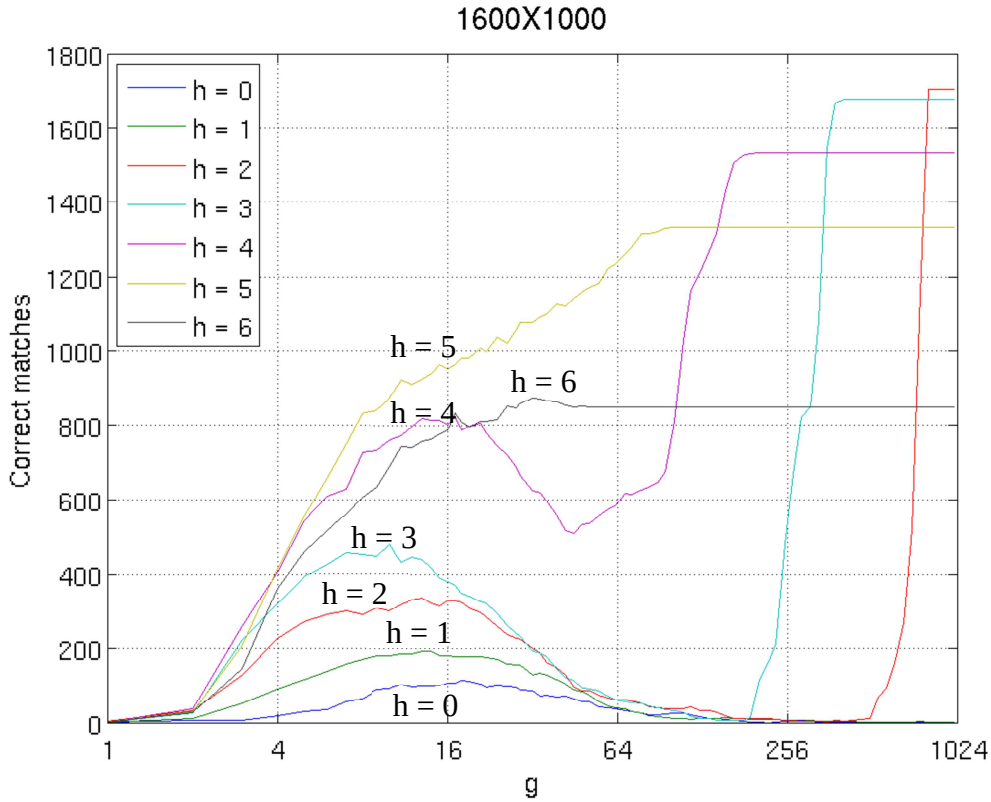


Figure 39: Correct matches for each used parameters for the 1600x1000 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

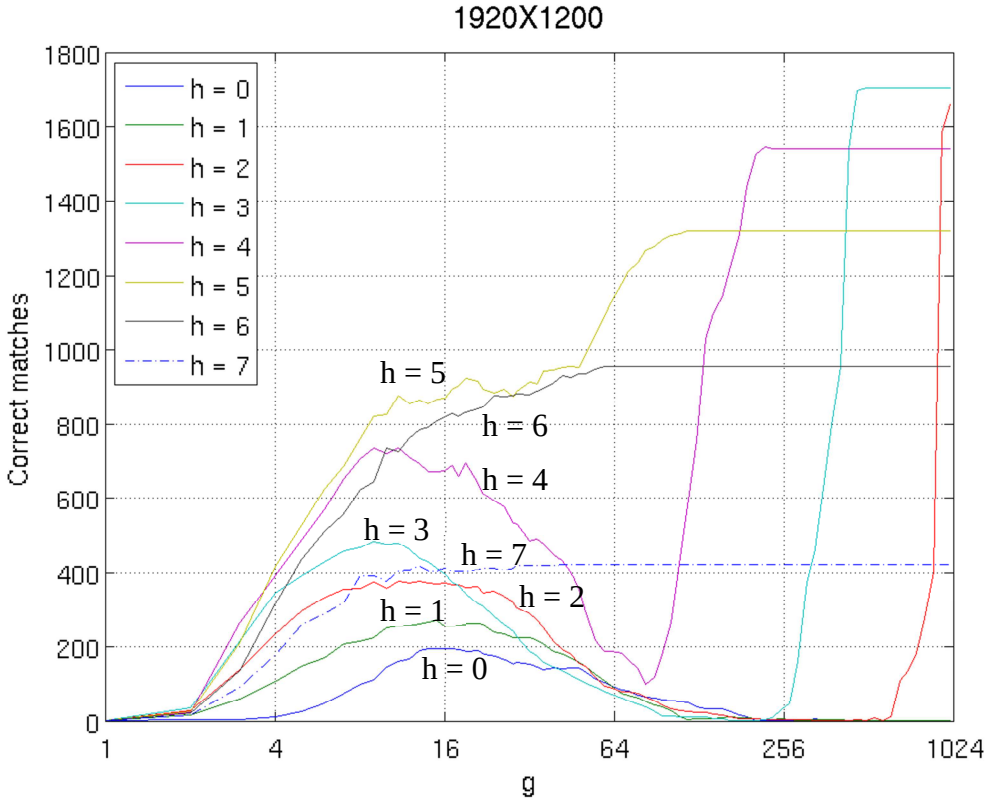


Figure 40: Correct matches for each used parameters for the 1920x1200 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

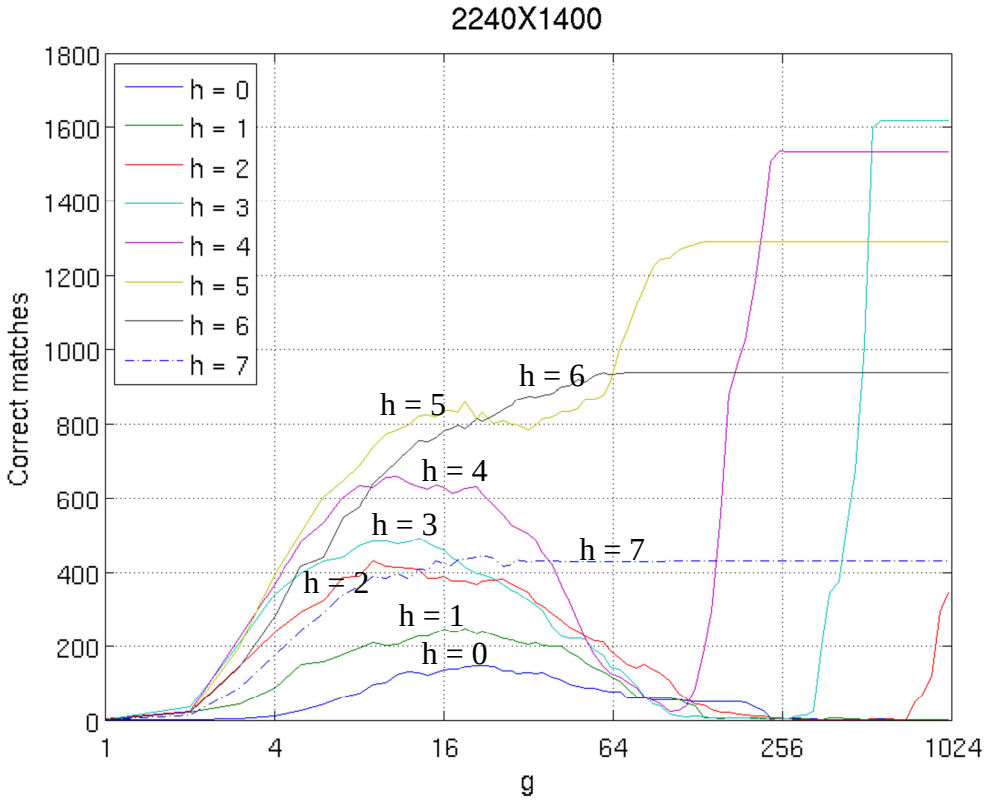


Figure 41: Correct matches for each used parameters for the 2240x1400 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

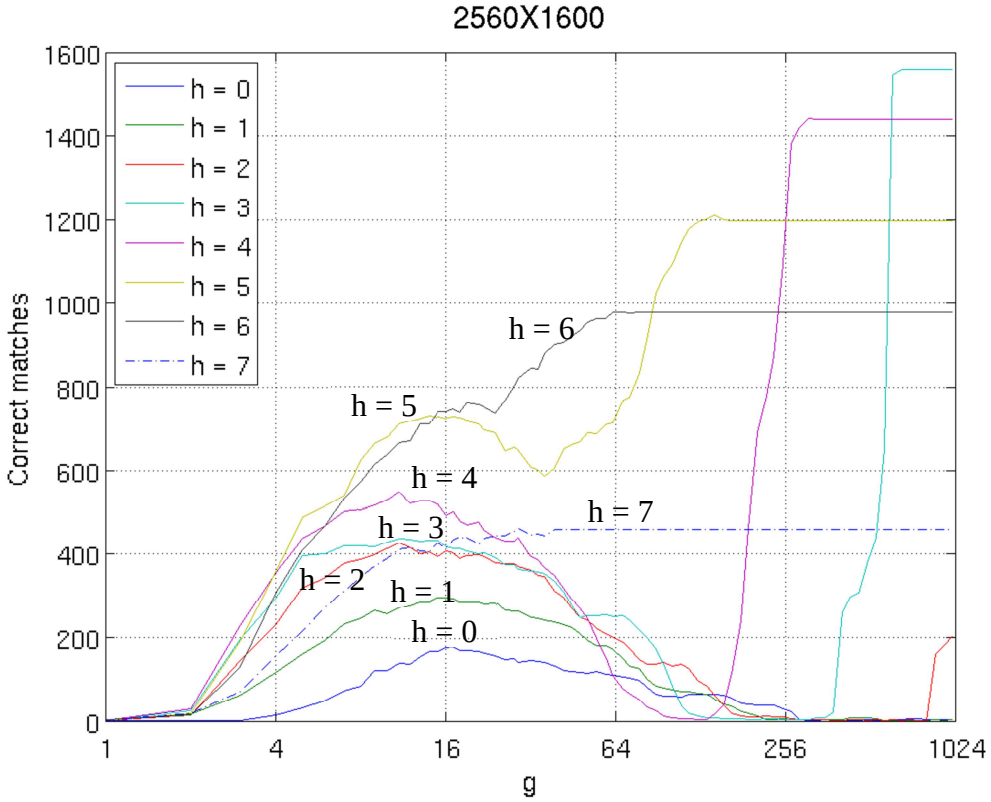


Figure 42: Correct matches for each used parameters for the 2560x1600 images of the Disney Dataset (Gross, 2012). An increase in the *h* parameter means a reduction on the grid resolution, while an increase in the *g* parameter means an increase on the grid square size, increasing also the descriptor lenght. (Author's figure).

With the curves from Figure 35 to Figure 42 it is possible to find the optimal parameters for images of different sizes. As the image size increases, higher h values start producing progressively better results, with higher h values implying a looser grid being used to describe the points. Since the size of the descriptor only depends on the g parameter, the size of the descriptor can remain constant, with only the resolution of the grid being adjusted accordingly to the image size.

It is interesting to note that the number of useful POIs with correct matches is the highest when using images close to 640x400 pixels. The reduction in matches is related to the way the Harris algorithm is implemented on OpenCV, where the size of the masks for the Sobel and block filters and the size for the non-maximum suppression are fixed and more properly adapted for images around this range.

6.4.2 Execution Time Analysis

For this test, the two databases mentioned in the first two tests from section 6.4.1 were again used to determine the execution time of the simplified Grid Algorithm, while comparing to different descriptors available on OpenCV: SIFT (Lowe, 2004), SURF (Bay et al., 2006), BRIEF (Calonder et al., 2010), BRISK (Leutenegger et al., 2011), ORB (Rublee et al., 2011) and FREAK (Alahi et al., 2012).

The execution time includes the extraction of the descriptors and the matching with the k-NN algorithm. The detection of the POIs was performed with the Harris Corner Detector algorithm present in OpenCV. The time spent on the detection was not measured on this test, but was measured earlier and can be found in Sections 6.3.1 and 6.3.2.

Measurements were made in the Zedboard's ARM Cortex A9 processor running at the clock of 866MHz with 512KB of L2 cache. They were taken using GNU/Linux's *<time.h>* library, with the *clock_gettime* function (Kerrisk, 2015a), using the *CLOCK_MONOTONIC* source. The clock is sampled just before running the descriptor function and just after finishing the k-NN matching, and the difference of time is shown in Table 9 and Table 10. For comparison the same process is executed in an Intel Core 2 Quad Q9550 processor, with 2.83 GHz and 12 MB of L2 cache. Correct matches are also shown for the selected descriptors.

Table 9: Correct matches and time of execution of descriptors for the IRCCyN IVC Quality Assessment Of Stereoscopic Images database (Callet, 2010).

	Mean correct matches	ARM mean time (ms)	Intel mean time (ms)
GRID	407.67	52.06	9.61
SIFT	387.00	1256.35	92.19
SURF	90.50	1183.40	59.65
BRIEF	388.00	85.18	5.57
BRISK	399.67	11623.80	1829.64
ORB	379.83	178.17	7.58
FREAK	316.33	1593.51	217.66

Table 10: Correct matches and time of execution of descriptors for the Middlebury 2006 Stereo Dataset (Scharstein, 2006).

	Mean correct matches	ARM mean time (ms)	Intel mean time (ms)
GRID	244.19	30.97	6.77
SIFT	256.14	776.64	58.65
SURF	129.67	836.27	41.64
BRIEF	208.33	43.49	3.59
BRISK	248.62	11604.40	1647.11
ORB	201.95	108.75	5.30
FREAK	186.05	1574.38	218.51

By analyzing the data from Table 9 and Table 10, it is clear that using the proposed algorithm brings a speedup comparing to the most efficient descriptor, BRIEF (Calonder et al., 2010), while being executed in the ARM Cortex A9 available in ZedBoard in both databases, with the tasks being performed in 0.61 and 0.71 times the best performing descriptor. On the other hand, on the Intel Core 2 Quad Q9550 processor it performed slower, taking 1.72 and 1.88 times the best performing descriptor. In both databases the number of correct detected matches is very close to the best performing descriptors.

6.5 Power Requirement

ZedBoard has a current sensing resistor of $10m\ ohm$ (R292), accessed through the J21 pins, which is used to determine the total current draw of the board and peripherals attached to the USB interface. The RMS voltage on R292 was measured using an OS-2062C oscilloscope from ICEL, and the value was determined to be $11.05mV$. By Ohm's law, the current can be calculated as shown on eq. (6.2). The measurements were done with the complete system running, using the FPGA Harris co-processor and the simplified Grid algorithm.

$$I = \frac{V}{R} = \frac{11.05mA}{10m\ \text{ohm}} = 1.105A \tag{6.2}$$

The power supply voltage, rated at 12V, was measured for greater accuracy on the calculations and found to be running at 11.93V. Thus, the total power can be determined using eq. (6.3).

$$P = V \cdot I = 11.93 \cdot 1.105 = 13.18W \tag{6.3}$$

This value is within the limits of the power supply included with ZedBoard, which is able to supply up to $3A$ at $12V$, leading to a maximum total power of $36W$.

7. DISCUSSION

This chapter presents some observations, problems and possible solutions related to the construction of the landmark acquisition system for Visual SLAM. Section 7.1 is concerned with the Visual SLAM problem, while Section 7.2 discusses the hardware used in the implementation of the landmark acquisition system. Finally, Section 7.3 relates to the implementation of the Star-ID based descriptor for stereo correspondence.

7.1 Visual SLAM

When feature detection was applied to the stereo image datasets used during the tests, using the default parameters, hundreds of landmarks were found, on average (Table 9 and Table 10). In contrast, the simulations of EKF-SLAM on the dual-core embedded ARM processors shown that a parallel implementation using OpenMP reached the limits of real time processing for a map of around 200 landmarks.

To reduce the number of detected landmarks within a single stereo frame, the adaptive threshold ratio could be tweaked (Section 5.2.9), and only corners with a high response value would be detected as such by the algorithm. The use of an EKF-SLAM algorithm with smaller local maps could improve the total number of landmarks, as discussed previously on Section 2.4.1. Particle Filter SLAM solutions that have linear complexity relative to the number of landmarks, such as FastSLAM, could also be used, as presented on Section 2.4.2.

7.2 Using SoC processors with embedded FPGA fabric

Implementation of a single-purpose soft co-processor that aids executing a specialized task by a standard processor showed that its time of execution could be significantly reduced when compared with the time taken when using the processor alone.

The main challenge of implementing a hardware solution in VHDL is the increased time and difficulty when compared with equivalent solutions written in software languages

used in embedded systems like C. This manifests itself in the need to seek optimizations to reduce the required logic so that the required number of logic elements used are within the limitations of current FPGAs. In this work, special care was taken to reduce the number of multipliers used within code and avoid spending unnecessary flip-flops to achieve the solution.

Even if the hardware itself can be implemented within the limits of logic elements in the FPGA used, there is no guarantee that it can be run within timing constraints. Time propagation of logic signals can't exceed the clock period used in worst case scenarios. This limits the number of cascaded operations that can be done in a single clock period, increasing the number of clock cycles needed to produce a result. FPGA clock frequencies are much lower than modern processors, in the particular case of this work, 100 MHz compared to 866 MHz.

On the first complete solution, there was only a single pixel being sent at once during I/O from the main processor to logic. Thus, there was no multiplexed input converter, and only one 7x7 matrix was loaded into the FPGA. After running the execution time tests with this circuit, it took almost twice the time to solve a circuit than the software reference in OpenCV (Table 4), even including optimizations such as avoiding using a device driver for AXI-4 communication and mapping the register directly into the software process virtual memory to speed-up copying (as discussed on Section 5.1.6 and 5.2.2).

The transactions using memory mapped addresses between the hard processor and the soft co-processor within the AXI4 interface were found to be the main factor that contributes to speed limitations for this particular architecture. Increasing the throughput on I/O transactions to use all the 32 bits available would improve significantly the performance of the co-processor.

The initial approach to achieve this was synthesizing four of the circuits designed at the first complete solution, what in practice would result in a four-core co-processor architecture, being able to process 4 pixels simultaneously without any significant change in code and use all the 32 bits available in the interface. In simulations this approach was successful, but in practice it consumed a number of logic cells superior to the available number on the FPGA. Only a dual-core solution was possible, but the speedup was only enough to get even with the reference.

A naive solution would be to buy a larger FPGA that could fit the whole circuit, as there are available chips that conform to the needs of this circuit. Unfortunately, there were no commercial boards featuring a Zynq-7000 SoC with larger FPGAs available that could fit the four co-processors, only one that could theoretically fit a three core design was available.

A more elegant solution to this problem was to send the four pixels simultaneously to the board, creating four 7x7 matrices for processing and exploiting the fact that each block on the design was initially only active for a single clock period and would be inactive for other 6. By allocating tasks to be done in their inactive time, the four 7x7 matrices could be processed in series in a pipeline configuration, thus getting in practice almost the same speedup gain from a quad-core co-processor using a single one pipelined. This was achieved by multiplexing the input converters as shown in Section 5.2.3. The only disadvantage is that the pipeline postpones the output by a few clock periods when comparing to the quad-core solution.

Use of local caching within BRAM modules was avoided due to the low quantity available on the particular FPGA model used. However, the use of a smaller image size could potentially enable caching to be used and reduce I/O transactions and, consequently, the execution time of the hardware implementation even further. This would require significant changes in the architecture used, mainly in the requirement of a special memory controller to be designed.

The power consumption of the SoC systems with an embedded FPGA (measurements can be seen on Section 6.5) is one of its greatest advantages when compared to using a standard personal computer alone or with additional GPGPU hardware, where power consumption is much higher. An estimative can be seen on (University of Pennsylvania, 2013).

7.3 Star-ID based descriptor

The prototype of a descriptor aimed for stereo correspondence in SLAM showed that the visual two-dimensional spatial information between Points of Interests can be extracted successfully using approaches adopted from Star Identification applications, and that this information can be used to successfully solve the correspondence problem. This feature was

used alone in this work to find the correspondence, but it could be integrated with existing descriptors to reduce errors.

8. CONCLUSION

This chapter presents the contributions from this work and the suggestions for improvements and further research that could be accomplished with future work.

8.1 Contribution

This work produced two significant contributions for Visual SLAM applications. The first was the proposition that treating Point of Interests as stars, and restricting the information available only to their two-dimensional spatial positions could be enough to solve the correspondence problem.

A prototype of such a descriptor, based on techniques already used for Star Identification problems, showed that this approach can indeed be used to solve the correspondence problem, and that the performance in practical applications is comparable to the best performing descriptors used for this purpose in both speed and number of useful correspondences. In the practical tests, it was shown to produce a speedup of 1.63 and 1.40 in two different databases on an ARM Cortex A9 processor when compared to the fastest descriptor analyzed, being especially interesting to embedded applications.

The second significant contribution was a practical analysis of a SoC architecture that incorporates an FPGA with an embedded ARM processor for Visual SLAM applications. Within the tasks of acquiring landmarks for SLAM with a visual system based on stereo cameras, the slowest performing task for feature-based correspondence in software was shown to be the detection of POIs. The reimplementation in hardware of this task, being solved using the Harris and Stephens corner detection algorithm, showed a significant execution time improvement over software, around half the time needed for the software reference.

8.2 Difficulties Found

Two main difficulties have arisen during the work. The first difficulty is related to the fact that the VRI research group, where this work was done, is relatively young. This restricted the research to the foundation levels of a Visual SLAM system, mainly the landmark acquisition system, as most basic software tools and libraries for vision, camera calibration and hardware interface had to be developed before they could be used.

The second difficulty is relative to the SoC with FPGA platform used, Zynq-7000 on ZedBoard, being released on June 2012, while this work started 3 months later on October. The documentation provided by Digilent (Digilent Inc., 2013) presented enough guidelines to allow for the use of the hardware, but overall knowledge about it was scarce or had to be adapted from other FPGAs. The situation today is very different, the community use of the board increased the information available about it, and Xilinx improved and released many tools that make the use of Zynq-7000 systems simpler, such as high level synthesis support on the Vivado tools, with its use was documented on book form (Crockett, Elliot, Enderwitz, & Stewart, 2014). Later on March 2015 Xilinx introduced the SDSoC development environment (Xilinx Inc., 2015) that expands the capabilities of High Level Synthesis. With these tools, it is possible to develop software in C or C++ and optimize portions of it directly on FPGA hardware without translating them to VHDL, using direct compilation, which could potentially lead to a much shorter development time.

8.3 Future work

This work completed the development of the landmark acquisition task needed for Visual SLAM. For a full Visual SLAM implementation, both efforts to increase number of landmarks that could be stored within the map and to limit the number of landmarks acquired from a single image to a smaller number needs to be done, since the EKF-SLAM implementation used within this work couldn’t solve the SLAM problem in real time for a large number of landmarks.

One of the features of EKF-SLAM is that it enables the fusion of different sensors. The fusion of a stereo camera setup with different sensors, such as lasers, could enable rich

information to be used coming from the cameras while working with a much larger range available for the laser sensors.

Only a single descriptor was repurposed from Star Identification to the Stereo Correspondence problem. The positive results from this work indicate that other solutions could also potentially be repurposed in the same fashion.

The Harris and Stephens co-processor developed for FPGA in this work didn’t take advantage from using small buffers to avoid copying the same data from the main memory when advancing lines with the sliding window. This could potentially increase the performance as the main limitation to the architecture is from I/O transactions.

APPENDIX A: MATHEMATICA SCRIPT FOR CALCULATING THE OPTIMAL BITSIZE FOR INDIVIDUAL BLOCKS FROM HARRIS CO-PROCESSOR

```
(*
Sobel x and y:
*)
bitSize0 = 8;
max0 = 2^bitSize0 - 1;
Sx = {{-1 a11, 0 a12, 1 a13}, {-2 a21, 0 a22, 2 a23}, {-1 a31, 0 a32, 1 a33}};
Sy = {{1 a11, 2 a12, 1 a13}, {0 a21, 0 a22, 0 a23}, {-1 a31, -2 a32, -1 a33}};
Ix := Total[Total[Sx]];
pixels = {a11, a12, a13, a21, a22, a23, a31, a32, a33};
area = Cuboid[{0, 0, 0, 0, 0, 0, 0, 0, 0},
    {max0, max0, max0, max0, max0, max0, max0, max0, max0}];
ArgMax[Ix, pixels ∈ area]
Ix-max = MaxValue[Ix, pixels ∈ area]
ArgMin[Ix, pixels ∈ area]
Ix-min = MinValue[Ix, pixels ∈ area]
Iy := Total[Total[Sy]];
ArgMax[Iy, pixels ∈ area]
Iy-max = MaxValue[Iy, pixels ∈ area]
ArgMin[Iy, pixels ∈ area]
Iy-min = MinValue[Iy, pixels ∈ area]
bitSize1 = Ceiling[Max[Log2[Abs[Ix-max]], Log2[Abs[Ix-min]]]] + 1
bitSize2 = Ceiling[Max[Log2[Abs[Iy-max]], Log2[Abs[Iy-min]]]] + 1

{0, 0, 255, 0, 0, 255, 0, 0, 255}

1020

{255, 0, 0, 255, 0, 0, 255, 0, 0}

-1020

{255, 255, 255, 0, 0, 0, 0, 0, 0}

1020

{0, 0, 0, 0, 0, 0, 255, 255, 255}
```

```
-1020

11

11

(*
a, b and c value of Matrix M
*)
var0 = {Ix0, Iy0};
area0 = Cuboid[{Ix-min, Iy-min}, {Ix-max, Iy-max}];
ArgMax[Ix0^2, var0 ∈ area0]
amax = MaxValue[Ix0^2, var0 ∈ area0]
ArgMin[Ix0^2, var0 ∈ area0]
amin = MinValue[Ix0^2, var0 ∈ area0]
ArgMax[Ix0 Iy0, var0 ∈ area0]
bmax = MaxValue[Ix0 Iy0, var0 ∈ area0]
ArgMin[Ix0 Iy0, var0 ∈ area0]
bmin = MinValue[Ix0 Iy0, var0 ∈ area0]
ArgMax[Iy0^2, var0 ∈ area0]
cmax = MaxValue[Iy0^2, var0 ∈ area0]
ArgMin[Iy0^2, var0 ∈ area0]
cmin = MinValue[Iy0^2, var0 ∈ area0]
bitSizea = Ceiling[Max[Log2[Abs[amax]], Log2[Abs[amin]]]] + 1
bitSizeb = Ceiling[Max[Log2[Abs[bmax]], Log2[Abs[bmin]]]] + 1
bitSizec = Ceiling[Max[Log2[Abs[cmax]], Log2[Abs[cmin]]]] + 1

{-1020, 0}

1040 400

{0, 0}

0

{-1020, -1020}

1040 400

{-1020, 1020}

-1040 400
```

```
{0, -1020}

1040 400

{0, 0}

0

21

21

21

(*
Block filter:
*)
```

$$max_1 = 9 \left(\text{Ceiling}\left[\frac{b_{max}}{2^5} \right] \right)$$ (*Right shift to reduce word size to 16 bits*)

$$bitSize_3 = \text{Ceiling}[\text{Log2}[max_1]] + 1$$

```
292617

20

(*
Harris response:
*)
```

$$max_2 = \text{Ceiling}\left[\frac{max_1}{2^4} \right]$$ (*Right shift to reduce word size to 16 bits*)

```
18289
```

```
k = 1/32 + 1/128;
r := a c - b^2 - k (a + c)^2;
var1 = {a, b, c};
area1 = Cuboid[{-max2, -max2, -max2}, {max2, max2, max2}];
ArgMax[r, var1 ∈ area1]
max3 = Ceiling[MaxValue[r, var1 ∈ area1]]
Ceiling[Log2[max3]] + 1
ArgMin[r, var1 ∈ area1]
min3 = Ceiling[MinValue[r, var1 ∈ area1]]
Ceiling[Log2[Abs[min3]]] + 1
```

```
{-18289, 0, -18289}

282223846

30

{-18289, -18289, 18289}

-668975042

31
```

APPENDIX B: SYNTHESIS LOG FOR THE FINAL VERSION OF THE HARRIS CO-PROCESSOR

```
Design Summary:
Number of errors:      0
Number of warnings:   31
Slice Logic Utilization:
  Number of Slice Registers:                25,973 out of 106,400   24%
    Number used as Flip Flops:              23,835
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:            2,138
  Number of Slice LUTs:                     25,538 out of  53,200   48%
    Number used as logic:                   23,734 out of  53,200   44%
      Number using O6 output only:          16,126
      Number using O5 output only:             163
      Number using O5 and O6:                7,445
      Number used as ROM:                          0
    Number used as Memory:                     851 out of  17,400    4%
      Number used as Dual Port RAM:            116
        Number using O6 output only:            92
        Number using O5 output only:             5
        Number using O5 and O6:                 19
      Number used as Single Port RAM:            0
      Number used as Shift Register:           735
        Number using O6 output only:           717
        Number using O5 output only:             0
        Number using O5 and O6:                 18
    Number used exclusively as route-thrus:    953
      Number with same-slice register load:    729
      Number with same-slice carry load:       224
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                 9,462 out of  13,300   71%
  Number of LUT Flip Flop pairs used:       30,696
    Number with an unused Flip Flop:         8,197 out of  30,696   26%
    Number with an unused LUT:               5,158 out of  30,696   16%
    Number of fully used LUT-FF pairs:      17,341 out of  30,696   56%
    Number of unique control sets:             974
    Number of slice register sites lost
      to control set restrictions:           2,709 out of 106,400    2%
```

```
  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                        94 out of     200   47%
    Number of LOCed IOBs:                       94 out of      94  100%
  Number of bonded IOPAD:                      130 out of     130  100%
    IOB Flip Flops:                             23

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                  8 out of     140    5%
    Number using RAMB36E1 only:                  8
    Number using FIFO36E1 only:                  0
  Number of RAMB18E1/FIFO18E1s:                  0 out of     280    0%
  Number of BUFG/BUFGCTRLs:                      9 out of      32   28%
    Number used as BUFGs:                        9
    Number used as BUFGCTRLs:                    0
  Number of IDELAYE2/IDELAYE2_FINEDELAYs:        0 out of     200    0%
  Number of ILOGICE2/ILOGICE3/ISERDESE2s:        0 out of     200    0%
  Number of ODELAYE2/ODELAYE2_FINEDELAYs:        0
  Number of OLOGICE2/OLOGICE3/OSERDESE2s:       83 out of     200   41%
    Number used as OLOGICE2s:                   83
    Number used as OLOGICE3s:                    0
    Number used as OSERDESE2s:                   0
  Number of PHASER_IN/PHASER_IN_PHYs:            0 out of      16    0%
  Number of PHASER_OUT/PHASER_OUT_PHYs:          0 out of      16    0%
  Number of BSCANs:                              0 out of       4    0%
  Number of BUFHCEs:                             0 out of      72    0%
  Number of BUFRs:                               0 out of      16    0%
  Number of CAPTUREs:                            0 out of       1    0%
  Number of DNA_PORTs:                           0 out of       1    0%
  Number of DSP48E1s:                          111 out of     220   50%
  Number of EFUSE_USRs:                          0 out of       1    0%
  Number of FRAME_ECCs:                          0 out of       1    0%
  Number of ICAPs:                               0 out of       2    0%
  Number of IDELAYCTRLs:                         0 out of       4    0%
  Number of IN_FIFOs:                            0 out of      16    0%
  Number of MMCME2_ADVs:                         2 out of       4   50%
  Number of OUT_FIFOs:                           0 out of      16    0%
  Number of PHASER_REFs:                         0 out of       4    0%
  Number of PHY_CONTROLs:                        0 out of       4    0%
```

```
  Number of PLLE2_ADVs:                          0 out of       4    0%
  Number of PS7s:                                1 out of       1  100%
  Number of STARTUPs:                            0 out of       1    0%
  Number of XADCs:                               0 out of       1    0%

Device Utilization Summary:

   Number of BUFGs                             9 out of 32     28%
   Number of DSP48E1s                        111 out of 220    50%
   Number of External IOB33s                  94 out of 200    47%
      Number of LOCed IOB33s                  94 out of 94    100%

   Number of External IOPADs                 130 out of 130   100%
      Number of LOCed IOPADs                 127 out of 130    97%

   Number of MMCME2_ADVs                       2 out of 4      50%
   Number of OLOGICE2s                        83 out of 200    41%
   Number of PS7s                              1 out of 1     100%
   Number of RAMB36E1s                         8 out of 140     5%
   Number of Slices                         9462 out of 13300  71%
   Number of Slice Registers               25973 out of 106400 24%
      Number used as Flip Flops            25973
      Number used as Latches                   0
      Number used as LatchThrus                0

   Number of Slice LUTS                    25538 out of 53200  48%
   Number of Slice LUT-Flip Flop pairs     30287 out of 53200  56%


Overall effort level (-ol):   High
Router effort level (-rl):    High
```

APPENDIX C: C++ CLASS FOR I/O BETWEEN THE MAIN PROCESSOR AND THE HARRIS CO-PROCESSOR THROUGH AXI4-LITE

a) hwharris.h

```
#ifndef HWHARRIS_H
#define HWHARRIS_H

#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/features2d/features2d.hpp>

#define GPIO_BASE_ADDRESS 0x70400000
#define MAP_SIZE sizeof(int)
#define MAP_MASK (MAP_SIZE - 1)

using namespace cv;

class hwHarris
{
public:
    hwHarris();
    ~hwHarris();
    void apply(Mat &input, Mat &next_input, vector<KeyPoint> &output);

private:
    int memfd;
    bool preload;
    void *mapped_base, *mapped_dev_base;
    off_t dev_base;
    unsigned volatile int* hw;
};

#endif // HWHARRIS_H
```

b) hwharris.cpp

```
#include "hwharris.h"

// Direct memory access based on
http://www.wiki.xilinx.com/file/view/usergpio.c/414351414/usergpio.c

hwHarris::hwHarris()
{
    dev_base = GPIO_BASE_ADDRESS;
    memfd = open("/dev/mem", O_RDWR | O_SYNC);
    if (memfd == -1) {
        printf("Can't open /dev/mem.\n");
        exit(0);
    }
    mapped_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, memfd,
dev_base & ~MAP_MASK);
    if (mapped_base == (void *) -1) {
        printf("Can't map the memory to user space.\n");
        exit(0);
    }
    mapped_dev_base = mapped_base + (dev_base & MAP_MASK);
    hw = (unsigned volatile int *) (mapped_dev_base);
    preload = 1;
}

hwHarris::~hwHarris()
{
    if (munmap(mapped_base, MAP_SIZE) == -1) {
        printf("Can't unmap memory from user space.\n");
        exit(0);
    }
    close(memfd);
}

void hwHarris::apply(Mat &input, Mat &next_input, vector<KeyPoint> &output)
{
    int rows = input.rows;
    int cols = input.cols;
    int i,j,k;
    int px[] = {-1, -1, -1};
    int py[] = {-1, -1, -1};
    unsigned int pixel_value0, pixel_value1, pixel_value2, pixel_value3;
    unsigned int corner_value;
```

```
    const unsigned int di = rows/2 - 3;
    const unsigned int dj = cols/2 - 3;
    for(i=3;i<rows/2;i++){
        for(j=0;j<(cols/2+3);j++){

            px[2] = px[1];
            py[2] = py[1];
            px[1] = px[0];
            py[1] = py[0];
            px[0] = j-3;
            py[0] = i;

            if(i>3 || j>1 || preload){
                if(i>3) preload = 0;
                for(k=-3;k<4;k++){
                    pixel_value0 = input.at<uchar>(i+k   , j   );
                    pixel_value1 = input.at<uchar>(i+k+di, j   );
                    pixel_value2 = input.at<uchar>(i+k,    j+dj);
                    pixel_value3 = input.at<uchar>(i+k+di, j+dj);

                    *hw = (pixel_value3<<24) + (pixel_value2<<16) +
(pixel_value1<<8) + pixel_value0;
                }
                if(px[2]>=3 && py[2]>=3){
                    corner_value = *hw;
                    if(corner_value&0x1){
                        KeyPoint kp(px[2]+dj,py[2]+di,3);
                        output.push_back(kp);
                    }
                    if(corner_value&0x2){
                        KeyPoint kp(px[2]+dj,py[2],3);
                        output.push_back(kp);
                    }
                    if(corner_value&0x4){
                        KeyPoint kp(px[2],py[2]+di,3);
                        output.push_back(kp);
                    }
                    if(corner_value&0x8){
                        KeyPoint kp(px[2],py[2],3);
                        output.push_back(kp);
                    }
                }
            }
        }
    }
    for(j=0;j<1;j++){
```

```
        for(k=0;k<7;k++){
            pixel_value0 = next_input.at<uchar>(k,0);
            pixel_value1 = next_input.at<uchar>(k+di,0);
            pixel_value2 = next_input.at<uchar>(k,dj);
            pixel_value3 = next_input.at<uchar>(k+di,dj);
            *hw = (pixel_value3<<24) + (pixel_value2<<16) + (pixel_value1<<8) +
pixel_value0;
        }
    }
    corner_value = *hw;
    if(corner_value&0x1){
        KeyPoint kp(px[1]+dj,py[1]+di,3);
        output.push_back(kp);
    }
    if(corner_value&0x2){
        KeyPoint kp(px[1]+dj,py[1],3);
        output.push_back(kp);
    }
    if(corner_value&0x4){
        KeyPoint kp(px[1],py[1]+di,3);
        output.push_back(kp);
    }
    if(corner_value&0x8){
        KeyPoint kp(px[1],py[1],3);
        output.push_back(kp);
    }
    for(j=0;j<1;j++){
        for(k=0;k<7;k++){
            pixel_value0 = next_input.at<uchar>(k,1);
            pixel_value1 = next_input.at<uchar>(k+di,1);
            pixel_value2 = next_input.at<uchar>(k,1+dj);
            pixel_value3 = next_input.at<uchar>(k+di,1+dj);
            *hw = (pixel_value3<<24) + (pixel_value2<<16) + (pixel_value1<<8) +
pixel_value0;
        }
    }
    corner_value = *hw;
    if(corner_value&0x1){
        KeyPoint kp(px[0]+dj,py[0]+di,3);
        output.push_back(kp);
    }
    if(corner_value&0x2){
        KeyPoint kp(px[0]+dj,py[0],3);
        output.push_back(kp);
    }
    if(corner_value&0x4){
```

```
        KeyPoint kp(px[0],py[0]+di,3);
        output.push_back(kp);
    }
    if(corner_value&0x8){
        KeyPoint kp(px[0],py[0],3);
        output.push_back(kp);
    }
}
```

APPENDIX D: VHDL CODE FOR THE HARRIS CO-PROCESSOR

a) user_logic.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;

entity user_logic is
	generic(
		C_NUM_REG    : integer := 1;
		C_SLV_DWIDTH : integer := 32
	);
	port(
		Bus2IP_Clk    : in  std_logic;
		Bus2IP_Resetn : in  std_logic;
		Bus2IP_Data   : in  std_logic_vector(31 downto 0);
		Bus2IP_BE     : in  std_logic_vector(3 downto 0);
		Bus2IP_RdCE   : in  std_logic_vector(0 downto 0);
		Bus2IP_WrCE   : in  std_logic_vector(0 downto 0);
		IP2Bus_Data   : out std_logic_vector(31 downto 0);
		IP2Bus_RdAck  : out std_logic;
		IP2Bus_WrAck  : out std_logic;
		IP2Bus_Error  : out std_logic
	);

	attribute MAX_FANOUT : string;
	attribute SIGIS : string;

	attribute SIGIS of Bus2IP_Clk : signal is "CLK";
	attribute SIGIS of Bus2IP_Resetn : signal is "RST";

end entity user_logic;

architecture IMP of user_logic is
	component system
		port(clk    : in  std_logic;
			rst    : in  std_logic;
			write  : in  std_logic;
```

```
                data   : in  std_logic_vector(31 downto 0);
                corner : out std_logic_vector(31 downto 0));
        end component system;

        signal read_data         : std_logic_vector(31 downto 0);
        signal slv_reg_write_sel : std_logic_vector(0 to 0);
        signal slv_reg_read_sel  : std_logic_vector(0 to 0);
        signal slv_ip2bus_data   : std_logic_vector(31 downto 0);
        signal slv_read_ack      : std_logic;
        signal slv_write_ack     : std_logic;

begin
        sys : system
                port map(clk    => Bus2IP_Clk,
                         rst    => Bus2IP_Resetn,
                         write  => Bus2IP_WrCE(0),
                         data   => Bus2IP_Data,
                         corner => read_data);

        slv_reg_write_sel <= Bus2IP_WrCE(0 downto 0);
        slv_reg_read_sel  <= Bus2IP_RdCE(0 downto 0);
        slv_write_ack     <= Bus2IP_WrCE(0);
        slv_read_ack      <= Bus2IP_RdCE(0);

        SLAVE_REG_READ_PROC : process(slv_reg_read_sel, read_data) is
        begin
                case slv_reg_read_sel is
                        when "1"    => slv_ip2bus_data <= read_data;
                        when others => slv_ip2bus_data <= (others => '0');
                end case;

        end process SLAVE_REG_READ_PROC;

        IP2Bus_Data <= slv_ip2bus_data when slv_read_ack = '1' else (others => '0');

        IP2Bus_WrAck <= slv_write_ack;
        IP2Bus_RdAck <= slv_read_ack;
        IP2Bus_Error <= '0';

end IMP;
```

b) main.vhd

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package main is
	constant width  : integer := 640;
	constant height : integer := 360;

	type matrix3u8 is array (integer range 1 to 3, integer range 1 to 3) of
unsigned(7 downto 0);
	type matrix5s11 is array (integer range 1 to 5, integer range 1 to 5) of
signed(10 downto 0);
	type matrix7u8 is array (integer range 1 to 7, integer range 1 to 7) of
unsigned(7 downto 0);
	type matrix3s16 is array (integer range 1 to 3, integer range 1 to 3) of
signed(15 downto 0);
	type matrix5s16 is array (integer range 1 to 5, integer range 1 to 5) of
signed(15 downto 0);
	type matrix3s32 is array (integer range 1 to 3, integer range 1 to 3) of
signed(31 downto 0);
end main;
```

c) system.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

entity system is
	port(
		clk  : in  std_logic;
		rst  : in  std_logic;
		write  : in  std_logic;
		data   : in  std_logic_vector(31 downto 0);
		corner : out std_logic_vector(31 downto 0)
	);
end entity system;

architecture RTL of system is
	signal loadt      : std_logic;
	signal readyt     : std_logic;
	signal t          : std_logic_vector(5 downto 0);
	signal icm        : matrix7u8;
	signal sxm        : matrix5s11;
```

```
signal sym        : matrix5s11;
signal mam        : matrix5s16;
signal mbm        : matrix5s16;
signal mcm        : matrix5s16;
signal bfam       : matrix3s16;
signal bfbm       : matrix3s16;
signal bfcm       : matrix3s16;
signal hrm        : matrix3s32;
signal trm        : matrix3s32;
signal max        : signed(31 downto 0);
signal thr        : signed(31 downto 0);
signal cornerbit  : std_logic;
signal corner_sr  : std_logic_vector(3 downto 0);

component loadControl
       port(clk   : in  std_logic;
              write : in  std_logic;
              rst   : in  std_logic;
              load  : out std_logic);
end component loadControl;

component inputConverterMux
       port(clk   : in  std_logic;
              rst   : in  std_logic;
              write : in  std_logic;
              load  : in  std_logic;
              data  : in  std_logic_vector(31 downto 0);
              a     : out matrix7u8;
              ready : out std_logic);
end component inputConverterMux;

component control
       port(clk   : in  std_logic;
              write : in  std_logic;
              rst   : in  std_logic;
              ready : in  std_logic;
              t     : out std_logic_vector(5 downto 0));
end component control;

component sobelX
       port(clk   : in  std_logic;
              rst   : in  std_logic;
              en    : in  std_logic;
              write : in  std_logic;
              a     : in  matrix3u8;
              ix    : out signed(10 downto 0));
```

```
end component sobelX;

component sobelY
	port(clk   : in  std_logic;
		rst   : in  std_logic;
		en    : in  std_logic;
		write : in  std_logic;
		a     : in  matrix3u8;
		iy    : out signed(10 downto 0));
end component sobelY;

component matrixM
	port(clk     : in  std_logic;
		rst     : in  std_logic;
		en      : in  std_logic;
		write   : in  std_logic;
		ix, iy  : in  signed(10 downto 0);
		a, b, c : out signed(15 downto 0));
end component matrixM;

component blockFilter
	port(clk   : in  std_logic;
		rst   : in  std_logic;
		en    : in  std_logic;
		write : in  std_logic;
		a     : in  matrix3s16;
		sum   : out signed(15 downto 0));
end component blockFilter;

component harrisResponse
	port(clk     : in  std_logic;
		rst     : in  std_logic;
		en0     : in  std_logic;
		en1     : in  std_logic;
		write   : in  std_logic;
		a, b, c : in  signed(15 downto 0);
		r       : out signed(31 downto 0));
end component harrisResponse;

component findMax
	port(clk      : in  std_logic;
		rst      : in  std_logic;
		en       : in  std_logic;
		write    : in  std_logic;
		response : in  signed(31 downto 0);
		max      : out signed(31 downto 0));
```

```
	end component findMax;

	component threshold
		port(clk   : in  std_logic;
			rst   : in  std_logic;
			en    : in  std_logic;
			write : in  std_logic;
			r0    : in  signed(31 downto 0);
			thr   : in  signed(31 downto 0);
			r1    : out signed(31 downto 0));
	end component threshold;

	component localMaximumBin
		port(r      : in  matrix3s32;
			corner : out std_logic);
	end component localMaximumBin;

	component cornerShiftRegister
		port(clk       : in  std_logic;
			rst       : in  std_logic;
			write     : in  std_logic;
			corner    : in  std_logic;
			corner_sr : out std_logic_vector(3 downto 0));
	end component cornerShiftRegister;

begin
	lc : loadControl
		port map(clk   => clk,
				write => write,
				rst   => rst,
				load  => loadt);

	ic : inputConverterMux
		port map(clk   => clk,
				rst   => rst,
				write => write,
				load  => loadt,
				data  => data,
				a     => icm,
				ready => readyt);

	crtl : control
		port map(clk   => clk,
				write => write,
				rst   => rst,
				ready => readyt,
```

```
                t     => t);

lines0 : for i in 1 to 5 generate
begin
      columns0 : for j in 1 to 5 generate
      begin
            sx : sobelX
                  port map(clk     => clk,
                              rst     => rst,
                              en      => readyt,
                              write   => write,
                              a(1, 1) => icm(i, j),
                              a(1, 2) => icm(i, j + 1),
                              a(1, 3) => icm(i, j + 2),
                              a(2, 1) => icm(i + 1, j),
                              a(2, 2) => icm(i + 1, j + 1),
                              a(2, 3) => icm(i + 1, j + 2),
                              a(3, 1) => icm(i + 2, j),
                              a(3, 2) => icm(i + 2, j + 1),
                              a(3, 3) => icm(i + 2, j + 2),
                              ix      => sxm(i, j));
            sy : sobelY
                  port map(clk     => clk,
                              rst     => rst,
                              en      => readyt,
                              write   => write,
                              a(1, 1) => icm(i, j),
                              a(1, 2) => icm(i, j + 1),
                              a(1, 3) => icm(i, j + 2),
                              a(2, 1) => icm(i + 1, j),
                              a(2, 2) => icm(i + 1, j + 1),
                              a(2, 3) => icm(i + 1, j + 2),
                              a(3, 1) => icm(i + 2, j),
                              a(3, 2) => icm(i + 2, j + 1),
                              a(3, 3) => icm(i + 2, j + 2),
                              iy      => sym(i, j));
            mm : matrixM
                  port map(clk   => clk,
                              rst   => rst,
                              en    => t(0),
                              write => write,
                              ix    => sxm(i, j),
                              iy    => sym(i, j),
                              a     => mam(i, j),
                              b     => mbm(i, j),
                              c     => mcm(i, j));
```

```
	end generate columns0;
end generate lines0;

lines1 : for i in 1 to 3 generate
begin
	columns1 : for j in 1 to 3 generate
	begin
		bfa : blockFilter
			port map(clk	=> clk,
					rst	=> rst,
					en	=> t(1),
					write	=> write,
					a(1, 1) => mam(i, j),
					a(1, 2) => mam(i, j + 1),
					a(1, 3) => mam(i, j + 2),
					a(2, 1) => mam(i + 1, j),
					a(2, 2) => mam(i + 1, j + 1),
					a(2, 3) => mam(i + 1, j + 2),
					a(3, 1) => mam(i + 2, j),
					a(3, 2) => mam(i + 2, j + 1),
					a(3, 3) => mam(i + 2, j + 2),
					sum	=> bfam(i, j));
		bfb : blockFilter
			port map(clk	=> clk,
					rst	=> rst,
					en	=> t(1),
					write	=> write,
					a(1, 1) => mbm(i, j),
					a(1, 2) => mbm(i, j + 1),
					a(1, 3) => mbm(i, j + 2),
					a(2, 1) => mbm(i + 1, j),
					a(2, 2) => mbm(i + 1, j + 1),
					a(2, 3) => mbm(i + 1, j + 2),
					a(3, 1) => mbm(i + 2, j),
					a(3, 2) => mbm(i + 2, j + 1),
					a(3, 3) => mbm(i + 2, j + 2),
					sum	=> bfbm(i, j));
		bfc : blockFilter
			port map(clk	=> clk,
					rst	=> rst,
					en	=> t(1),
					write	=> write,
					a(1, 1) => mcm(i, j),
					a(1, 2) => mcm(i, j + 1),
					a(1, 3) => mcm(i, j + 2),
					a(2, 1) => mcm(i + 1, j),
```

```
                                a(2, 2) => mcm(i + 1, j + 1),
                                a(2, 3) => mcm(i + 1, j + 2),
                                a(3, 1) => mcm(i + 2, j),
                                a(3, 2) => mcm(i + 2, j + 1),
                                a(3, 3) => mcm(i + 2, j + 2),
                                sum     => bfcm(i, j));

                hr : harrisResponse
                       port map(clk   => clk,
                                rst   => rst,
                                en0   => t(2),
                                en1   => t(3),
                                write => write,
                                a     => bfam(i, j),
                                b     => bfbm(i, j),
                                c     => bfcm(i, j),
                                r     => hrm(i, j));
                tr : threshold
                       port map(clk   => clk,
                                rst   => rst,
                                en    => t(5),
                                write => write,
                                r0    => hrm(i, j),
                                thr   => thr,
                                r1    => trm(i, j));
       end generate columns1;
end generate lines1;

fm : findMax
       port map(clk      => clk,
                rst      => rst,
                en       => t(4),
                write    => write,
                response => hrm(2, 2),
                max      => max);

thr <= resize(max(31 downto 7), 32) + resize(max(31 downto 9), 32);

lm : localMaximumBin
       port map(r      => trm,
                corner => cornerbit);

sr : component cornerShiftRegister
       port map(clk       => clk,
                rst       => rst,
                write     => write,
```

```
                        corner    => cornerbit,
                        corner_sr => corner_sr);

        corner <= "00000000000000000000000000000" & corner_sr;

end architecture RTL;
```

d) loadControl.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.main.all;

entity loadControl is
        port(
                clk   : in  std_logic;
                write : in  std_logic;
                rst   : in  std_logic;
                load  : out std_logic
        );
end entity loadControl;

architecture RTL of loadControl is
        signal i     : integer range 1 to (height / 2 - 3)    := 1;
        signal j     : integer range 1 to (width / 2 + 3) * 7 := 1;
        signal loadt : std_logic                              := '1';
begin
        store : process(clk, write, rst) is
        begin
                if rst = '0' then
                        i     <= 1;
                        j     <= 1;
                        loadt <= '1';
                elsif (clk'event and clk = '1' and write = '1') then
                        if (j < (width / 2 + 3) * 7) then
                                if (j < 49) then
                                        loadt <= '1';
                                else
                                        loadt <= '0';
                                end if;
                                j <= j + 1;
                        else
                                j <= 1;
                                if (i < (height / 2 - 3)) then
```

```
                                    loadt <= '1';
                                    i     <= i + 1;
                              else
                                    i <= 1;
                              end if;
                        end if;
                  end if;
            end process store;
            load <= loadt;
end architecture RTL;
```

e) inputConverterMux.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

entity inputConverterMux is
      port(
            clk   : in  std_logic;
            rst   : in  std_logic;
            write : in  std_logic;
            load  : in  std_logic;
            data  : in  std_logic_vector(31 downto 0);
            a     : out matrix7u8;
            ready : out std_logic
      );
end entity inputConverterMux;

architecture RTL of inputConverterMux is
      signal i      : integer range 1 to 7          := 1;
      signal i_1    : integer range 1 to 7          := 1;
      signal i_2    : integer range 1 to 7          := 1;
      signal i_3    : integer range 1 to 7          := 1;
      signal data_1 : std_logic_vector(31 downto 0) :=
"00000000000000000000000000000000";
      signal data_2 : std_logic_vector(31 downto 0) :=
"00000000000000000000000000000000";
      signal data_3 : std_logic_vector(31 downto 0) :=
"00000000000000000000000000000000";
      constant z    : unsigned(7 downto 0)          := "00000000";
```

```
	constant at   : matrix7u8                    := ((z, z, z, z, z, z, z), (z,
z, z, z, z, z, z), (z, z, z, z, z, z, z), (z, z, z, z, z, z, z), (z, z, z, z, z, z,
z), (z, z, z, z, z, z, z), (z, z, z, z, z, z, z));
	signal a_0    : matrix7u8;
	signal a_1    : matrix7u8;
	signal a_2    : matrix7u8;
	signal a_3    : matrix7u8;

	component inputConverter
		port(clk   : in  std_logic;
			rst   : in  std_logic;
			write : in  std_logic;
			i     : in  integer range 1 to 7;
			data  : in  std_logic_vector(7 downto 0);
			a     : out matrix7u8);
	end component inputConverter;
begin
	ic0 : component inputConverter
		port map(clk   => clk,
				rst   => rst,
				write => write,
				i     => i,
				data  => data(7 downto 0),
				a     => a_0);
	ic1 : component inputConverter
		port map(clk   => clk,
				rst   => rst,
				write => write,
				i     => i_1,
				data  => data_1(15 downto 8),
				a     => a_1);
	ic2 : component inputConverter
		port map(clk   => clk,
				rst   => rst,
				write => write,
				i     => i_2,
				data  => data_2(23 downto 16),
				a     => a_2);
	ic3 : component inputConverter
		port map(clk   => clk,
				rst   => rst,
				write => write,
				i     => i_3,
				data  => data_3(31 downto 24),
				a     => a_3);
```

```
	ctrl : process(clk, rst, write) is
	begin
		if rst = '0' then
			i <= 1;
		elsif (clk'event and clk = '1' and write = '1') then
			if i = 1 then
				a <= a_0;
			elsif i = 2 then
				a <= a_1;
			elsif i = 3 then
				a <= a_2;
			elsif i = 4 then
				a <= a_3;
			else
				a <= at;
			end if;
			if (i < 7) then
				i <= i + 1;
			else
				i <= 1;
			end if;
			i_1    <= i;
			i_2    <= i_1;
			i_3    <= i_2;
			data_1 <= data;
			data_2 <= data_1;
			data_3 <= data_2;
		end if;
	end process ctrl;
	ready <= '1' when (i > 1 and i <= 5 and load = '0') else '0';
	--a     <= a_0 when i = 1 else a_1 when i = 2 else a_2 when i = 3 else a_3
when i = 4 else at;
end architecture RTL;
```

f) inputConverter.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

entity inputConverter is
	port(
		clk   : in  std_logic;
```

```
		rst   : in  std_logic;
		write : in  std_logic;
		i     : in  integer range 1 to 7;
		data  : in  std_logic_vector(7 downto 0);
		a     : out matrix7u8
	);
end entity inputConverter;

architecture RTL of inputConverter is
	signal at  : matrix7u8;
	constant z : unsigned(7 downto 0) := "00000000";

begin
	name : process(clk, rst, write) is
	begin
		if rst = '0' then
			at <= ((z, z, z, z, z, z, z),
				(z, z, z, z, z, z, z),
				(z, z, z, z, z, z, z),
				(z, z, z, z, z, z, z),
				(z, z, z, z, z, z, z),
				(z, z, z, z, z, z, z),
				(z, z, z, z, z, z, z));
		elsif (clk'event and clk = '1' and write = '1') then
			at(i, 1) <= at(i, 2);
			at(i, 2) <= at(i, 3);
			at(i, 3) <= at(i, 4);
			at(i, 4) <= at(i, 5);
			at(i, 5) <= at(i, 6);
			at(i, 6) <= at(i, 7);
			at(i, 7) <= unsigned(data);
		end if;
	end process name;
	a <= at;
end architecture RTL;
```

g) control.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity control is
	port(
		clk   : in  std_logic;
```

```
		write : in  std_logic;
		rst   : in  std_logic;
		ready : in  std_logic;
		t     : out std_logic_vector(5 downto 0)
	);
end entity control;

architecture RTL of control is
	signal tt    : std_logic_vector(5 downto 0);
begin
	store : process(clk, write, rst) is
	begin
		if rst = '0' then
			tt <= "000000";
		elsif (clk'event and clk = '1' and write = '1') then
			tt(5) <= tt(4);
			tt(4) <= tt(3);
			tt(3) <= tt(2);
			tt(2) <= tt(1);
			tt(1) <= tt(0);
			tt(0) <= ready;
		end if;
	end process store;
	t <= tt;
end architecture RTL;
```

h) sobelX.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

-- |-1  0 +1|
-- |-2  0 +2|
-- |-1  0 +1|

entity sobelX is
	port(
		clk   : in  std_logic;
		rst   : in  std_logic;
		en    : in  std_logic;
		write : in  std_logic;
		a     : in  matrix3u8;
```

```
              ix    : out signed(10 downto 0)
       );
end entity sobelX;

architecture RTL of sobelX is
       signal p11, p13, p31, p33 : signed(8 downto 0);
       signal p21, p23           : signed(9 downto 0);
       signal ixt                : signed(10 downto 0);

begin
       p11 <= -signed(resize(a(1, 1), 9));
       p21 <= -signed(resize(a(2, 1), 10)) - signed(resize(a(2, 1), 10));
       p31 <= -signed(resize(a(3, 1), 9));
       p13 <= signed(resize(a(1, 3), 9));
       p23 <= signed(resize(a(2, 3), 10)) + signed(resize(a(2, 3), 10));
       p33 <= signed(resize(a(3, 3), 9));
       ixt <= resize(p11, 11) + resize(p13, 11) + resize(p21, 11) + resize(p23, 11)
+ resize(p31, 11) + resize(p33, 11);

       reg : process(clk, rst, en, write) is
       begin
              if rst = '0' then
                     ix <= to_signed(0, 11);
              elsif (clk'event and clk = '1' and write = '1' and en = '1') then
                     ix <= ixt;
              end if;
       end process reg;

end architecture RTL;
```

i) sobelY.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

-- |+1 +2 +1|
-- | 0  0  0|
-- |-1 -2 -1|

entity sobelY is
       port(
              clk   : in  std_logic;
```

```
		rst   : in  std_logic;
		en    : in  std_logic;
		write : in  std_logic;
		a     : in  matrix3u8;
		iy    : out signed(10 downto 0)
	);
end entity sobelY;

architecture RTL of sobelY is
	signal p11, p13, p31, p33 : signed(8 downto 0);
	signal p12, p32           : signed(9 downto 0);
	signal iyt                : signed(10 downto 0);

begin
	p11 <= signed(resize(a(1, 1), 9));
	p12 <= signed(resize(a(1, 2), 10)) + signed(resize(a(1, 2), 10));
	p13 <= signed(resize(a(1, 3), 9));
	p31 <= -signed(resize(a(3, 1), 9));
	p32 <= -signed(resize(a(3, 2), 10)) - signed(resize(a(3, 2), 10));
	p33 <= -signed(resize(a(3, 3), 9));
	iyt <= resize(p11, 11) + resize(p12, 11) + resize(p13, 11) + resize(p31, 11)
+ resize(p32, 11) + resize(p33, 11);

	reg : process(clk, rst, en, write) is
	begin
		if rst = '0' then
			iy <= to_signed(0, 11);
		elsif (clk'event and clk = '1' and write = '1' and en = '1') then
			iy <= iyt;
		end if;
	end process reg;

end architecture RTL;
```

j) matrixM.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity matrixM is
	port(
		clk   : in  std_logic;
		rst   : in  std_logic;
```

```
              en      : in  std_logic;
              write   : in  std_logic;
              ix, iy  : in  signed(10 downto 0);
              a, b, c : out signed(15 downto 0)
       );
end entity matrixM;

architecture RTL of matrixM is
       signal at, bt, ct : signed(20 downto 0);
begin
       at <= resize(ix * ix, 21);
       bt <= resize(ix * iy, 21);
       ct <= resize(iy * iy, 21);
       -- division by 2^5 = 32

       reg : process(clk, rst, en, write) is
       begin
              if rst = '0' then
                     a <= to_signed(0, 16);
                     b <= to_signed(0, 16);
                     c <= to_signed(0, 16);
              elsif (clk'event and clk = '1' and write = '1' and en = '1') then
                     a <= at(20 downto 5);
                     b <= bt(20 downto 5);
                     c <= ct(20 downto 5);
              end if;
       end process reg;
end architecture RTL;
```

k) blockFilter.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

entity blockFilter is
       port(
              clk   : in  std_logic;
              rst   : in  std_logic;
              en    : in  std_logic;
              write : in  std_logic;
              a     : in  matrix3s16;
              sum   : out signed(15 downto 0)
```

```
	);
end entity blockFilter;

architecture RTL of blockFilter is
	signal sumt : signed(19 downto 0);
begin
	sumt <= resize(a(1, 1), 20) + resize(a(1, 2), 20) + resize(a(1, 3), 20) +
resize(a(2, 1), 20) + resize(a(2, 2), 20) + resize(a(2, 3), 20) + resize(a(3, 1),
20) + resize(a(3, 2), 20) + resize(a(3, 3), 20);
	-- division by 2^4 = 16

	reg : process(clk, rst, en, write) is
	begin
		if rst = '0' then
			sum <= to_signed(0, 16);
		elsif (clk'event and clk = '1' and write = '1' and en = '1') then
			sum <= sumt(19 downto 4);
		end if;
	end process reg;

end architecture RTL;
```

l) harrisResponse.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- k = 1/32 + 1/128
-- r = det - k*tr^2
-- r = a*c - b*b - k*(a+c)^2

entity harrisResponse is
	port(
		clk     : in  std_logic;
		rst     : in  std_logic;
		en0     : in  std_logic;
		en1     : in  std_logic;
		write   : in  std_logic;
		a, b, c : in  signed(15 downto 0);
		r       : out signed(31 downto 0)
	);
end entity harrisResponse;
```

```
architecture RTL of harrisResponse is
	signal act, bbt : signed(31 downto 0);
	signal ac, bb   : signed(31 downto 0);
	signal a_plus_c : signed(16 downto 0);
	signal tr2      : signed(33 downto 0);
	signal k_tr2t   : signed(29 downto 0);
	signal k_tr2    : signed(29 downto 0);
	signal rt       : signed(31 downto 0);
begin
	act      <= a * c;
	bbt      <= b * b;
	a_plus_c <= resize(a, 17) + resize(c, 17);
	tr2      <= a_plus_c * a_plus_c;
	k_tr2t   <= resize(tr2(33 downto 5), 30) + resize(tr2(33 downto 7), 30);
	rt       <= resize(ac, 32) - resize(bb, 32) - resize(k_tr2, 32);

	reg0 : process(clk, en0, write) is
	begin
		if (clk'event and clk = '1' and write = '1' and en0 = '1') then
			ac    <= act;
			bb    <= bbt;
			k_tr2 <= k_tr2t;
		end if;
	end process reg0;

	reg1 : process(clk, rst, en1, write) is
	begin
		if rst = '0' then
			r <= to_signed(0, 32);
		elsif (clk'event and clk = '1' and write = '1' and en1 = '1') then
			r     <= rt;
		end if;
	end process reg1;

end architecture RTL;
```

m) findMax.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

entity findMax is
```

```
	port(
		clk	: in  std_logic;
		rst	: in  std_logic;
		en	: in  std_logic;
		write	: in  std_logic;
		response : in  signed(31 downto 0);
		max	: out signed(31 downto 0)
	);
end entity findMax;

architecture RTL of findMax is
	constant npixel : integer := ((height/2-3)*(width/2+3))*4;
	signal i		: integer range 1 to npixel := 1;
	signal internal_max : signed(31 downto 0)   :=
"10000000000000000000000000000000";
begin
	fm : process(clk, rst, en, write) is
	begin
		if rst = '0' then
			max <= "10000000000000000000000000000000";
		elsif (clk'event and clk = '1' and write = '1' and en = '1') then
			if response > internal_max then
				internal_max <= response;
			end if;
			if (i < npixel) then
				i <= i + 1;
			else
				i		<= 1;
				internal_max <= "10000000000000000000000000000000";
				max		<= internal_max;
			end if;
		end if;
	end process fm;
end architecture RTL;
```

n) threshold.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity threshold is
	port(
		clk	: in  std_logic;
```

```
		rst   : in  std_logic;
		en    : in  std_logic;
		write : in  std_logic;
		r0    : in  signed(31 downto 0);
		thr   : in  signed(31 downto 0);
		r1    : out signed(31 downto 0)
	);
end entity threshold;

architecture RTL of threshold is
	signal r0_delay : signed(31 downto 0);
	signal r1t      : signed(31 downto 0);
begin
	r1t <= r0_delay when r0_delay >= thr else to_signed(0, 32);

	reg : process(clk, rst, write) is
	begin
		if rst = '0' then
			r0_delay <= to_signed(0, 32);
			r1       <= to_signed(0, 32);
		elsif (clk'event and clk = '1' and write = '1') then
			r0_delay <= r0;
			if en = '1' then
				r1 <= r1t;
			else
				r1 <= to_signed(0, 32);
			end if;
		end if;
	end process reg;
end architecture RTL;
```

o) localMaximumBin.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.main.all;

entity localMaximumBin is
	port(
		r      : in  matrix3s32;
		corner : out std_logic
	);
end entity localMaximumBin;
```

```
architecture RTL of localMaximumBin is
begin

	corner <= '1' when (r(2,2) > r(1,1)) and
			   (r(2,2) > r(1,2)) and
			   (r(2,2) > r(1,3)) and
			   (r(2,2) > r(2,1)) and
			   (r(2,2) > r(2,3)) and
			   (r(2,2) > r(3,1)) and
			   (r(2,2) > r(3,2)) and
			   (r(2,2) > r(3,3)) else '0';
end architecture RTL;
```

p) cornerShiftRegister.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cornerShiftRegister is
	port(
		clk	: in  std_logic;
		rst	: in  std_logic;
		write	: in  std_logic;
		corner	: in  std_logic;
		corner_sr : out std_logic_vector(3 downto 0)
	);
end entity cornerShiftRegister;

architecture RTL of cornerShiftRegister is
	signal c : std_logic_vector(5 downto 0);
begin
	sr : process(clk, rst, write) is
	begin
		if rst = '0' then
			c <= "000000";
		elsif (clk'event and clk = '1' and write = '1') then
			c(5) <= c(4);
			c(4) <= c(3);
			c(3) <= c(2);
			c(2) <= c(1);
			c(1) <= c(0);
			c(0) <= corner;
```

```
            end if;
        end process sr;
        corner_sr <= c(5 downto 2);
end architecture RTL;
```

APPENDIX E: MODIFIED GRID DESCRIPTOR

a) grid.h

```
#ifndef GRID_H
#define GRID_H

#include <opencv2/core/core.hpp>
#include <opencv2/features2d/features2d.hpp>

using namespace cv;

namespace grid {
    void posDesc(vector<KeyPoint> &kp, Mat &dp, int g, int h);
}

#endif // GRID_H
```

b) grid.cpp

```
#include "grid.h"

void grid::posDesc(vector<KeyPoint> &kp, Mat &dp, int g, int h){
    int center = g/2+1;
    int needed_bytes = ((g*(g+1))>>3) + 1;
    dp = Mat::zeros( kp.size(), needed_bytes, CV_8U );
    vector<KeyPoint> ikp;
    for(int i=0;i<(int)kp.size();i++){
        ikp.push_back(kp.at(i));
        ikp.at(i).pt.x = (kp.at(i).pt.x)/(1<<h);
        ikp.at(i).pt.y = (kp.at(i).pt.y)/(1<<h);
    }
    for(int i=0;i<(int)ikp.size();i++){
        KeyPoint selpoint = ikp.at(i);
        int X, Y;
        int tot;
        uchar dsc[needed_bytes];
        for(int k=0;k<(int)needed_bytes;k++){
            dsc[k] = 0;
```

```
        }
        float lx = center - selpoint.pt.x;
        float ly = center - selpoint.pt.y;
        for(int j=0;j<(int)ikp.size();j++){
            X = ikp.at(j).pt.x + lx;
            Y = ikp.at(j).pt.y + ly;
            if(X>=0 && Y>=0 && X<=g && Y<=g){
                tot = g*Y+X;
                dsc[tot>>3] = dsc[tot>>3]|(1<<(tot&0x7));
            }
        }
        for(int j=0;j<(int)needed_bytes;j++){
            dp.at<uchar>(i,j) = dsc[j];
        }
    }
}
```

APPENDIX F: CUSTOM METHODS FOR POI CORRESPONDENCE TESTS

a) Ratio test

```
void Matcher::ratioTest(vector<vector<DMatch> > &matches, double maxRatio){
    uint i;
    double ratio;
    for(i=0; i<matches.size(); i++){
        ratio = matches.at(i).at(0).distance / matches.at(i).at(1).distance;
        if(ratio>maxRatio){
            matches.at(i).clear();
        }
    }
}
```

b) Symmetry test

```
void Matcher::symmetryTest(vector<vector<DMatch> > &matches1, vector<vector<DMatch>
> &matches2, vector<DMatch>& symMatches){
    uint i;
    DMatch local;
    for(i=0;i<matches1.size();i++){
        if(matches1.at(i).size()){
            local = matches1.at(i).at(0);
            if(matches2.at(local.trainIdx).size()){
                if(local.queryIdx == matches2.at(local.trainIdx).at(0).trainIdx){
                    symMatches.push_back(DMatch(matches1.at(i).at(0).queryIdx,
                                                matches1.at(i).at(0).trainIdx,
                                                matches1.at(i).at(0).distance));
                }
            }
        }
    }
}
```

c) Epipolar test

```
void Matcher::rectifiedTest(const vector<DMatch>& matches,
                            vector<DMatch>& rectMatches,
                            vector<KeyPoint>& kpu1,
                            vector<KeyPoint>& kpu2) {
    for (vector<DMatch>::const_iterator matchIterator1= matches.begin();
         matchIterator1!= matches.end(); ++matchIterator1) {
        if (fabsf(kpu2[(*matchIterator1).trainIdx].pt.y -
kpu1[(*matchIterator1).queryIdx].pt.y) < 5) {
            rectMatches.push_back(DMatch((*matchIterator1).queryIdx,
                                         (*matchIterator1).trainIdx,
                                         (*matchIterator1).distance));
        }
    }
}
```

APPENDIX G: CLASS FOR CONTROLLING FOCUS AND EXPOSURE THROUGH LIBWEBCAM

a) focus.h

```
#ifndef FOCUS_H
#define FOCUS_H

#include <webcam.h>
#include <iostream>
using namespace std;

class Focus
{
public:
    Focus(const char*);
    ~Focus();
    int setAutoFocus(int);
    int setFocus(int);
    int setAutoExposure(bool);
    int setExposureTime(int);
    int getExposureTime();

private:
    int cam_id;
};

#endif // FOCUS_H
```

b) focus.cpp

```
#include "focus.h"

Focus::Focus(const char* device)
{
    int ret = c_init();
    if(ret) cerr << "Unable to c_init (%d)." << endl;
    cam_id = c_open_device(device);
```

```
    setAutoFocus(0);
    setFocus(0); //Bug fix.
    setAutoExposure(false);
}

Focus::~Focus()
{
    c_close_device(cam_id);
    c_cleanup();
}

int Focus::setAutoFocus(int val)
{
    CControlValue value;
    value.value = val;
    int ret = c_set_control(cam_id, CC_AUTO_FOCUS, &value);
    return ret;
}

int Focus::setFocus(int val)
{
    CControlValue value;
    value.value = val;
    int ret = c_set_control(cam_id, CC_FOCUS_ABSOLUTE, &value);
    return ret;
}

int Focus::setAutoExposure(bool status)
{
    CControlValue value;
    if(status)
        value.value = 3;
    else
        value.value = 1;
    int ret = c_set_control(cam_id, CC_AUTO_EXPOSURE_MODE, &value);
    return ret;
}

int Focus::setExposureTime(int val)
{
    CControlValue value;
    value.value = val;
    int ret = c_set_control(cam_id, CC_EXPOSURE_TIME_ABSOLUTE, &value);
    return ret;
}
```

9. REFERENCES

Ahn, S. H., Choi, J. W., Doh, N. L., & Chung, W. K. (2008). A practical approach for EKF-SLAM in an indoor environment: fusing ultrasonic sensors and stereo camera. *Autonomous Robots, 24*(3), 315-335. doi: 10.1007/s10514-007-9083-2

Ahn, Sunghwan, Lee, Kyongmin, Chung, Wan Kyun, & Oh, Sang-Rok. (2007). SLAM with visual plane: Extracting vertical plane by fusing stereo vision and ultrasonic sensor for indoor environment *Proceedings of the 2007 Ieee International Conference on Robotics and Automation, Vols 1-10* (pp. 4787-4794).

Alahi, A., Ortiz, R., & Vandergheynst, P. (2012). FREAK: Fast Retina Keypoint. *2012 Ieee Conference on Computer Vision and Pattern Recognition (Cvpr)*, 510-517.

Altera Corp. (2014a). Nios II Processor: The World's Most Versatile Embedded Processor. Retrieved April 28, 2014, from http://www.altera.com/devices/processor/nios2/ni2-index.html

Altera Corp. (2014b). SoC Overview. Retrieved February 25, 2014, from http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html

Amaricai, A., Gavriliu, C. E., & Boncalo, O. (2014, 2-4 Sept. 2014). *An FPGA sliding window-based architecture harris corner detector.* Paper presented at the Field Programmable Logic and Applications (FPL), 2014 24th International Conference on.

Analog Devices Inc. (2010). ADAU1761 Datasheet. from http://www.analog.com/static/imported-files/data_sheets/ADAU1761.pdf

Ascani, A., Frontoni, E., Mancini, A., & Zingaretti, P. (2008). *Feature group matching for appearance-based localization*. New York: Ieee.

Aulinas, J., Petillot, Y., Salvi, J., & Llado, X. (2008). The SLAM problem: a survey. *Artificial Intelligence Research and Development, 184*, 363-371. doi: Doi 10.3233/978-1-58603-925-7-363

Avnet Inc. (2014). ZedBoard Getting Started Guide. from http://www.zedboard.org/sites/default/files/documentations/GS-AES-Z7EV-7Z020-G-V7.pdf

Aydogdu, M. F., Demirci, M. F., & Kasnakoglu, C. (2013, 12-14 Dec. 2013). *Pipelining Harris corner detection with a tiny FPGA for a mobile robot.* Paper presented at the Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference on.

Bailey, Donald G. (2011). *Design for embedded image processing on FPGAs*. Singapore: John Wiley & Sons (Asia).

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). SURF: Speeded up robust features. *Computer Vision - Eccv 2006 , Pt 1, Proceedings, 3951*, 404-417.

Birem, M., & Berry, F. (2012, 20-23 May 2012). *FPGA-based real time extraction of visual features.* Paper presented at the Circuits and Systems (ISCAS), 2012 IEEE International Symposium on.

Bouguet, J. (2013). Camera Calibration Toolbox for Matlab. Retrieved May 14, 2014, from http://www.vision.caltech.edu/bouguetj/calib_doc/

Callet, P. L. (2010). IRCCyN IVC Quality Assessment of Stereoscopic Images. Retrieved May 15, 2014, from http://www.irccyn.ec-nantes.fr/~lecallet/platforms.htm

Calonder, M., Lepetit, V., Strecha, C., & Fua, P. (2010). BRIEF: Binary Robust Independent Elementary Features. In K. Daniilidis, P. Maragos & N. Paragios (Eds.), *Computer Vision-Eccv 2010, Pt Iv* (Vol. 6314, pp. 778-792).

Choi, Jinwoo, Lee, Kyoungmin, Ahn, Sunghwan, Choi, Minyong, & Chung, Wan Kyun. (2006). *A practical solution to SLAM and navigation in home environment*.

Chu, Pong P. (2008a). *FPGA prototyping by Verilog examples : Xilinx Spartan -3 version*. Hoboken, N.J.: J. Wiley & Sons.

Chu, Pong P. (2008b). *FPGA prototyping by VHDL examples : Xilinx Spartan-3 version*. Hoboken, N.J.: Wiley-Interscience.

Crockett, Louise Helen, Elliot, Ross, Enderwitz, Martin, & Stewart, Robert. (2014). The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC.

Di Carlo, S., Gambardella, G., Prinetto, P., Rolfo, D., Trotta, P., & Lanza, P. (2013, 2-4 Sept. 2013). *FEMIP: A high performance FPGA-based features extractor & matcher for space applications*. Paper presented at the Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on.

Digia plc. (2014). QtNetwork Module. Retrieved May 25, 2014, from qt-project.org/doc/qt-4.8/qtnetwork.html

Digilent Inc. (2013). Embedded Linux Hands-on Tutorial - ZedBoard. Retrieved April 28, 2014, from http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,1028&Prod=ZEDBOARD

Durrant-Whyte, H., & Bailey, T. (2006). Simultaneous localization and mapping: Part I. *Ieee Robotics & Automation Magazine, 13*(2), 99-108. doi: 10.1109/mra.2006.1638022

Fix, Evelyn, & Hodges Jr, Joseph L. (1951). Discriminatory analysis-nonparametric discrimination: consistency properties: DTIC Document.

Fritsch, Jannik, Kuehnl, Tobias, & Geiger, Andreas. (2013). A New Performance Measure and Evaluation Benchmark for Road Detection Algorithms *International Conference on Intelligent Transportation Systems (ITSC)*.

Geiger, Andreas. (2015). The KITTI Vision Benchmark Suite. Retrieved February 2, 2015, from http://www.cvlibs.net/datasets/kitti/

Geiger, Andreas, Lenz, Philip, Stiller, Christoph, & Urtasun, Raquel. (2013). Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)*.

Geiger, Andreas, Lenz, Philip, & Urtasun, Raquel. (2012). Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Gross, Changil Kim; Henning Zimmer; Yael Pritch; Alexander Sorkine-Hornung; Markus. (2012). Scene Reconstruction from High Spatio-Angular Resolution Light Fields. Retrieved February 2, 2015, from http://www.disneyresearch.com/project/lightfields/

Guivant, J. E., & Nebot, E. M. (2001). Optimization of the simultaneous localization and map-building algorithm for real-time implementation. *Ieee Transactions on Robotics and Automation, 17*(3), 242-257. doi: Doi 10.1109/70.938382

Harris, Chris, & Stephens, Mike. (1988). *A combined corner and edge detector*. Paper presented at the Alvey vision conference.

Hartley, Richard, & Zisserman, Andrew. (2003). *Multiple view geometry in computer vision* (2nd ed.). Cambridge ; New York: Cambridge University Press.

Hartmann, J., Klussendorff, J. H., & Maehle, E. (2013). *A Comparison of Feature Descriptors for Visual SLAM*. New York: Ieee.

Ho, K. (2012). A survey of algorithms for star identification with low-cost star trackers. *Acta Astronautica, 73*, 156-163. doi: DOI 10.1016/j.actaastro.2011.10.017

IEEE. (2006). IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 0_1-560. doi: 10.1109/IEEESTD.2006.99495

IEEE. (2007). IEEE Standard VHDL Language Reference Manual Amendment 1: Procedural Language Application Interface. *IEEE Std 1076c-2007 (Amendment to IEEE Std 1076-2002)*, c1-214. doi: 10.1109/IEEESTD.2007.4299594

ImageMagick Studio LLC. (2015). ImageMagick: Convert, Edit, Or Compose Bitmap Images. Retrieved February 2, 2015, from http://www.imagemagick.org/

Jacob, B., & Guennebaud, G. (2014). Eigen is a C++ template library for linear algebra. Retrieved May 25, 2014, from http://eigen.tuxfamily.org/index.php?title=Main_Page

Jung, I. K., & Lacroix, S. (2001). *A robust interest points matching algorithm*. Los Alamitos: Ieee Computer Soc.

Kerrisk, Michael. (2015a). CLOCK_GETRES(2) - Linux Programmer's Manual. Retrieved February 2, 2015, from http://man7.org/linux/man-pages/man2/clock_gettime.2.html

Kerrisk, Michael. (2015b). MMAP(2) - Linux Programmer's Manual. Retrieved February 17, 2015, from http://man7.org/linux/man-pages/man2/mmap.2.html

Laganière, Robert. (2011). *OpenCV 2 Computer Vision Application Programming Cookbook: Over 50 Recipes to Master this Library of Programming Functions for Real-time Computer Vision*: Packt Publishing Ltd.

Lee, S., & Lee, S. (2013). Embedded Visual SLAM Applications for Low-Cost Consumer Robots. *Ieee Robotics & Automation Magazine, 20*(4), 83-95. doi: Doi 10.1109/Mra.2013.2283642

Leutenegger, S., Chli, M., & Siegwart, R. Y. (2011). BRISK: Binary Robust Invariant Scalable Keypoints. *2011 Ieee International Conference on Computer Vision (Iccv)*, 2548-2555.

libwebcam. (2014). libwebcam. Retrieved May 23, 2014, from http://sourceforge.net/projects/libwebcam/

Liebe, Carl Christian. (1993). Pattern recognition of star constellations for spacecraft applications. *Aerospace and Electronic Systems Magazine, IEEE, 8*(1), 31-39.

Linaro. (2014). Linaro: open source software for ARM SoCs. Retrieved May 23, 2014, from http://www.linaro.org/

Logitech. (2014). HD Webcam C525. Retrieved May 23, 2014, from http://www.logitech.com/en-us/product/hd-webcam-c525

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision, 60*(2), 91-110. doi: Doi 10.1023/B:Visi.0000029664.99615.94

Montemerlo, M., Thrun, S., Koller, D., Wegbreit, B., Aaai, & Aaai. (2002). *FastSLAM: A factored solution to the simultaneous localization and mapping problem*.

Montemerlo, Michael, Thrun, Sebastian, Roller, Daphne, & Wegbreit, Ben. (2003). *FastSLAM 2.0: an improved particle filtering algorithm for simultaneous localization and mapping that provably converges*. Paper presented at the Proceedings of the 18th international joint conference on Artificial intelligence, Acapulco, Mexico.

Mortensen, E. N., Deng, H., & Shapiro, L. (2005). A SIFT descriptor with global context. In C. Schmid, S. Soatto & C. Tomasi (Eds.), *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol 1, Proceedings* (pp. 184-190). Los Alamitos: Ieee Computer Soc.

Munguia, R., Castillo-Toledo, B., & Grau, A. (2013). A Robust Approach for a Filter-Based Monocular Simultaneous Localization and Mapping (SLAM) System. *Sensors, 13*(7), 8501-8522. doi: Doi 10.3390/S130708501

Na, Meng, & Jia, Peifa. (2006). *A survey of all-sky autonomous star identification algorithms*.

NVIDIA Corp. (2014). CUDA Parallel Computing Platform Retrieved February 25, 2014, from http://www.nvidia.com/object/cuda_home_new.html

OpenCV. (2014). OpenCV Open Source Computer Vision. Retrieved May 14, 2014, from http://opencv.org/

OpenCV. (2015). Common Interfaces of Feature Detectors. Retrieved February 2, 2015, from http://docs.opencv.org/modules/features2d/doc/common_interfaces_of_feature_detectors.html

Padgett, C., & KreutzDelgado, K. (1997). A grid algorithm for autonomous star identification. *Ieee Transactions on Aerospace and Electronic Systems, 33*(1), 202-213. doi: 10.1109/7.570743

Paz, L. M., Pinies, P., Tardos, J. D., & Neira, J. (2008). Large-Scale 6-DOF SLAM With Stereo-in-Hand. *Ieee Transactions on Robotics, 24*(5), 946-957. doi: 10.1109/tro.2008.2004637

Pedroni, Volnei A. (2010). *Circuit design and simulation with VHDL* (2nd ed.). Cambridge, Mass.: MIT Press.

Rosten, Edward, & Drummond, Tom. (2006). Machine learning for high-speed corner detection *Computer Vision–ECCV 2006* (pp. 430-443): Springer.

Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. (2011). ORB: an efficient alternative to SIFT or SURF. *2011 Ieee International Conference on Computer Vision (Iccv)*, 2564-2571.

Scharstein, D. (2006). Middlebury Stereo Datasets. Retrieved May 23, 2014, from http://vision.middlebury.edu/stereo/data/

Schleicher, D., Bergasa, L. M., Ocana, M., Barea, R., & Lopez, E. (2010). Real-time hierarchical stereo Visual SLAM in large-scale environments. *Robotics and Autonomous Systems, 58*(8), 991-1002. doi: DOI 10.1016/j.robot.2010.03.016

Siegwart, Roland, Nourbakhsh, Illah Reza, & Scaramuzza, Davide. (2011). *Introduction to autonomous mobile robots* (2nd ed.). Cambridge, Mass.: MIT Press.

Solà, Joan. (2013, 01/17/2013). Simultaneous localization and mapping with extended Kalman filter. Retrieved March 7, 2014, from http://www.iri.upc.edu/people/jsola/JoanSola/objectes/curs_SLAM/SLAM2D/SLAM%20course.pdf

Spratling, Benjamin B, & Mortari, Daniele. (2009). A survey on star identification algorithms. *Algorithms, 2*(1), 93-107.

Taylor, Adam P. (2015). A Double-Barreled Way to Get the Most from Your Zynq SoC. *Xcell Journal, 1*(90), 38-45.

The Mathworks, Inc. (2014). Introducing MATLAB Engine. Retrieved May 25, 2014, from http://www.mathworks.com/help/matlab/matlab_external/introducing-matlab-engine.html

Thrun, S., Liu, Y. F., Koller, D., Ng, A. Y., Ghahramani, Z., & Durrant-Whyte, H. (2004). Simultaneous localization and mapping with sparse extended information filters. *International Journal of Robotics Research, 23*(7-8), 693-716. doi: Doi 10.1177/0278364904045479

Thrun, Sebastian. (2002). Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, 1-35.

Thrun, Sebastian, Burgard, Wolfram, & Fox, Dieter. (2005). *Probabilistic robotics*. Cambridge, Mass.: MIT Press.

Todt, Eduardo. (2005). *Visual Landmark Detection for Navigation in Outdoor Environments.* Institut d'Organització i Control de Sistemes Industrials, Barcelona.

Tsai, R. Y. (1987). A VERSATILE CAMERA CALIBRATION TECHNIQUE FOR HIGH-ACCURACY 3D MACHINE VISION METROLOGY USING OFF-THE-SHELF TV CAMERAS AND LENSES. *Ieee Journal of Robotics and Automation, 3*(4), 323-344.

University of Pennsylvania. (2013). Computer Power Usage. Retrieved April 6, 2015, from https://secure.www.upenn.edu/computing/resources/category/hardware/article/computer-power-usage

Wikipedia. (2015). Lenna. Retrieved February 2, 2015, from http://en.wikipedia.org/wiki/Lenna

Xilinx Inc. (2009). UG627 - XST User Guide. v 11.3. from http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf

Xilinx Inc. (2011). UG761 - AXI Reference Guide. v 13.1. from http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

Xilinx Inc. (2012). UG687 - XST User Guide for Virtex-6, Spartan-6 and 7 Series Devices. 14.1. Retrieved May 25, 2014, from http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/xst_v6s6.pdf

Xilinx Inc. (2013a, August 6, 2013). UG474 - 7 Series FPGAs Configurable Logic Block - User Guide. from http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

Xilinx Inc. (2013b). Zynq-7000 Combined Product Table. from http://www.xilinx.com/publications/prod_mktg/zynq7000/Zynq-7000-combined-product-table.pdf

Xilinx Inc. (2014a). MicroBlaze Soft Processor Core. Retrieved April 28, 2014, from http://www.xilinx.com/tools/microblaze.htm

Xilinx Inc. (2014b). UG473 - 7 Series FPGAs Memory Resources. v 1.10. from http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

Xilinx Inc. (2014c). Zynq-7000 All Programmable SoC. Retrieved February 25, 2014, from http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm

Xilinx Inc. (2015). SDSoC Development Environment. Retrieved April 5, 2015, from http://www.xilinx.com/products/design-tools/sdx/sdsoc.html

Zhang, Z. Y. (2000). A flexible new technique for camera calibration. *Ieee Transactions on Pattern Analysis and Machine Intelligence, 22*(11), 1330-1334. doi: 10.1109/34.888718