UNIVERSIDAD POLITÉCNICA DE
MADRID

FPGA-BASED ACCELERATION OF
TRACKING ALGORITHM TO
IMPROVE PARKINSON'S PATIENTS
QUALITY OF LIFE

Doctorado en Tecnologías y Sistemas de Comunicaciones

Giuseppe Conti

2020

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

DEPARTAMENTO DE SEÑALES, SISTEMAS Y RADIOCOMUNICACIONES.

FPGA-based acceleration of tracking algorithm to improve Parkinson's patients quality of life

Autor: Giuseppe Conti

Director: Dr. David Jiménez Bermejo

El tribunal nombrado para juzgar la tesis arriba indicada el día _____ de _____ de _____
, compuesto de los siguientes doctores:

Presidente: _____

Vocal: _____

Vocal: _____

Vocal: _____

Secretario: _____

Realizado el acto de lectura y defensa de la Tesis doctoral, a _____ de _____
de _____, acuerdan la calificación de:

Calificación: _____

EL PRESIDENTE

EL SECRETARIO

LOS VOCALES

UNIVERSIDAD POLITÉCNICA DE
MADRID

FPGA-BASED ACCELERATION OF
TRACKING ALGORITHM TO
IMPROVE PARKINSON'S PATIENTS
QUALITY OF LIFE

Doctorado en Tecnologías y Sistemas de Comunicaciones

Giuseppe Conti

2020

Summary

The use of electronic devices in biomedical and medical applications is increasing, with the intention of giving doctors and healthcare personnel new tools to monitor and checkup on their patients. All the professional communities involved, have a common purpose: to improve the quality of people's lives.

This thesis presents a study on the optimization of the tracking system designed for patients with Parkinson's disease, tested at a day hospital center. The work performed significantly improves the efficiency of the computer vision based system in terms of energy consumption and hardware requirements. This study analyzes the tracking algorithm in its whole implementation, and it optimizes the performances of the background subtraction by segmenting every frame previously characterized by a Gaussian mixture model (GMM). This module has been determined to be the most demanding part in terms of computational resources, and therefore, this thesis proposes a method for its implementation by means of a low-cost development board based on Zynq XC7Z020 SoC (system on chip). The platform used is the ZedBoard, which combines an ARM Processor unit and an FPGA.

Module optimisation is necessary to achieve performance, consumption and cost requirements. This research has optimised the embedded system, making use of a reconfigurable hardware solution, to implement the GMM module which is the most demanding. For this purpose, a hybrid software/hardware design has been chosen, to deal with the requirements imposed by the architecture that introduces a data type variation over the original software model implementation, without losing flexibility for future adaptations or modifications. Furthermore, the difficulties in integration with the rest of the complete system modules has been resolved, performing the functional tests (real or simulation) necessary for its verification.

This thesis proposes a working methodology using tools that facilitate HW/SW co-design, achieving an implementation that covers the research needs without the drawbacks of a dedicated hardware design. The knowledge required and acquired to perform this work includes expertise from many other fields of study. In fact, the plethora of ingredients needed from other fields must include: research and biomedical skills to be able to understand the problem and find a solution, electrical and electronic expertise to design the architecture, hardware verification techniques to perform the validation of the design, Operating system and Linux expertise to build a system tailored to the board and the project resources, advanced programming skills to create and modify all the code necessary for the algorithmic part, driver development to perform the communication between the CPU and the FPGA, multi-threading

and all types of software optimization to allow the performance improvements to take advantage of the FPGA.

In the chapters of this thesis, all construction stages are detailed, starting from the research phase down to specific technical steps. To summarize: the algorithm is modified to allow its synthesis on the FPGA. In the following stage it has been implemented on the FGPA and connected to the CPU using the proper interfaces.

A driver is developed to connect the algorithm running on the CPU with the part accelerated on the FPGA. All the systems are tested and checked up to find the right setup.

A further optimization strategy is applied to the system improving the RAM usage and the multi-threading architecture of the CPU. The system achieves real-time performance and low-power consumption while accurately performing the target tracking. Finally, the results and findings of this study, validated in real medical settings, are discussed and analyzed within.

Keywords

APSoC, Background Subtraction, Computer Vision, Embedded System, FPGA, GMM, HLS, Human Detection, Image Analysis, Image Processing, Linux, MoG, OpenCV, Patient Privacy, Tracking, VDMA, ZedBoard, ZYNQ-7000.

Resumen

El uso de dispositivos electrónicos para aplicaciones biomédicas y médicas está aumentando, con la intención de dar a los médicos y al personal sanitario, nuevas herramientas para controlar y revisar a sus pacientes. Todas las comunidades profesionales involucradas tienen un propósito común: mejorar la calidad de vida de las personas.

Esta tesis presenta un estudio sobre la optimización de un sistema de seguimiento de personas, diseñado para los pacientes que han desarrollado la enfermedad de Parkinson, probado en un centro de día de la comunidad de Madrid. El trabajo realizado mejora significativamente la eficiencia del sistema de visión artificial, en términos de consumo de energía y necesidad de recursos hardware. Este estudio analiza el algoritmo de seguimiento en toda su implementación y, optimiza las prestaciones de la sustracción de fondo segmentando cada cuadro previamente caracterizado por un Gaussian mixture model (GMM). Este módulo ha sido identificado como la parte más exigente en términos de recursos de computación, y por lo tanto, esta tesis propone un método más eficiente para su implementación mediante una placa de desarrollo de bajo costo basada en Zynq XC7Z020 SoC (System on Chip). La plataforma utilizada es la ZedBoard, que combina un procesador ARM y una FPGA.

La optimización del módulo es necesaria para cubrir los requisitos de prestaciones, consumo y coste. En esta investigación se va a optimizar el sistema embebido, haciendo uso de hardware reconfigurable para el módulo GMM que es el más demandante. Para ello, se ha realizado un diseño híbrido software/hardware, solventando los requisitos impuestos por la arquitectura de trabajo que obliga a variar los tipos de datos que integraba el modelo software original en su implementación, pero añadiendo flexibilidad para adaptaciones o modificaciones futuras. Y solucionando las dificultades en la integración con el resto de módulos del sistema completo, y las pruebas funcionales (reales o con simulación) necesarias para su verificación.

En esta tesis se plantea una metodología de trabajo utilizando herramientas que faciliten el co-diseño HW/SW, obteniendo una implementación que cubra las necesidades sin las penalizaciones de un diseño dedicado. El conocimiento requerido y adquirido para realizar este trabajo incluye más áreas de interés. De hecho, todos los elementos necesarios incluyen: investigación y experiencia en el área biomédica para poder entender el problema y encontrar una solución, conocimientos avanzados de electrónica para diseñar la arquitectura, técnicas de verificación de hardware para realizar la validación del diseño, experiencia en sistemas operativos, especialmente en Linux, para diseñar un sistema adaptado a la placa y a los recursos del proyecto, habilidades avanzadas de programación para crear y modificar todo el código necesario para

la parte algorítmica, desarrollo de controladores para realizar la comunicación entre la CPU y la FPGA, conocimiento de procesamiento multi-hilo y todo tipo de optimización de software para permitir la mejora del rendimiento aprovechando la FPGA.

En los capítulos de esta tesis se detallan todas las etapas para realizarla, desde la fase de investigación hasta todas las etapas técnicas. A continuación se presenta un breve resumen de todo el proceso. El algoritmo se modifica para permitir su síntesis en la FPGA. En la siguiente etapa se ha implementado el algoritmo sobre de la FGPA y se ha realizado la comunicación con la CPU utilizando las interfaces adecuadas. Se ha desarrollado un controlador para conectar el algoritmo que corre en la CPU con la parte acelerada en la FPGA. Todos los sistemas se comprueban para encontrar la configuración correcta. Una estrategia de optimización adicional se aplica al sistema mejorando el uso de la RAM y la arquitectura multi-hilo de la CPU. El sistema logra un rendimiento en tiempo real y un bajo consumo de energía mientras realiza el seguimiento de personas con precisión. Finalmente, los resultados y los logros de este estudio, validados en entornos médicos reales y en publicaciones, son presentados y analizados en la ultima parte de este trabajo.

Palabras clave

APSoC, Sustracción de fondo, Visión Artificial, Sistema Embebido, FPGA, GMM, HLS, Detección de personas, Análisis de imágenes, Procesamiento de imágenes, Linux, MoG, OpenCV, Privacidad del paciente, Seguimiento, VDMA, ZedBoard, ZYNQ-7000

Index

1	Introduction	1
1.1	Motivation	2
1.2	Methodology	3
1.3	Problem Statement	4
1.4	Summary of Contributions	8
1.5	Outline of the dissertation	9
2	Scientific and Technological Background	10
2.1	State of the Art	10
2.2	FPGA and Zynq Architecture	13
2.3	ZedBoard	16
2.4	Vivado HLS, Vivado and XSDK	17
2.4.1	Vivado	18
2.4.2	Vivado HLS	19

2.4.3	XSDK	21
2.5	OpenCV	23
3	Architecture and Workflow	24
3.1	Design Methodology	24
3.2	Proposed Background Sub-tractor Algorithm	26
3.3	Model Architecture Implementation	28
3.4	Algorithm customization for acceleration	30
3.5	Vivado HLS Design Porting	35
4	Hardware Acceleration	37
4.1	Hardware Model	41
4.2	Functional Model	50
4.3	Operating System Customization	52
4.4	Driver Implementation	57
4.4.1	VMDA configuration	59
4.4.2	Video Stream Data Read and Write transactions	62
4.4.3	Background Subtractor Configuration	63
4.4.4	Video Stream Data Read and Write transactions	64
4.4.5	Parameters Calculation and Automating	66
5	Test and Validation	69

5.1	Hardware	69
5.1.1	Vivado HLS pre and post synthesis	71
5.1.2	Vivado FPGA synthesis verification	73
5.2	Software	80
5.3	Experimental Setup	84
5.4	Testing Environment in APM	87
6	Results	89
7	Conclusions	100
	References	104

Index of figures

1.1	Typical application scenario.	4
1.2	ICT4Life project structure.	6
1.3	System elements considered: the Zenith camera (source), the ZedBoard (processing system) and control server (sink).	7
2.1	FPGA internal structure from [1]	14
2.2	Zynq 7000 architecture from [2]	15
2.3	Zynq Theoretical Bandwidth from [3]	16
2.4	ZedBoard photo with description from [4]	17
2.5	Xilinx System Debugger workflow from [5]	22
3.1	From Software library to Hardware IP workflow.	25
3.2	Complete algorithm solution implemented in Software.	28
3.3	Software processing workflow to be modified and implemented into the FPGA.	29
3.4	From dynamic Mat type to C++ vector of struct.	31

4.1	Vivado HLS Algorithm starting point report.	39
4.2	Vivado HLS Algorithm first valid estimation.	39
4.3	Vivado HLS Algorithm Performance estimation.	40
4.4	Complete connectivity diagram from [3]	42
4.5	High Performance (AXI_HP) Connectivity from [3]	42
4.6	Vivado Hardware internal structure.	43
4.7	VDMA interface Operation from [6]: Left AXI4 Memory Map to AXI4-Stream Read; Right AXI4-Stream to AXI4 Memory Map Write;	45
4.8	VDMA Configuration parameters screenshot	46
4.9	VDMA Performance from [6]	46
4.10	Vivado address configuration	47
4.11	FPGA Utilization: Left RGB version of the design; Right Gray-scale version of the design;	48
4.12	Vivado timing analysis: Top RGB version of the design; Bottom Gray-scale version of the design;	48
4.13	Vivado power analysis: Top RGB version of the design; Bottom Gray-scale version of the design;	49
4.14	Complete Zynq structure used in this work.	51
4.15	Ram Distribution and utilization on the ZedBoard.	56
4.16	Userspace I/O driver structure from [7]	57
4.17	Direct Memory Access architecture from [8]	59
4.18	VDMA control register layout from [6]	60

4.19	VDMA status register layout from [6]	60
5.1	Engineers time dedication performing Verification from [9]	70
5.2	BackGround Subtractor algorithm debug on Vivado HLS	71
5.3	Vivado Hardware Test Diagram	74
5.4	FPGA AXI stream and VDMA verification	75
5.5	Vectors Sum Test algorithm for AXI master debug	76
5.6	Frame Buffering strategy used to move data in this work.	82
5.7	Multi threading Frame Buffering.	83
5.8	Evolution of the Number of frames to buffer in RAM.	86
5.9	Camera Installation in APM.	87
5.10	Camera live view.	88
6.1	Process measurement comparison.	93
6.2	Images comparison (a-t).	97

Glossary

APSoC	All Programmable System on Chip
ACP	Accelerator Coherency Port
AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced Risc Machines
AXI	Advanced eXtensible Interface
AXI.GP	Advanced eXtensible Interface General Purpose
AXI.HP	Advanced eXtensible Interface High Performance
BGS	Background Subtractor
BRAM	Block RAM
CLB	Configurable Logic Block
CV	Computer Vision
DMA	Direct Memory Access
DT	Device Tree
DRC	Design Rule Check
FPGA	Field Programmable Gate Array
FPS	Frame Per Second
FSBL	First Stage Boot Loader
GMM	Gaussian Mixture Model
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HDL	Hardware Description Language
HDMI	High Definition Multimedia Interface
HLS	High Level Synthesis
IDE	Integrated Development Environment
IP	Intellectual Property
MM2S	Memory Mapped to Stream
MoG	Mixture of Gaussian
OTG	On The Go
PL	Programmable Logic
PS	Processing System
RGB	Red Blue Green
S2MM	Stream to Memory Mapped

SD	Secure Digital
SoC	System on Chip
SoF Start Of Frame	
SoM	System on Module
UART	Universal Asynchronous Receive Transmit
UI	User Interface
UIO	userspace I/O
USB	Universal Serial Bus
VDMA	Video Direct Memory Access

Chapter 1

Introduction

Nowadays human pedestrian tracking and object detection systems are covering an important role in real life applications. These systems attract considerable interest at the industrial and scientific communities, and a lot of methods have been proposed and developed. Moreover, computer vision is constantly growing, and its deployment is reaching embedded systems. New customizable hardware solutions, operating systems and software applications, have already been developed for research purposes concluding in robust prototypes that have derived in ready to market solutions. This kind of systems are integrated in a single chip and embedded on a board. This emerging technology known as FPGA-SoC (Field Programmable Gate Array - System on Chip) and FPGA-SoM (Field Programmable Gate Array - System on Module) consists in a chip that encapsulates both FPGA and a processor (such as an ARM core) that share the entire system resources (RAM Memory, connections, peripherals, SD storage, etc). These technologies, already adopted within cameras for sensor acquisition and coding stages [10, 11, 12], allow to achieve real time performance for complex Computer Vision systems capable to provide flexibility, high precision and low power consumption. The variety of applications that can range is wide, such as Advanced Driver Assistance System (ADAS), industrial supervision application, video surveillance, traffic detection, pedestrian detection and tracking, patient monitoring, biomedical devices, etc. Biomedical health and life science represent a special sector where the use of SoC and SoM is becoming popular. Thanks to their features and flexibility, these embedded platforms provide an excellent support to improve healthcare and biomedical-related processes. This is possible due the advantage of using a SoC or a SoM, that allows the combination of SW existing libraries and HW acceleration

in a single compact device. As a consequence, the use of this technology allows to reduce size and power consumption of traditional medical and biomedical equipment. An example of biomedical device made on an FPGA is showed in [13], where a 3D ultrasound sensing device is developed. The proposed sensor is able to perform real-time 2D and 3D complete ultrasound reconstruction with a power consumption of around 6 W. The platform enables teleultrasonography by allowing the exploitation of ultrasound diagnosis by any untrained operator in remote rural areas, underdeveloped regions, rescue scenarios, obtaining diagnostic results very close to a standard teleultrasonography device, that is a bulky, high power consumption and expensive machine available only in hospitals.

1.1 Motivation

Modern times see the continuous requirement of new technologies always faster, with more capabilities and low power requirements. In our homes, that are everyday smarter including more sensors and electronic applications, the scalability of the devices is becoming more important day by day. This need is generated from the request of powerful systems, able to perform complex operations, that help human beings to undertake from the simple everyday routine up to learn new things. A special opportunity that technology is giving us consists into improve the health monitoring; in fact, it is common to see heart rate sensors, smartphone and smart watches, monitoring for example training activity or tracking the steps walked or the sleeping time. The technology has reached hospital and care centers, the collaboration between doctors, physiotherapists, care givers, technicians and engineers is encouraged due to new projects and initiatives allowing the improvement of the health care systems. The innovation opportunities offered by this biomedical sector are endless, because the development of these technologies has started in recent years. Thanks to these developments it is possible to improve patients quality of life, providing to the doctors and medical professionals new tools very useful to assist patients. Furthermore as the health care systems are in constant growing, this includes adding new functionalities everyday more complex. In terms of hardware and electronic systems, the need of devices capable to satisfy the these requirements, is translated into a technology that could be easily integrated with sensors and human prosthesis for example. Other important aspects to take into account are represented by the low power and real time capabilities. The first aspect is important because it allows to create wearable and portable devices powered by batteries, while the interaction with patients in real time could be required depending on the application. The FPGAs represent a technology with great capabilities that allows this integration. In fact these devices are low power, easy to inte-

grate with sensors and electronic devices, came with an integrated CPU that allows running an operating system, and are suitable to accelerate software applications to reach real time. Taking FPGAs as base to start developing and accelerating a real time application dedicated to Parkinson's patients, represents a great challenge that will be covered in this thesis.

1.2 Methodology

To achieve the goal of this thesis, detailed in next section, a multidisciplinary knowledge is required. Various stages are needed starting from the requirements gathering, research stage, programming skills, hardware design capabilities, operating system knowledge, Linux driver development, multi-threading programming, and network communication. The first stage is represented by the biomedical analysis, where a complete and accurate observation of the day hospital environment has been performed, talking with the doctors and physiotherapists of the center to recollect information about the Parkinson's patients, the parameters to monitor and individuate where to place the sensors and the cameras. The information provided by medical professionals are very important because not only is useful to identify the system requirements, but also it allows to determine important recommendations to install all the electronic devices for data recollection, placing them in the right position and in a non-intrusive way. In fact an important requirement is to install all the equipment respecting the Parkinson's patients safety, considering the reduction of risk of unnecessary harm associated with health care, and reducing also the visibility of the electronic devices to avoid disturbing the patients during their stay in the center. Once the requirements gathering has been done, the state of art is analyzed to choose the suitable tracking algorithm for indoor environments. Also in this stage its portability to the FPGA architecture is analyzed. In fact the final decision also depends on the complete algorithm portability and integration with the entire system. The next stage is represented by the definition of the development environment. In fact all the technical tools needed to develop this thesis are analyzed and chosen. This is a deep technical analysis that starts from the camera type and follows with the FPGA board selection, the programming language to use, the tools needed for the algorithm acceleration, the operating systems. Also all the communications between the FPGA the CPU and the complete system are evaluated. The algorithm time analysis follows. In this stage all the algorithm parts are inspected in terms of resources needed, time of execution and computational load. The main objective is to identify the slowest and more demanding part of the code to accelerate and implement it on the FPGA. The algorithm is then modified and prepared to be ported on the FPGA. The next stage is represented by the algorithm acceleration, identifying the

bottlenecks and applying all the optimizations to meet the requirements in term of time and performance. Once the algorithm has been accelerated, the last stages that follow are the integration and test. The integration stage consists into integrate the accelerated algorithm contained into the FPGA with the CPU and the rest of the system. Also in this process other bottlenecks are individuated and optimized, in fact the testing stage is connected with the integration. An exhaustive battery of test and the right verification strategy should be applied to the entire embedded system, to check all the complex functionalities that it is able to perform. Furthermore is very important the iteration of these two stages to resolve all the issues and optimize all the bottleneck, up to meet the real time condition.

1.3 Problem Statement

This work has been carried out within the larger framework of the ICT4Life project [14]. ICT4Life aims to develop a complete platform with the goal to deliver an innovative service to patients affected by dementia-related diseases, mainly Parkinson's and Alzheimer's. By integrating video analysis capabilities into the platform, visual monitoring of patient behavior and patient progress is enabled.

Figure 1.1: Typical application scenario.

This type of information is useful for health professionals and both formal and informal caregivers, since it helps them to estimate the severity and evolution of the patients' health and alert them in case of issues related to Parkinson's, such as accidents resulting from falls. The overall system has been developed following a user-centered methodology and validated in real life scenarios, such as the one shown in Figure 1.1. The platform, described in [15], consists of different sensors: visual, motion and depth, that together form a multi-sensor tracking system [16]. The types of devices vary based on the requirements analyzed for the different scenarios and the typology of patients, including:

- Wearable bracelets providing patients' motion data from embedded gyroscope and accelerometer, and health information, such as heart rate and skin temperature [17].
- Location sensors, a binary non-intrusive sensor placed to delimit the patient's behavior analysis zone.
- Depth vision cameras (Microsoft Kinect v2) for extraction of depth information to integrate deep motion analysis to monitor user activity, status and evolution.
- 360-degree panoramic camera (Zenith), covering targeted areas such as halls and living rooms in senior centers. It provides better images to deploy the tracking algorithms and avoid occlusions.
- Wireless sensor network (WSN) anchors monitoring of the radio signals from patients' wearables in a non-intrusive manner. They provide extra info to determine the trajectories derived from the camera images. [18]. Anchors are placed on the roof in the same areas the cameras are installed.

All the information coming from the sensors is integrated and fused in an adaptive multi-sensory application. The real-time collected data are synchronized and organized in this application. To fulfill ethical requirements, the system allows safe storage of information, specifically of the processed data, with a privacy-by-design approach. Figure 1.2 depicts the architecture and interconnection of sensors.

In accordance with the objectives of the project, the real-time requirements, and the necessary flexibility and openness were fundamental in the selection of the hardware framework to be used for the implementation. The Zedboard is a low-cost development board for integrating implementations over the Xilinx Zynq combining an FPGA and a processor on a single chip. It establishes very fast and programmable connections between the processor and the FPGA

ideal for real time video processing. Furthermore, it is flexible by not imposing tight requirements on developers. More concretely, the work in this thesis is focused on human pedestrian tracking using the Zenith camera (depicted in the orange dotted square of picture 1.2). An important requirement is the need to track the patients while also respecting their privacy. To meet this need, the camera stream is not recorded as is. This has been achieved by the design of a system module to calculate the spatio-temporal coordinates/trajectories of the patients.

Figure 1.2: ICT4Life project structure.

The patients with cognitive diseases such as Parkinson's, have problems regarding mobility and often do not actively move. The trajectory data are relevant because they provide daily estimations about how much the patients are moving. This information helps the doctors to understand the patient's status, and to modify the dosages of their medications based on these observations. To balance the aims of the video monitoring system, the context of the

patients and the real-time needs, the video frame rate has been fixed at 15 fps. This selection provides enough frames to capture most actions granularly while minimizing processing costs.

Figure 1.3: System elements considered: the Zenith camera (source), the ZedBoard (processing system) and control server (sink).

Figure 1.3 shows the different elements involved in the module previously described, where the camera is connected with the FPGA board, which processes the video stream and sends the data to the server. The system communicates via an Ethernet network connection.

The camera used in this project is a fish-eye camera by Vivotek; more specifications are available in [19]. There are two basic features that should be taken into account when developing a real-time video processing solution: the acquisition frame rate and the input resolution of the video. The sensory device chosen can work with different resolutions (results are provided in the following sections) and a frame rate of 15 fps. Therefore, we have assumed that to grant real-time processing each frame should be processed at around 66.67 ms or less. This requirement represents the time to meet by the algorithm.

1.4 Summary of Contributions

This thesis describes the design of an embedded system that uses OpenCV Libraries to track patients with Parkinson's disease, tested in the Asociación Parkinson Madrid (APM) ¹ day hospital center. The main contribution of this thesis is detailed as follows:

- The design of a novel architecture with a special combination of HW/SW capabilities, where a mixture of Gaussian (MoG) and a Kalman filter trackers were employed, optimizing the background subtraction to be executed on the FPGA. This marks an important state-of-the-art advance from previous studies, by ensuring the complete interaction and management of the video stream. While other studies focus on the background subtraction, or on the processing part, in this work we consider the complete process chain, including the relevant times needed by the input and the output stages. In fact, this hybrid system aims to accelerate the software solution designed for the Parkinson's patients, taking advantage of the Zynq platform flexibility.
- A special effort focuses on the patients privacy, because the system cannot store the streamed video. In fact, the Ethernet camera video stream input is connected to the SoC, that process the data locally. It performs the tracking and estimates the patients coordinates at every frame. The output of the SoC is represented by the patients coordinates and a tracking ID that are sent to the system, without a video streaming satisfying the privacy requirements.
- This system has been verified in APM, thanks to the effort of doctors, nurses, physiotherapists and the patients. This represent a great achievement in a real environment.

The principal results of this thesis has been published in:

G. Conti, M. Quintana, P. Malagón, and D. Jiménez, “*An fpga based tracking implementation for parkinson's patients*”, *Sensors*, vol. 20, p. 3189, Jun 2020. [20]

while its contribution to the rest of the system has been published in:

A. Belmonte-Hernández, V. Solachidis, T. Theodoridis, G. Hernández-Peñaloza, G. Conti, N. Vretos, F. Alvarez, and P. Daras, “*Person tracking association using multimodal systems*”, in

¹Asociación Parkinson Madrid (APM) <http://www.parkinsonmadrid.org>

2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), pp. 1–6, Aug 2017. [16]

A. Belmonte-Hernández, G. Hernández-Peñaloza, F. Alvarez, and G. Conti, “*Adaptive fingerprinting in multi-sensor fusion for accurate indoor tracking*”, IEEE Sensors Journal, vol. 17, pp. 4983–4998, Aug 2017. [18]

and in [14, 15, 17].

A part of the FPGA hardware verification methodology has been published in:

G. Conti, C. Kyrkou, T. Theocharides, G. Hernández-Peñaloza, and D. Jiménez, “*A novel fpga-based test-bench framework for sdi stream verification*”, EURASIP Journal on Image and Video Processing, vol. 2020, p. 31, Jul 2020. [21]

1.5 Outline of the dissertation

This thesis is organized as follows:

- Chapter 2, introduces the state of art and a summary about the technological resources and utilities used in this work.
- In Chapter 3, the proposed algorithm is detailed, also the customization method to convert the algorithm from software to hardware and its introduction into the FPGA is presented.
- Chapter 4, describes the Hardware implementation of the architecture and its connection with the system.
- Chapter 5, the Verification of the system and its experimental setup is presented.
- Chapter 6, the results of the system are illustrated.
- Section 7, Finally, conclusions and future work are drawn.

Chapter 2

Scientific and Technological Background

This chapter contains the state of art and a brief summary about the technologies used in the development of this thesis. The information that has been included is the minimum necessary to understand the hardware structure and its utilization. For more information, it is possible to consult the references that are cited in this chapter.

2.1 State of the Art

A literature survey about hardware visual tracker implementations is presented in [22] where the authors introduce various solutions and their relative results obtained in the last two decades. This survey highlights that most of the tracking approaches developed recently based on an FPGA architecture lack heterogeneous FPGA-SoC and FPGA-SoM implementations including more kind of processing devices.

A comparison between CPUs, GPUs and FPGAs is presented in [23] and also shows the advantages of using FPGAs instead of GPUs or CPUs. The work also compares three different algorithms GMM (Gaussian mixture model), ViBE (visual background extractor) and PBAS (pixel-based adaptive segmenter), which are all excellent options for developing the tracking

systems. An interesting part of the study focuses on the precision and the HW accuracy, including a complete comparison based on those metrics.

A recent example of FPGA-only proposals is represented by this work [24], where a background subtraction on a Spartan 3 FPGA is presented. The authors have written the Intellectual Property (IP) in Verilog, and they propose a method to improve its performance. Frames whose resolution is 128*128 pixels are processed, and an implementation developed in Matlab is used to convert and introduce the image in the system without specifying the interface.

A Mixture of Gaussian (MoG) Background Subtraction is presented in [25]. The authors use a FMC-IMAGON extension card for HDMI video input and output, and reach good performances, processing Full HD formats, due to the level of optimization that they arise. In their work, a video stream from the HDMI input is processed and outputted directly as a stream that contains only the foreground(FG) Mask, to the HDMI output port. The CPU only control the process parameters through the AXI Lite interface [26], and do not interact with the Video Stream. The authors made a very interesting work reducing all the bottlenecks, but it is not possible to use the advantage of MoG, sending the foreground mask output video flow to the next stage (for example to perform human pedestrian tracking).

A new method for moving object tracking, which improves the traditional KLT (Kanade–Lucas–Tomasi) algorithm, is presented in [27]. The improved KLT algorithm proposed by the authors enhances the extraction feature by introducing the motion information to filter out the static corner points. Additionally, a frame processing order strategy is adopted to take advantage of FPGA parallelism and reduce the execution time. The authors' work is remarkable, and it has been compared with our proposal in Section 6.

In [28] a tracking system for amphibious spherical robots was implemented. The robots in this system use a GMM to detect a moving target entering the field of view. The authors achieved good results employing 8-bit gray scale frames, and an internal image resolution of 160 x 120. They used a OV7670 CMOS VGA-specific camera with PMOD interface with I2C protocol and a parallel specific interface to communicate with FPGA. The authors also studied the precision and discussed their implementation drawbacks.

A different approach for background subtraction is presented in [29], where the authors present an algorithm that uses the hue component of HSV color space [30] and the gray-scale pixels' intensities to implement an efficient background subtraction algorithm for real time applications. The internal hardware implementation has limited computational requirements and memory resources because it works with 160x128 and 320x240 resolutions. The authors of

this paper made a special effort to to analyze the performance metrics and the quality of the foreground mask.

A surveillance Zynq XC7Z020 FPGA board (ZedBoard) based tracking system is presented in [31]. The authors used an OV7670 dedicated camera, controlled by I2C, directly connected to the ZedBoard PMOD input bus. The resolution of the processed signal is 640 x 480, with an 8-bit grayscale. The method employed to complete the segmentation between background and foreground regions is a pixel subtraction between a stored frame (background) and an input frame.

Another relevant method is presented in [32], where a Siamese Neural Network is implemented focusing on the mobile devices. The authors reduced the 32-bit floating point and activation parameters into 8-bit keeping a good precision. They also reduced the inference time of the network using pruning and quantization techniques. The tests are done using the VOT2015 dataset [33], whose sequences are not constant in terms of resolution. Finally, the system is compared with VOT2015 dataset testing results of 62 state-of-the-art trackers, and their FPGA implementation achieves a robust energy-efficient and area-concise solution.

The solution presented in [34] introduces a moving object detection and tracking system based on dynamic background difference and Kalman filtering. Wenchao Liu et al., improved the dynamic background difference combining the time averaging method and the recursive filter for the real-time background image updating. The processing system combines the HW/SW resources to archive real-time performances.

An interesting methodology used to accelerate image processing application in an embedded system is introduced in [35], where the authors developed an approach for hardware/software codesign using platform based SystemC TLM (Transaction level modeling) able to perform Hardware synthesis using SystemC to accelerate system design and verification process. Thanks to this technique, the authors explain how is possible to model and validate a complex embedded systems reducing implementation times, achieving good system performance.

An other work that uses High level synthesis to accelerate Hyperspectral Images processing application is presented in [36]. Authors develop an embedded system using an Intel Cyclone V SoC. The algorithm is implemented using C++ and OpenCL framework. The algorithm is synthesized into hardware using the Intel FPGA OpenCL Offline Compiler. The results achieved in this work shows a reasonable speedups using a device with reduced computing and embedded memory resources. The authors affirm that the use of HLS raise the abstraction level, simplifying and speeding up the hardware generation from software algorithms, even if a

considerable amount of hardware design skills is required to implement a complete embedded system.

Finally, the last relevant study to mention is based on optical flow estimation and a Gaussian mixture model [37]. The authors proposed a novel algorithm focused on environments wherein the camera is not static; the study was implemented and tested on a Xilinx Virtex-5 FPGA. Jaechan Cho et al., compared their work with previous research, and found that they had achieved good performances in terms of speed, precision and detection accuracy thanks largely to the contribution of the optimized hardware structure and the background compensation.

The previous study is considered to be the model for the improved system presented within this paper. The Gaussian mixture model is adopted to segment the frames extracted from the video sequence and extract the regions which with a high probability belong to moving objects (humans in our case). Jaechan Cho et al., proved that this algorithm is suitable to be embedded on an FPGA device, and therefore, to ease the deployment of the system and reduce its power consumption. In our case we have improved their results by optimizing the HW architecture by including a temporal buffer, reducing the number of operations and improving the communications among the different modules. In order to complement the proposed HW implementation and test a complete solution for trajectory estimation, a Kalman Filter [34] has been included in the system as well. This filter has been widely used in related literature to propose a temporal model and predict the current location of the object based on the previous position and the current estimation during the detection step.

2.2 FPGA and Zynq Architecture

The experimental setup of this thesis has been implemented using FPGA, which stands for "Field-Programmable Gate Array". An FPGA is an integrated circuit that can be customized for a specific application. In fact, FPGAs are "field-programmable", it means that they can be programmed to desired application or functionality requirements after manufacturing. The structure shown in figure 2.1 represents the FPGA internal architecture. It is possible to distinguish three main zones:

- The *Logic Blocks* are physical array of logic gates, that can be wired in different configurations to perform various operations.
- The *Programmable Interconnect* represents a programmable matrix of wires that is able

to connect the logic blocks between them and with the I/O blocks, according to the functionality implemented.

- The *I/O blocks* are special blocks at chip periphery used for external connections of the FPGAs with other devices outside the chip.

Figure 2.1: FPGA internal structure from [1]

Thanks to their flexibility, FPGAs can be optimized for any computing task, revealing the potential to perform operations faster than a hard-wired processor, with the advantage that they can be reprogrammed as the design evolves.

This thesis uses an evolution of FPGAs, a hybrid system called Zynq that combines a CPU and an FPGA in the same chip.

The ZYNQ-7000 System on Chip (SoC) architecture, shown in figure 2.2, has been developed by Xilinx. It integrates a dual-core ARM cortex A9 microprocessors as the processing system (CPU) and the Artix-7 FPGA logic as the programmable logic. The FPGA and the CPU share the available resources, a highlights of this architecture extracted from the Xilinx online specification [2] are listed as follows:

- Dual-core ARM Cortex A9 ARM v7-A 32-bit microprocessor, with speeds up to 1GHz.
- On board peripherals of the PS, include: DMA controller, DDR Memory controller, Quad-SPI flash controller, SD/SDIO controller, GPIO, USB host/device/OTG controller, Gigabit Ethernet controller and CAN/SPI/UART/I2C controllers.

- The programmable logic is based on the Artix-7 FPGA logic with a 6-input look-up table based CLB, 36KB block RAM and DSP48E1 digital signal processing slices.
- Programmable logic is on a separate power domain for on demand power ON and shut-down, allowing for a software centric fabric configuration.
- Interconnect is based on the ARM AMBA 3.0, using AXI point-to-point channels for data and address transfers between master and slave clients.
- Three different interface types, AXI_ACP, AXI_HP and AXI_GP, exist between PS and PL depending on the bandwidth and application requirements.

Figure 2.2: Zynq 7000 architecture from [2]

A very important aspect is the bandwidth shown in the table of figure 2.3 because it needs to meet the data transfer requirements of the design.

Interface	Type	Bus Width (bits)	IF Clock (MHz)	Read Bandwidth (MB/s)	Write Bandwidth (MB/s)	R+W Bandwidth (MB/s)	Number of Interfaces	Total Bandwidth (MB/s)
General Purpose AXI	PS Slave	32	150	600	600	1,200	2	2,400
General Purpose AXI	PS Master	32	150	600	600	1,200	2	2,400
High Performance (AFI) AXI_HP	PS Slave	64	150	1,200	1,200	2,400	4	9,600
AXI_ACP	PS Slave	64	150	1,200	1,200	2,400	1	2,400
DDR	External Memory	32	1,066	4,264	4,264	4,264	1	4,264
OCM	Internal Memory	64	222	1,779	1,779	3,557	1	3,557

Figure 2.3: Zynq Theoretical Bandwidth from [3]

A special highlight is directed to the the High Performance (AXI_HP) and DDR rows that give a preview about the bandwidth or the possible bottleneck of the system.

2.3 ZedBoard

The ZedBoard is a ZYNQ-7000 based development board from Avnet Inc. that came with the ZYNQ part number XC7Z020-CLG484-1. The choice of the development board was based on the availability of the board and the support for implementing a full working Linux system able to run the algorithm and receive the camera streaming form the Ethernet port. The development board shown in figure 2.4, features the necessary interfaces for all the required components to realize the tracking System. Some of the highlights of the board from [4], which played an important role in realizing the system are following:

- Zynq-7000 SoC based on XC7Z020-CLG484-1 with enough FPGA resources for the Background Subtractor processing IP core.
- Dual-core ARM Cortex-A9 667 MHz able to run the complete algorithm.
- On board 512 MB DDR3 RAM and SD card interface to run Linux operating system and to satisfy the memory requirements.

- Communication interfaces include Ethernet port for connectivity, the USB-UART for debugging.
- ZedBoard also supports I/O expansion via an FMC connector and PMOD, supporting the interface of many devices that could be added to improve the system.

Figure 2.4: ZedBoard photo with description from [4]

The complete Board specifications are available in [38].

2.4 Vivado HLS, Vivado and XSDK

The High-level synthesis tools have increased their popularity in the recent years. They allow to designers working at a higher-level of abstraction using a software tool to translate software algorithms into hardware structures. The implementation of complex embedded systems is

simplified giving to less expert designers the opportunity to develop high-performance and energy-efficient heterogeneous systems. A complete analysis of these tools is presented in [39], where the authors introduce a complete analysis of recent HLS tools, comparing them and showing an overview about the areas of interest in the HLS research community. Also a comparison methodology developed by authors is presented, to value the HLS tools in term of performance and use of resources, executing a common set of benchmarks. This section focuses on the Xilinx software tools environment used during the development of this thesis, to interact with the ZedBoard and to synthesize the hardware for the ZYNQ XC7Z020-CLG484-1.

2.4.1 Vivado

The Vivado Integrated Design Environment (IDE) is a suite of tools made by Xilinx [40] that provides an intuitive graphical user interface (GUI) with powerful features, that allows synthesizing digital circuits and program them into the Xilinx FPGAs. This suite of tools permits to perform sophisticated operations and allow hardware designers to work at higher levels of abstraction becoming more efficient. Vivado is usefull to create hardware blocks using programming language (e.g., Verilog, VHDL and SystemVerilog) that compiles a design into a register-transfer level (RTL), and as next step, the tools can translate RTL specifications into a digital circuit model that is flashed into the FPGA. Vivado also allows to pack the hardware block into a Intellectual property (IP) to be used in a bigger design at block design level. One of the principal utility offered by this tool is the connection between more IPs and to all accessible FPGA resources, allowing to create an electronic digital device. A list of features is following:

- Register transfer level (RTL) design in VHDL, Verilog, and SystemVerilog
- Intellectual property (IP) integration for cores
- Behavioral, functional, and timing simulation with Vivado simulator
- Vivado synthesis
- Vivado implementation for place and route
- Vivado serial I/O and logic analyzer for debugging
- Vivado power analysis

- SDC-based Xilinx design constraints (XDC) for timing constraints entry
- Static timing analysis
- High-level floorplanning
- Detailed placement and routing modification
- Bitstream generation

This list taken from Vivado documentation [41] and the description that proceed it, represents a concise explanation about the wide range of options offered by Vivado, to implement digital circuits for Xilinx FPGAs, to learn more please consult the cited documentation.

2.4.2 Vivado HLS

Vivado High-Level Synthesis (HLS) is a tool developed by Xilinx [42] able to transforms an algorithm written in C, C++ or SystemC, into a register transfer level (RTL) implementation that is possible to synthesize in a specific digital circuit for Xilinx FPGAs. This represents another step in abstraction that enables hardware designer to work in C, C++, or SystemC, while creating high-performance hardware on the FPGAs. The automatic RTL conversion depends on the ability to perform several things automatically that an RTL designer does. The analysis and decisions automatically performed by HLS, are based upon user-provided input specification and design constraints. A list of task automatized by HLS is following:

- interfaces implementation and connection with to the rest of the system.
- algorithm analysis to exploits its concurrency.
- introduction of registers as necessary to avoid critical paths and obtain the desired clock frequency.
- data mapping into memory elements balancing bandwidth and resource usage.
- control logic generation to control the data path.
- transform algorithm calculation into logic elements to archive the most efficient implementation, applying user specified and automatic optimizations.

Vivado HLS, requires to the designer the introduction of a function in C, C++, or SystemC, to describe the interface, to provide an FPGA target device, to set the desired clock period and to give optimization directives. A testbench to verify the design is also required, that proves its correctness by checking the results. Is not possible to translate any arbitrary software code with HLS, in fact some common software concepts does not have a hardware correspondent implementation. The designers need to introduce additional information (`#pragmas`) providing some directives to HLS that permits to create a most efficient design. Even if Vivado HLS can compile almost any C/C++ program, it does not support all type of code instructions and syntax. A list of coding limitation extracted from [43] is following:

- No dynamic memory allocation and pointers.
- System calls are avoided in hardware to reduce complexity.
- Recursive functions are not supported.
- standard libraries (e.g., common `math.h` functions) are supported, but uncommon ones are not.
- The interface must be precisely defined including the communication standard to use.

while on other hand, it can perform advanced optimizations (pipelining, memory partitioning, bitwidth optimization) to implement an efficient hardware. Also HLS can support a variety of different interfaces such us direct memory access, streaming, on-chip memories. The output of Vivado HLS is the RTL hardware design that is packed into an Intellectual property (IP). Finally the IP could be integrated into an FPGA hardware design to be synthesized through the rest of the hardware design flow. Furthermore, it is possible to use the testbenches to aid in the verification process. Vivado HLS generates the following outputs:

- Synthesizable Verilog or VHDL.
- RTL simulations based on the design testbench.
- Static analysis of performance and resource usage.
- Metadata at the boundaries of a design, making it easier to integrate into a system.

Once the RTL-level design is ready, it is integrated in the standard RTL design flow and other tools such us the Xilinx Vivado Design Suite are used. As described in the previous paragraph,

Vivado can perform the logic synthesis, creating the FPGA logical elements translating from the RTL-level design. Finally the bitstream is generated. It contains the binary representation of each FPGA resource, including logic elements, wire connections, and on-chip memories that allow to perform the functionalities of the original C/C++ algorithm translated by Vivado HLS. This represents the end of the hardware generation process, where the bitstream can be loaded on the FPGA ready to work.

2.4.3 XSDK

The Xilinx Software Development Kit (XSDK) is the Integrated Design Environment (IDE) based on Eclipse, customized to create embedded applications for Xilinx's embedded systems based on Zynq and MicroBlaze soft-core microprocessor. The SDK allows to code the application, debug and build it, and analyze its performance. The main features are:

- Zynq UltraScale+ MPSoC, Zynq-7000 SoCs, and MicroBlaze support.
- Complete Integrated Design Environment (IDE) connected with the Vivado embedded hardware design environment.
- Support the design and debug flows, including multi-processor and hardware/software co-debug capabilities
- Editor, compilers, build tools, flash memory management, and JTAG debug integration
- Full suite of libraries and device drivers
- Xilinx Software Command Line Tool (XSCT) available for scripting

The SDK is permit to cross compile the application for Zynq and MicroBlaze ARM CPUs, but also permits to debug it thanks to the System Debugger. This is a very important feature that helps developers to verify the code directly on the embedded system.

Figure 2.5: Xilinx System Debugger workflow from [5]

All the common debug features are supported (setting breakpoints or watchpoints, stepping through program execution, viewing the program variables and stack, and viewing the contents of the memory in the system). XSDK is Based on the Open-Source Target Communication Framework (TCF) and allows to create a debug session, through a single JTAG cable in case of bare metal application or through Ethernet connection in case of Embedded Linux application. Figure 2.5 shows the debugger workflow, the application is compiled by the XSDK as elf file executable for ARM CPUs. It is launched on the platform including the Xilinx hw_server as the underlying debug engine. The developer can interact with the SDK GUI performing the Debug. The XSDK translates each user interface action into a sequence of TCF commands. The output is processed from system Debugger that displays the current state of the program on the SDK GUI, returning the debug information to the developer.

XSDK also includes user-customizable drivers for all supported Xilinx hardware IPs, and a board support package (BSP) that is a collection of libraries and drivers for the development boards (Included the ZedBoard used in this work). More information and the complete documentation is available in [44].

2.5 OpenCV

OpenCV (Open Source Computer Vision Library) [45] is an open source computer vision and machine learning software library developed by Intel, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. It is a multi-platform library written in C/C++, in fact it runs under Linux, Windows and Mac OS X. Also the development of interfaces make it available for Python, Ruby, Matlab, and other languages. The library focuses mainly on real-time image processing, and contains more than 2500 optimized algorithms, that are implemented in multithreading to improve video processing performance. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, etc. Some of the processing functionalities offered by OpenCV library are listed below:

- Read and write images
- Capture and save videos
- Image processing such as filtering and transformation
- Feature detection
- Video or image object detection such as human body parts, cars, signage, etc.
- Video analysis

OpenCV library offers many interesting features [46] allowing to be used in a multitude of applications such as: robotics, medicine, industrial automation, security, and transportation. In the robotics field, the library could be useful to determine a robot's location. Some other function where it can be used are: navigation, Obstacle avoidance, and Human-Robot Interaction. In industrial automation, OpenCV can be used to determine defects of stock, object sorting, barcodes and packages, document analysis, supporting human to perform dangerous jobs, and many more. Also OpenCV is adopted in security services, because it allows to develop applications for surveillance and biometrics, for transportation, it can be useful to detect driver vigilance and develop autonomous vehicles. In the emerging field of medical applications, OpenCV can help doctors in vision-guided robotic surgeries, or in analysis tools through classification and detection of cells or tumors, 2D/3D organ reconstruction, infrared thermal body scanning segmentation, and all medical image features.

Chapter 3

Architecture and Workflow

The complete tracking solution is analyzed in this chapter. The main idea consists into evaluate a high level C++ software implementation based on the OpenCV libraries, optimizing it block by block where necessary. This allows to explore the design step by step and provide a flexible and fast solution for the embedded platform. The algorithm is divided into blocks to perform a time analysis to individuate the more demanding module. A software testing phase is performed to examine the algorithm in terms of data structure, variable types, and memory occupation. The C++ code analysis is also necessary to recognize which part need to by modified and optimized. A complete set of optimization to implement in the algorithm is described, and the steps to introduce the algorithm into Vivado HLS is finally detailed.

3.1 Design Methodology

The algorithm analysis, optimization and porting from the software implementations to its final application into the FPGA is described in this section. The various steps are represented in figure 3.1, that covers from the design introduction in C++ up to the Vivado HLS steps to achieve the hardware structure to flash into the FPGA.

The *C++ code Design* is the first stage, where the code is analyzed and modified to apply all the changes necessities before its introduction into the HLS translation tool. These changes that are implemented from the programming side using a normal software IDE (as for example

Eclipse, Visual Studio, etc). They involve the data structure, variable types, loops and C++ software instructions, to make the code compatible with Vivado HLS.

Figure 3.1: From Software library to Hardware IP workflow.

The *C++ Functional Verification* represents the step where the code is imported into Vivado HLS. In this step the functionality of the code is proven adding a test-bench that is also used to verify the code and all the optimizations that are introduced to reduce the algorithm time.

The *High Level Synthesis* is the part of the chain where the code is translated into RTL (VHDL, VERILOG).

The *RTL Verification* allows to prove that the translated code is maintaining its functionality, analyzing the results of the test-bench through the comparison with the results obtained in the previous stages.

The *IP RTL Export* consists into pack the synthesis output performed in the preceding steps, into a component library.

The *FPGA Hardware Integration* represents the step where the IP is connected with the rest of the system into the FPGA.

The *Bitstream Generation* is the end of the chain that creates the final output that is suitable to be executed on the FPGA.

In the diagram of figure 3.1 there are two iterations between the first four steps. It means that these steps are repeated until the solution achieve the requirements needed. Once the algorithm specification, modification and optimization are detailed in the following paragraphs, a more accurate description about the software porting to hardware using Vivado HLS, is provided at the end of this chapter in section 3.5.

3.2 Proposed Background Sub-tractor Algorithm

The algorithm chosen for HW acceleration is MoG because of its stability for background subtraction, and a Kalman filter for the tracking because of its robustness based on a statistical model. This section is focused on the initial step (MoG) embedded on an FPGA device whose purpose is to segment between foreground (FG) and background (BG) regions in the scene. For that purpose static elements inherent to the scenarios should be modeled through a statistical distribution. That defined distribution will allow the determination of the moving objects present in the scene, considered as Regions of Interest (ROIs), for an algorithm whose aim is to define pedestrian trajectories. A new person could be brought into the scene or a present object removed from it. In order to adapt to changes, the solution proposed in [47] has been initially adopted. The training set is updated by adding new samples and discarding the old ones. At time t :

$$X^T = x^t, \dots, x^{t-T} \quad (3.1)$$

For each new sample, the training data set X^T is updated and $\hat{p}(x|X^T, BG)$ is re-estimated, the probability of every pixel ($\hat{x}$) (with the prior knowledge of the history) is modeled by a mixture of K Gaussian distributions (X^T) defining the inherent elements of the scene (BG). However, among the samples from the recent history there could be some values that belong to the Foreground (FG) objects, and this estimate is denoted by $\hat{p}(\bar{x}|X^T, BG + FG)$. MoG is employed with $M = 3$ components

$$\hat{p}(\bar{x}|X^T, BG + FG) = \sum_{m=1}^M \hat{\pi}_m N(\bar{x}; \mu^m, \hat{\sigma}_m^2 I) \quad (3.2)$$

Where $\hat{\pi}_1, \dots, \hat{\pi}_M$ are the estimates of the means and $\hat{\sigma}_1, \dots, \hat{\sigma}_M$ are the estimates of the variances that describe the Gaussian components. The covariance matrices are assumed to be diagonal and the identity matrix I has proper dimensions. The mixing weights denoted by $\hat{\pi}_m$ are non-negative and add up to one. Efficiency of this initial proposal was improved in [48] by density estimation. For that aim a uniform kernel is used to count the number of samples k from the data set X^T that lie within the volume V of the kernel. The volume V is a hypersphere with diameter D . The density estimate is given by:

$$\hat{p}(\bar{x}|X^T, BG + FG) = \frac{1}{TV} \sum_{m=t-T}^t b^{(m)} K\left(\frac{\|\bar{x}^m - \bar{x}\|}{D}\right) \quad (3.3)$$

Where the kernel function $K(u) = 1$ if $u < 1/2$ and 0 otherwise. The volume V of the kernel is proportional to D_d where d is the dimensionality of the data. In practice T is large and keeping all the samples in X^T would require excessive memory and calculating the previous equation would be too slow. It is reasonable to choose a fixed number of samples $K \ll T$ and randomly select a sample from each sub-interval T/K . This might give too sparse sampling of the interval T . The X^T also contains samples from the foreground. Therefore, for automatic learning a set of the corresponding indicators $b^{(1)}, \dots, b^{(T)}$ is kept. The indicator $b^{(m)}$ has a value 0 if the sample is assigned to the foreground. The background model considers only the samples with $b(m) = 1$ that were classified as belonging to the background. Therefore if $\hat{p}(\bar{x}|X^T, BG + FG) > c_{thr}$ the pixel is classified as background.

3.3 Model Architecture Implementation

This section analyzes the complete processing chain and includes a discussion on how to implement it on the FPGA. The original implementation of the algorithm is done using the OpenCV libraries and C++ programming language. In the Figure 3.2 the processing chain is presented, divided into 4 sub-modules. The input is a video stream processed frame by frame. The first module checks the video stream to evaluate its correctness (resolution, video components, etc.). Next, a background subtraction is performed. In the following stage, a contour finder and blob extraction algorithm prepare the data for a Kalman tracker that calculates the trajectories of the patients. Finally, the output is generated as a JSON format stream, ready for its transmission over the network. The module is never recorded in compliance with legal requirements.

Figure 3.2: Complete algorithm solution implemented in Software.

We have first developed a SW version of the system written in C++ language and using OpenCV 3.1 libraries. The background subtractor block is implemented using MoG [49], with both gray-scale and RGB inputs, considering different resolutions. The tracking stage that follows the MoG is performed by a Kalman Appearance Tracker [50]. The target's motion is modeled by a 2-D translation at a single scale and searched in an area around the previous target position. In the subsequent frame, the candidate windows around the predicted target position are reduced and compared to the template. To evaluate the execution times of the entire processing chain, an experiment has been performed. It consists into run the algorithm to process some sequences of different duration and resolution using an i7 processor and the ZedBoard CPU. Even if did not initially meet the requirements, the experiment represents our reference and the starting point. Consequently, we have evaluated the computational load of each block of the algorithm. Table 3.1 shows the time in second that are needed to process each sequence.

Sequence	Resolution	Duration (s)	Average Num of People/Frame	i7 BS (s)	i7 KT (s)	ZedB BS (s)	ZedB KT (s)
shop1.mp4	640x480	19	3.5	6.62	0.02	107.28	0.33
shop2.mp4	640x480	19	8	6.78	0.04	112.37	0.83
hosp1.mp4	1056x1056	38	4.3	28.19	0.03	485.37	0.39
hosp2.mp4	1056x1056	11	3	8.56	0.009	146.82	0.14
hosp3.mp4	1056x1056	23	3.3	17.07	0.015	292.46	0.22
hosp4.mp4	1056x1056	40	2.3	29.7	0.03	524.87	0.45
hosp5.mp4	1056x1056	45	5	32.79	0.03	556.54	0.38
lab1.mp4	854x480	6	2	3.16	0.007	52.57	0.12
lab2.mp4	1920x1080	6	3	15.97	0.01	271.97	0.14

Table 3.1: Software time measurements in seconds, processing a video test sequence using only the CPU. BS stands for background subtraction, KT stands for Kalman Tracking, i7 represents the personal computer, ZedB stands for ZedBoard.

Is it possible to note that the background subtractor is the more demanding in term of performance, for this reason, the decision to accelerate this block and implement it in hardware. This means that the FPGA hardware implementation replaces the OpenCV software implementation to reduce the processing time, as indicated in Figure 3.3. It is possible to distinguish two paths: the red represents the full-software process, the slower one, while the green path represents the accelerated version implemented in the FPGA.

Figure 3.3: Software processing workflow to be modified and implemented into the FPGA.

In the figure 3.3 the variation of the processing chain is presented. The video with the patients at the monitoring scene is not recorded attending legal issues, furthermore the Debug output of the Kalman tracker is not available outside the board. The Debug Output has been added to further evaluate the chain than includes the FPGA (the green path). When enabled, it saves the video data as FGmask images streaming and the tracking data in a file. The debug data analysis is done copying the debug data in a computer and processing it with a tool made by us. This method grants the patient privacy because the recordings are not available, and when the debug output is enabled, it generates the data that is analyzed outside the board with the tool. Without the right interpretation, the data has no sense, is not possible to see the patients faces or other pattern useful to recognize them.

3.4 Algorithm customization for acceleration

To accelerate the algorithm the Vivado HLS workflow is chosen. This depends on the need to generate a Hardware implementation maintaining the global model flow. A pure Hardware implementation using VHDL or Verilog HDL could achieve a highest level of optimization, but in this case the use of the original C++ algorithm flow and the interaction with OpenCV libraries is mandatory to maintain the functionality of the model and grant its verification. Even if Vivado HLS allows to accelerate the C++ algorithm, its code needs to be modified. This is because some C++ features are not supported, but also the design must be adapted to the meet the FPGA internal resource utilization. Modifying the C++ code and creating a test bench that simulate the algorithm conditions before and after the Background Subtraction, allows to Vivado HLS the correct translation in HDL language and proper hardware generation. The modifications are detailed in the following part of this paragraph. The first change to apply to the C++ code is to remove the dynamic data structures. An FPGA has internal statics memories called Block RAM (BRAM), for that reason dynamic memory allocations are not viable. The original algorithm has dynamic memory allocation with C++ pointers and OpenCV Mat type, which is the standard type used to store images. The data structure employed by MoG Background Subtractor in OpenCV library is composed by an input RGB image, an output grayscale image and another dynamic image that contains the algorithm data calculations with the MoG operators (mean, weight, variance and used modes). The structure changes applied are the following:

- from input image dynamic Mat to input image static matrix;

- from output image dynamic Mat to output image static matrix;
- from MoG dynamic operators to static vector of struct that contain the MoG operators;

Figure 3.4: From dynamic Mat type to C++ vector of struct.

Another very important parameter is memory usage, because the FPGA has limited memory recourse (4.9 Mbits in the case of SoC XC7Z020). This means that data type is very important because according to the type of variable used by the algorithm, the needs for memory storage are different. A data variable analysis is done to determine the data type, and the range of values that are used by the algorithm. The main idea is to reduce memory utilization maintaining a correct representation of values. For that reason an empirical deep analysis has been done. The method used is to save the values of variables into a log file, while the algorithm is processing some test sequences. A parsing of the log files is done to get the range (maximum and minimum) of values that could assume a variable. This experiment is done monitoring all the variables that are used in the algorithm.

Variable	Type	Size (bit)	Range Min	Range Max	Value Example
Image In	unsigned char	8	0	255	127
Image Out	unsigned char	8	0	255	127
modeUsed[]	int	32	0	2	1
Variance[]	float	32	1.1961722047999501e-03	75	1.2135922443121672e-03
Weight[]	float	32	-1.6666667070239782e-03	1.4984889030456543e+01	-9.6153846243396401e-04
Mean[]	float	32	0	255	4.0640697479248047e+01

Table 3.2: Principal algorithm variable data types and values

Table 3.2 shows the most significant values as example. It is important to understand that truncating the variable could generate a lost of data and precision. For that reason a good compromise between memory occupation and precision must be achieved, trying to reduce memory usage loosing less precision as possible. A software battery of tests has been performed to find the compromise. In this stage, as the tests are only software, the analysis of data has been performed visualizing the video sequences and checking the errors of the algorithm according to the values assumed by the variables. This is a fast check in order to evaluate the memory reduction effect of the variables, on the software implementation. After porting the algorithm in the FPGA, some changes in the number representation could happen again, due to truncating differences between HArduware and Software. A more accurate analysis of the complete processing chain is performed and described in chapter 6.

Variable	Original Type	Size (in bit)	FPGA Type	Size (in bit)
Image In	unsigned char	8	unsigned char	8
Image Out	unsigned char	8	unsigned char	8
modeUsed[]	int	32	unsigned char	8
Variance[]	float	32	fixed	16
Weight[]	float	32	fixed	16
Mean[]	float	32	fixed	16

Table 3.3: Variable data type change resumed

Table 3.3 shows the changes applied to the data. While the Image In and Image Out maintain the same format, the change that allow to improve the internal memory use is applied to the *modeUsed*, *Variance*, *Weight* and *Mean* vectors, especially the last three vectors store the highest amount of data. The change from float to fixed allows to reduce the three vectors to half size, but it could introduce precision errors due to the truncated and rounded representation of the values. This will be evaluated in the chapter 6.

Table 3.4 shows an example of data memory occupation. This allows to quantify the amount of memory that the algorithm needs to operate. Also is important to remark that the FPGA internal memory is 4.9 Mbits (612.5 Kbyte), this means that the complete data to process a frame cannot fit into the internal memory. As a consequence, a strategy to copy and process all the data is needed. This will be described in chapter 4.

Variable	320x240	320x240	640x480	640x480	1056x1056	1056x1056
	original size	reduced size	original size	reduced size	original size	reduced size
Image In	0.230	0.230	0.921	0.921	3.345	3.345
Image Out	0.076	0.076	0.307	0.307	1.115	1.115
modeUsed[]	0.307	0.076	1.228	0.307	4.460	1.115
Variance[]	0.921	32	3.686	1.843	13.381	6.690
Weight[]	0.921	0.460	3.686	1.843	13.381	6.690
Mean[]	2.764	1,382	11.059	5.529	40.144	20.072

Table 3.4: Variable data usage comparison example according to image resolution. All the sizes are expressed in MB.

After describing the memory data occupation, another important change that is applied to the algorithm, is represented by an unsupported C++ feature used in OpenCV MoG library, to increase the algorithm performance: the *parallel_for_* loop [51], a framework useful to parallelize the code to be suitable for running taking advantage of CPU multithreading. In fact this framework is able to divide and assign the loop iterations to the different CPU threads, improving the algorithm performances. Unfortunately, this is not possible to implement on an FPGA because has a different structure, and neither Vivado HLS support this framework. For that reason the MoG algorithm from OpenCV library code is modified using the normal *for* and a series of adjustments to implement the algorithm in a suitable way to be supported by Vivado HLS.

An important optimization applied to the algorithm is replacing division operations with multiplications. The factor under analysis is the learning rate of the algorithm α^T that is calculated inside the FPGA with the following division

$$k = \alpha^T / \text{weight} \quad (3.4)$$

After a study about the evolution and convergence of this factor, it has the same magnitude order of α^T and can be approximated in the following way:

$$k = \alpha^T + \text{weight} * m_k \quad (3.5)$$

$$m_k = \begin{cases} 0.000288, & \text{if } \alpha^T < 0.025 \\ 0.003221, & \text{if } \alpha^T > 0.025 \end{cases}$$

Where m_k is a constant founded empirically observing α^T and *weight* evolution. The algorithm equations 3.4 and 3.5 have a similar convergence that depend more on α_T because *weight* assumes values between 0 and 1, tending to one as long as the iterations of the algorithm are increased.

The MoG has also a shadows detection, that internally contains more loops and divisions. In this work, to improve the algorithm speed, this feature has been disabled.

The tracking stage that follows the MoG is performed by a Kalman Appearance Tracker [50]. The target motion is modeled by a 2-D translation at a single scale and searched in an area around the previous target position. In the subsequent frame, candidate windows around the predicted target position are reduced and compared to the template.

The next optimization that helps Vivado HLS into create a better translation is to make copies of variables and split the formulas performing only sums or multiplications. For example split this expression

$$\text{dist2} = \text{dData}[0] * \text{dData}[0] + \text{dData}[1] * \text{dData}[1] + \text{dData}[2] * \text{dData}[2];$$

into these

```
dData0a = dData0b = dData[0];
dData1a = dData1b = dData[1];
dData2a = dData2b = dData[2];
mul0 = dData0a * dData0b;
mul1 = dData1a * dData1b;
mul2 = dData2a * dData2b;
dist2 = mul0 + mul1 + mul2;
```

This change, even if looks like a larger code, allows to improve the parallelization, because it creates copies of the original data avoiding collisions with other processes that need to read or write the same data. These copies made separate multiplications and are stored in different variables(registers), that are summed to obtain the final result. The modification of the different formulas allows Vivado HLS to give the right priority to the creation of the final result, generating an optimized hardware structure that multiplies and sums the data. It avoids data collision and the use of additional register to store the variables copies is minimal and do not represent a waste of space in the memory of FPGA. This improves the parallelization process because it allows to read, write and process more data at the same time, since the original data is read and written in the new result registers, that can achieve

the results in the next clock cycles in an independent way, while the original input resources are available for other purposes or could be deleted.

The last optimizations applied to the algorithm are the Vivado HLS pragma, a set of directives that allow to improve loops pipeline, memory resources distributions, reduce latency, improve throughput performance, etc. For more information is possible to consult the Xilinx guide [43].

3.5 Vivado HLS Design Porting

This section describes the design workflow followed from the modification of the algorithm to its working implementation using the Xilinx tools. The algorithm have been modified as described in the previous sections 3.2 and 3.3, and is ready to be optimized using Vivado HLS, where the code is introduced and a test-bench is created to check the C++ functionality. This step permits to understand if the algorithm is responding as expected after the modifications. The next stage is the synthesis where the algorithm is translated into RLT (Verilog or VHDL). At this point, all the Vivado HLS specific optimizations and directives are applied to define which type of SoC or SoM is adopted, the Hardware resources to use, the communication buses and protocols, etc. If the C++ code could be synthesized by the tool, it means that all types of configuration needed have been successfully applied and the result is ready for the next steps.

After the synthesis has been completed, the RTL Verification follows to check if the design is meeting the time requirements and if it is performing as expected. These three steps (C++ Functional verification, High Level Synthesis and RTL verification) could be iterated if the design needs improvements or do not satisfy the requirements. In this case the directives could be changed, some optimizations could be added to reduce latency or improve throughput. The experiments are iterated until an optimized combinations in terms of timing and resources is found and the design meets the requirements.

Once the RTL Design is verified and meets the time requirements, it is packed and exported as IP, a package that can be included in every Vivado Hardware design. The next step is the integration in Vivado, another tool by Xilinx that is close to the Hardware because it is able to create the final binary that will be flashed into the FPGA. In Vivado, the IP is integrated using specific internal blocks to communicate with the ARM CPU, is connected with the RAM to read and write on the memory, and with all component necessities to process the

video stream. Finally, when the FPGA internal architecture is complete (more details will be explained in next paragraph) and the design pass the DRC (Design Rule Check), is possible to proceed with Vivado to create the bitstream, the final file that contains the all the hardware specifications that are flashed into the FPGA. At this time, the accelerator is ready to work on the FPGA, the operating system will be configured to recognize it as a hardware peripheral. The final stage that follows is the integration with the rest of the algorithm. In order to take advantage of the speed improvement that the FPGA brings, the embedded system (CPU, FPGA, OS and software) needs to be optimized using all potential strategies to improve data exchange and reduce possible latencies.

Chapter 4

Hardware Acceleration

This chapter describes the strategies and improvements applied to accelerate the design. It starts describing the some technical requirements and follows describing all the modification that are necessities to reach real time.

The ICT4Life consortium European research project has stated that 15 FPS is the lower limit to track Parkinson's-affected patients. As the detection and tracking algorithm is being evolved, we require a flexible solution for modeling and prototyping (mainly in SW) with the potential for posterior editing for optimization purposes, in case the requirements are not satisfied. It is important to follow a design methodology described in the previous chapter, that enables the functional and model evaluation of the algorithm via software simulation, while enabling module modifications. Last, we select an embedded system, so it can be deployed in the target location, or even integrated in the camera.

Considering these requirements we propose processing the tracking information close to the camera, using a hybrid System-on-Chip based solution, with CPU and FPGA. The SoC board is connected to the Zenith camera through IP/TCP. Based on the latest advances, with special mention to the work of Bulat et al., [23] which compared three tracking algorithms focusing on the advantages of the use of FPGAs over GPU and CPU, the FPGA-SoC is a suitable choice for the base architecture to run and to accelerate the algorithm, and to meet the required timings and also respecting the privacy needs of Parkinson's patients by gathering only the tracking information, without storing images or video data.

For prototyping, we selected the Zedboard, with the Xilinx Z-7020, available when the project started and described in the paragraph 2.3. The board includes a Linux OS, in order to accelerate the integration of the module in the system, as it provides mechanisms to add libraries and modules to handle IP/TCP, Web Server, multithreading, memory handling, etc. The tracking module was developed in C++ using OpenCV libraries. When required, Xilinx Vivado HLS offers the possibility of designing and modelling in C++, providing, as output, a synthesized submodule for the SoC's FPGA. The C++ sub-modules can be both simulated and synthesized, considering some restrictions in code structure and implementation that are detailed in Section 3.3. A pure HW implementation using VHDL or Verilog HDL could achieve a higher level of optimization, but in this case the use of the original C++ algorithm flow and the interaction with OpenCV libraries is essential to maintain the functionality of the model, grant its verification and allow future modifications and improvements of the system.

As Starting point the algorithm introduced in the paragraph 3.2 has been modified as described in paragraph 3.3. The type or variable has been replaced and truncated, the loops has been adapted and the code is ready to be synthesized with Vivado HLS. The tool give a more complete view about the algorithm synthesis, but its analysis focuses on some parameters only to highlight the acceleration process that is possible to reach optimizing the hardware. In fact the parameters analyzed are in terms of resourced used and timing. The *Resources Usage* indicator highlight the internal usage of the FPGA, according to its memories, registers and various processing units used to implement the design. In this case the indicator permits to determine if the algorithm is able to store the image and data needs by its variable in the FPGA memories, to perform the Background Subtractor calculations. Taking into account a 480x480 RGB frame to make an example, according to the memory needs discussed in the paragraph 3.3, only a part of the image and vectors having size 480x32, could be stored into the FPGA internal memory. This means that to process an entire 480x480 frame, it needs to be divided it into 15 parts, and to be processed part by part.

A brief description about the *Performance Estimates* indicators is following:

- *Timing* - this indicator represents the target clock frequency, clock uncertainty, and the estimate of the fastest achievable clock frequency. In this case the ZedBoard has a 100 MHz clock that is used as the FPGA Master clock, for that reason the target period is 10ns.
- *Latency* - this measure indicate the number of cycles needed to produce the output. The initiation interval represents the number of clock cycles to process current data, before new inputs can be applied.

- *Final Timing* - this table evaluate the performances after the synthesis process is completed. It shows the Clock Period required and the Clock Period achieved post synthesis and post implementation. The tool in this table shows if the design meet the timing or not.

Performance Estimates

- Timing (ns)

- Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.40	1.25

- Latency (clock cycles)

- Summary

Latency		Interval		Type
min	max	min	max	
?	?	?	?	none

Resource Usage

	Verilog
SLICE	2656
LUT	6388
FF	6776
DSP	18
BRAM	180
SRL	146

Final Timing

	Verilog
CP required	10.000
CP achieved post-synthesis	8.919
CP achieved post-implementation	10.668
Timing not met	

Figure 4.1: Vivado HLS Algorithm starting point report.

Performance Estimates

- Timing (ns)

- Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.40	1.25

- Latency (clock cycles)

- Summary

Latency		Interval		Type
min	max	min	max	
576195	2281155	576195	2281155	none

Resource Usage

	Verilog
SLICE	2656
LUT	6388
FF	6776
DSP	18
BRAM	180
SRL	146

Final Timing

	Verilog
CP required	10.000
CP achieved post-synthesis	8.919
CP achieved post-implementation	10.668
Timing not met	

Figure 4.2: Vivado HLS Algorithm first valid estimation.

Figure 4.1 represents a report about the first algorithm implementation. As is possible to note, Vivado HLS is not able to estimate the algorithm processing latency. The final result is that timing is not met. Once the algorithm modification process starts, the tool is able to quantify the latency as is possible to see in figure 4.2. In fact the algorithm is able to complete its execution in an interval that varies between 576195 and 2281155 clock cycles, in terms of time it corresponds to 5.76ms and 22.81ms. This result means that the algorithm, in the worst case, is able to process one 480x480 frame in around 0.34 seconds (2.9 FPS). Unfortunately it is not capable to process a stream of 15 FPS.

Figure 4.3: Vivado HLS Algorithm Performance estimation.

As expected, introducing the algorithm in Vivado HLS and adapting it to make it work is not enough. In fact the design do not met the requirements, it needs to be improved reducing the execution time. Chapter 3 describes a set of optimizations very useful, that has been applied to the design according the workflow shown in figure 3.1, iterating until the performances meet the requirements. The optimization result is shown in figure 4.3, as is possible to note, the algorithm has improved its execution. The optimized interval varies between 138390 and 153764 clock cycles, in time it means 1.38ms and 1.53ms. This result not only meet the time as noticed by Vivado HLS, but also means that the algorithm running on the FPGA, is able to process one 480x480 frame in around 22.95ms (43.57 FPS). The optimized version of the BGS algorithm is capable to process a stream of 15 FPS. Even if this result take in account the input/output times, is important to remark that the final result is the represented by the complete processing chain introduced in figure 3.3 of previous chapter.

All the resources needed by the complete tracking algorithm, that include CPU and FPGA and the communication between them, must be optimized to reach real time. More details will be described in the following paragraphs and chapters.

4.1 Hardware Model

The Background Subtractor algorithm has been optimized as described in the previous paragraph, the output of Vivado HLS is an intellectual properties (IP) that is ready to be integrated with the rest of the system. The tool that is used for the integration is Vivado, as described in the paragraph 2.4, that allow to connect the IP with other blocks to provide the its communication with the CPU. An important role is played by sending data between the CPU and the FPGA. Is important to choose the right method to have fast and capable channels, and to avoid bottleneck. According to the documentation [26] good performances could be archived by using the AXI interface. There are three types of AXI buses:

- AXI master is employed for memory-mapped interfaces and allows high throughput bursts of up to 256 data transfer cycles with just a single address phase. This will be used for the vectors data interchange.
- AXI lite is a light-weight, single transaction memory-mapped interface, suitable for control signal.
- AXI stream that allows unlimited data burst size suitable for video streaming flows. This will be used for the video data interchange.

More details from the functional point of view will be described in the next paragraph. Once the communication interface has been identified, the right port with high capability must be chosen. An estimation about the possible bit rate, also confirmed by this work [52] measurements, to set a requirement for the hardware bandwidth is possible to calculate as follows:

$$\begin{aligned}
 1056 * 1056 * 3 &= 3.34MB && \text{Biggest frame occupation provided by the camera} \\
 3.34MB * 15FPS &= 50MB/s && \text{Memory needed by 15 FPS stream}
 \end{aligned}$$

Analyzing the Zynq-7000 technical reference manual [3] the solution that is suitable for this case is provided by the *AXI_HP* ports shown in figure 4.4 and highlighted in figure 4.5.

Figure 4.4: Complete connectivity diagram from [3]

Figure 4.5: High Performance (AXI_HP) Connectivity from [3]

In fact the *AXI_HP* ports allow to the FPGA to act as master and are connected directly to the DDR Memory controller. Each interface includes two FIFO buffers for read and write traffic as depicted in figure 4.5. Also is possible to select the channel width in 64 or 32 bit. Furthermore this ports have a high bandwidth, according to the table introduced in figure 2.3 of the paragraph 2.2, the theoretical throughput is around 1200 MB/s respectively for the read and the write operations. An other connection to choose is represented by the control signal. A suitable option is represented by the AXI general purpose interfaces *AXI_GP*. In fact this ports are connected directly to the *Central Interconnect* as shown in figure 4.4, can work as Master and Slave, offering a capability of 8 reads and 8 writes channels of a 32 bits width. The *AXI_GP* port has a bandwidth of 600 MB/s, as indicated in the table of figure 2.3 of the paragraph 2.2, that fully satisfies the control signal needs. According to the information provided, the choice is to connect the AXI master and Axi streaming interfaces to the *AXI_HP* ports, while the AXI lite to the *AXI_GP* port. As the interface analysis is complete and the bandwidth requirement satisfied, is possible to proceed creating the project in Vivado.

Figure 4.6: Vivado Hardware internal structure.

Figure 4.6 shows a capture from Vivado IDE using the IP integrator [53], a feature that allows to create complex system designs by instantiating and interconnecting IP from the Xilinx IP catalog with the Background Subtractor IP created with Vivado HLS, described in this thesis. The schematic of figure 4.14 introduce all the necessary blocks and its relative connections

to produce a working circuit, that will be synthesized and flashed into the FPGA. The block *apply_fpga_0* represents the IP which contains the Background Subtractor algorithm. The schematic contains other blocks that are grouped by type and explained as follows:

- The *AXI interconnect* block is a core that develops connection functions. Specially, this block allows to connect one or more AXI master devices to one or more AXI slave devices. The documentation is accessible from [54]
- The *AXI Video Direct Memory Access (VDMA)* is an IP soft core designed for Video Streaming purposes. It provides high-bandwidth direct memory access between memory and AXI4-Stream video type target peripherals. More details are available in [6].
- The *Concat* Block is an IP that permits to join various bus signals of different widths at its input, into a bigger bus that contain all signals at its output. For more information please consult [55].
- The *Processor System Reset* module is a block used to control customized reset signals for all the blocks of the design, including the processor, the interconnect and peripherals. The documentation is accessible from [56]
- The *ZYNQ7 Processing system* is the IP core that permits to interface the CPU with the FPGA. The core manage all their logic connection, that are grouped in three main categories: the extended multiplexed I/O (EMIO), programmable logic I/O, and the AXI I/O groups. More details about the core are available in [6].

Analyzing the schematic is possible to individuate the various connections of Background Subtractor IP into the FPGA that is interfacing whit the system.

The port *src_in* and *dst_out* ports of Background Subtractor IP, are connected with the VDMA ports *M_AXIS_MM2S* and *S_AXIS_S2MM*. The VDMA communicate through the *axi_interconnect_1* with the port *S_AXI_HP0* of the *ZYNQ7 Processing system*. This data path represents the Video input output of the Background Subtractor module, that is managed by the VDMA. It is connected directly to the RAM through the port HP0, and uses the AXI stream bus.

The port *m_axi_CRTL_BUS* of Background Subtractor IP, is connected with the port *S00_AXI* of the *axi_interconnect_2* module that establish the communication through the port *S_AXI_HP1* with the *ZYNQ7 Processing system*. This data path represents the vectors input output of the Background Subtractor module, it uses the AXI master bus and it is connected directly to the RAM through the port HP1.

The port *s_axi_CRTL_BUS* of Background Subtractor IP, is connected with the port *M01_AXI* of the *axi_interconnect_0* module, where is also connected the *VDMA* through the port *M00_AXI*. The *axi_interconnect_0* module group the two buses and route them to the port *M_AXI_GP0* of the *ZYNQ7 Processing system*. This data path represents the main control signal bus, that permit to the CPU managing the FPGA through software commands. It uses the AXI lite bus and it is available over the GP0 port. The remaining signals of the schematic are the clock and reset signals that are used for synchronizations purposes.

Once the connections has been specified, an in-depth description of the AXI VDMA clarify its operating mode. As already mentioned, this core for video data, provide a high-bandwidth direct memory access between the target AXI4-Stream video device and the memory. The core has four different AXI4 interfaces of which the following three are used for the implementation:

- AXI4-Lite Slave interface is used to access the control and status registers of the core and hence controlling the operation of the VDMA functionality.
- AXI4 Read Master interface fetches the data from the memory and transfers the buffered data to the AXI MM2S (Memory Map to Stream) Stream Master which connects to the target Background Subtractor core, consuming the data. The left part of figure 4.7 shows the abstracted view of the read interface.
- AXI4 Write Master transfers the buffered data from the AXI S2MM (Stream to Memory Map) Stream Slave, connected to the output stream of the Background Subtractor, to the memory. The right part of figure 4.7 shows the abstracted view of the write interface.

Figure 4.7: VDMA interface Operation from [6]: Left AXI4 Memory Map to AXI4-Stream Read; Right AXI4-Stream to AXI4 Memory Map Write;

As shown in figure 4.7, the VDMA core uses the AXI data mover and a line buffers for high throughput data transfer [6]. The DMA operations for the read channel and the write can be controlled independently using the control registers (MM2S and S2MM register of figure 4.7). Frame properties such as the frame width, frame height and stride also need to be written into their respective registers for the VDMA to arrange the continuous data from the memory into the two dimensional frame data. Monitoring of the read and write operations of the channels is possible by reading the DMA status registers of the respective channel. The read and write channels are configured to operate in free running mode to reduce the complexity of the control functionality. Buffers are enabled for both read and write channels to support up to three buffers with circular buffering mode to buffer the data in cyclic fashion.

Figure 4.8: VDMA Configuration parameters screenshot

Channel	Clock Frequency (in MHz)	Frame Size (In Bytes)	Maximum Total Data Throughput (MBytes/sec)	Percent of Theoretical
MM2S	80	1 MB	287	89.68
	150	1 MB	539	89.83
S2MM	80	1 MB	290	90.6
	150	1 MB	542	90.3

Figure 4.9: VDMA Performance from [6]

Figure 4.8 shows a composition of the VDMA configuration in Vivado. These parameters represents the final configuration after hard testing and setting up over this project structure. In different configurations, some video artifacts appear and the video sync fails. (More details in the testing Chapter). Figure 4.9 shows the VDMA video performance for each channel. Even the slowest clock configurations, satisfy the streaming requirements that needs a video flow of 50MB/s.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
apply_fpga_0	s_axi_CRTL_BUS	Reg	0x43C0_0000	64K	0x43C0_FFFF
axi_vdma_0	S_AXI_LITE	Reg	0x4300_0000	64K	0x4300_FFFF
axi_vdma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOW0CM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOW0CM	0x0000_0000	512M	0x1FFF_FFFF
apply_fpga_0					
Data_m_axi_CRTL_BUS (32 address bits : 4G)					
processing_system7_0	S_AXI_HP1	HP1_DDR_LOW0CM	0x0000_0000	512M	0x1FFF_FFFF

Figure 4.10: Vivado address configuration

Figure 4.10 shows the address configuration of each interface. In fact these addresses are the used from the driver to establish the communication between the software algorithm and the hardware Background Subtractor synthesized on the FPGA. The operating system needs to be configured to recognize the FPGA as a peripheral at the addresses indicated. More details will be described in the drivers paragraphs. For a complete evaluation of the Background Subtractor, two design versions has been generated: RGB and Gray-scale. The main difference focuses on the input image components that are three for the RGB version while it is only one for the Gray-scale version. In hardware, the schematic presents the same blocks and connections, for this reason its comparison is not detailed. The main difference is quantifiable in term of memory utilization. Figure 4.11 shows the Vivado utilization report, where is possible to note that the RGB version use more Block Ram, LUTRAM and LUT. This is a direct consequence of the image component number, in fact the RGB version needs more memory resources to save the input image pixels and the vectors data employed by the

algorithm calculation. The output image do not introduce differences because maintains the same dimension in both versions.

Figure 4.11: FPGA Utilization: Left RGB version of the design; Right Gray-scale version of the design;

Figure 4.12: Vivado timing analysis: Top RGB version of the design; Bottom Gray-scale version of the design;

Finally two Vivado features relevant for this work are the timing and power analysis shown in figures 4.12 and 4.13. According to the Vivado guide [57], the main parameter to considered are:

- Setup Area that indicates the Max Delay Analysis.
- Hold Area that indicates the Min Delay Analysis.
- Pulse Width Area that indicates the Pin Switching Limit.

Futhermore 0 ns means that all timing constraints are met, a negative value indicate that there are some violations, a positive value indicate the delay. Analyzing the timing constraints, each implementations meet them, only in the worst cases a non-significant delay could be introduced by the electronic circuits.

Summary (RGB Design Version)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 2.297 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 51.5°C
 Thermal Margin: 33.5°C (2.7 W)
 Effective θ_{JA} : 11.5°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Medium

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power

Summary (Grayscale Design Version)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 2.025 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 48.3°C
 Thermal Margin: 36.7°C (3.0 W)
 Effective θ_{JA} : 11.5°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Medium

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power

Figure 4.13: Vivado power analysis: Top RGB version of the design; Bottom Gray-scale version of the design;

An important consideration observing the timing is that there are minimum differences between the RGG and Gray-scale implementation. This means that for the FPGA, due to pipeline optimization, processing one component of a Grayscale image is similar to process three components of a RGB image. More detail will be highlighted in the Results chapter.

Figure 4.13 depicts the power analysis. As is possible to note, the RGB version needs a little bit more power than Gray-scale. This introduce a direct corresponds beetween number of memory element and power needs. In fact the greatest parameter difference is observable in the BRAM parameter. All other parameter grows as a consequence of reaching more memory elements. Also the temperature manifest a little increment in the RGB version. Even if in this analysis the power differences are highlighted, the consumption of this technology is still low power. This means that it could be adopted also in battery powered systems (such as: IoT applications, UAV, Biomedical sensors, etc).

4.2 Functional Model

After the software and hardware description, a global explanation of the system functionality is provided in this paragraph. Vivado HLS is used to generate an intellectual property (IP) HW block. Vivado is used to connect the Background Subtractor IP with the other hardware recourse as described in the previous paragraph. Vivado is also used to generate the bitstream file, this represents the final result of the Xilinx tools workflow, with the HW optimized background subtraction module, and it is shown in the lower part (programmable logic or FPGA) of Figure 4.14 where the complete system module is presented. In fact, it is possible to distinguish two zones: the processing system (CPU) zone and the programmable logic (FPGA) zone; a global description about their interaction while running the algorithm must also be noted. The Zenith camera inputs the video streaming through the Ethernet port to the CPU, which extracts the video frames and writes them to the input buffer in RAM. Once ready, the CPU communicates with the FPGA, that reads the video stream from the input buffer, processes the background subtractor and writes the result in the RAM out buffer. The CPU receives a signal from the FPGA specifying that the processing is done, and reads the output buffer. In the final stage, the CPU reads the output buffer and introduces the output stream from the last part of the algorithm, where the Kalman tracker extracts the people coordinates and trajectories. The JSON streaming then is prepared with the processed data and outputs it using the Ethernet interface from the CPU to the control server.

Figure 4.14: Complete Zynq structure used in this work.

The FPGA internal structure showed in the programmable logic part of the Figure 4.14, includes two additional blocks, the AXI video direct memory access (AXI VDMA) [6] cores, which are a Xilinx IP that provide high-bandwidth direct memory access between the memory RAM and the background subtractor module. A functional description of the FPGA is the following: On one side, all the AXI lite signals (depicted in yellow in Figure 4.14) are the control signal bus communication with the CPU; the AXI stream (depicted in green in Figure 4.14) is the bus used for the video streaming; and the AXI master (depicted in red in Figure 4.14) is used to interchange the vector data (*modeUsed*, *mean*, *variance* and *weight*). The background subtractor module, which uses the three interfaces, receives the video input images from the AXI VDMA and the vector data from the AXI master bus. Once the transfer is completed, the processing starts. When the module completes the background subtraction, the output is performed using the AXI streaming for the output images, and the AXI master for the vectors. On the other side, the CPU communicates with the FPGA through two ports: the master general purpose port (M_AXI_GP depicted in purple in Figure 4.14) that is used to control the FPGA in every internal stage over the AXI lite interface, and the slave high

performance ports (S_AXI_HP depicted in blue in Figure 4.14) that are used to perform the data read/write (R/W) operations on the DDR controller. The S_AXI_HP port communicates with the FPGA internal AXI stream and AXI master interfaces and permits to the FPGA the direct access to the video input/output buffers and to the vectors buffer zone.

4.3 Operating System Customization

This paragraph is intended to provide more details regarding the different strategies applied and the modules of the system required to run the algorithm. The Operating System (OS) is important because is the base of an embedded system, it needs to be robust, stable and tailored to run the application in real time. The first version of operating system used was Yocto linux [58], an open source project which provides templates, tools and methods that permits to create custom Linux-based systems for embedded and IOT products, stable and adaptable to industry standard. Even if this Yocto is a very good option, the high level of modification needed to deal with the FGPA on the ZedBoard is provided by Petalinux [59, 60], a set of tools developed by Xilinx able to generate a Linux OS distribution based on Yocto, customized for the Xilinx boards. In fact it is possible to build the OS using a Board Support Packages (BSP) as base, this includes the right boot loader and more board specifications to grant that the OS boot and functionality in this case on the Zed Board. An important feature required is the device tree [61, 62] customized compilation. The device tree is a standard used to describe the non-discoverable hardware to the kernel during the boot operation. An example of device tree is following:

```
/dts-v1/;
/ {
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };

    gpio@101F3000 {
        compatible = "arm,pl061";
    };
}
```

```
};

interrupt-controller@10140000 {
    compatible = "arm,pl190";
};

external-bus {
    ethernet@0,0 {
        compatible = "smc,smc91c111";
    };

    i2c@1,0 {
        compatible = "acme,a1234-i2c-bus";
        rtc@58 {
            compatible = "maxim,ds1338";
        };
    };

    flash@2,0 {
        compatible = "samsung,k8f1315ebm", "cfi-flash";
    };
};
};
```

Listing 4.1: Device tree example from [61]

This example shown in 4.1 represents a part of a hardware description with the CPU, the gpio interface, the interrupt controller, the Ethernet peripheral, the i2c interface and the flash card. In the case of Zed Board, is important to include non only the board information, but also the FPGA addresses as specified in paragraph 4.1 figure 4.10, because the OS have to find the device as peripheral to interchange data.

```
amba_pl {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    compatible = "simple-bus";
    ranges;

    apply_fpga@43c00000 {
        (compatible = "generic-uio");
        reg = <0x43c00000 0x10000>;
        xlnx,s-axi-crtl-bus-addr-width = <0x8>;
    };
};
```

```

        xlnx,s-axi-crtl-bus-data-width = <0x20>;
    };

    dma@43000000 {
        #dma-cells = <0x1>;
        clock-names = "s_axi_lite_aclk", "m_axi_mm2s_aclk", "m_axi_mm2s_aclk",
            "m_axi_s2mm_aclk", "m_axi_s2mm_aclk";
        clocks = <0x1 0xf 0x1 0xf 0x1 0xf 0x1 0xf 0x1 0xf>;
        compatible = "generic-uio";
        interrupt-parent = <0x4>;
        interrupts = <0x0 0x1d 0x4 0x0 0x1e 0x4>;
        reg = <0x43000000 0x10000>;
        xlnx,addrwidth = <0x20>;
        xlnx,flush-fsync = <0x1>;
        xlnx,num-fstores = <0x3>;

        dma-channel@43000000 {
            compatible = "xlnx,axi-vdma-mm2s-channel";
            interrupts = <0x0 0x1d 0x4>;
            xlnx,datawidth = <0x18>;
            xlnx,device-id = <0x0>;
        };

        dma-channel@43000030 {
            compatible = "xlnx,axi-vdma-s2mm-channel";
            interrupts = <0x0 0x1e 0x4>;
            xlnx,datawidth = <0x8>;
            xlnx,device-id = <0x0>;
            xlnx,genlock-mode;
        };
    };
};

```

Listing 4.2: Part of the ZedBoard device tree with FPGA definitions

A part of the ZedBoard device tree is showed in 4.2, the Background Subtractor accelerated function is available to the address *43c00000*, while the channels for the data memory transfers are available at *43000000* and *43000030* addresses. Petalinux provide to create the device tree for the ZedBoard including the FPGA (clock signals, type of bus, interrupt, etc), also it allows its customization. In this case the customization is represented by the word in blue "generic-uio" in 4.2 that enables the assignment of the FPGA to the UIO drivers user space. This address the FPGA as an OS peripheral allowing to interchange control signals and read/write the memory buffers directly from the same code that implement the algorithm.

An important feature allowed by Petalinux is the customization of the operating system kernel, this is mandatory to enable the I/O (UIO) driver. To grant all the functionalities, before recompiling the kernel, the following features must be set up:

- *Contiguos Memory Allocator* from *Kernel Feature* to allow the use of DMA;
- *DMA Contiguos Memory Allocator* from *Device Driver* → *Generic Driver Options* to allow the use of DMA;
- *Xilinx Device Configuration* from *Device Driver* → *Character Devices* to program FPGA from linux on Zynq devices;
- */dev/mem Virtual Device Support* from *Device Driver* → *Character Devices* to access devices directly in physical memory from */dev/mem* interface;
- *Userspace I/O platform driver with generic IRQ handling* from *Device Driver* → *Userspace I/O Driver* to enable UIO support;

In [63] is discussed an other important extension to enable before recompile the kernel to enable real time scheduling: *PREEMPT_RT*

In fact this extension allows to increase the determinism and to reduce latencies of the Linux kernel, improving the scheduler management of the various tasks executed by the CPU. This management is important because the real-time tasks have to meet the deadline, by properly assigning the CPU. More details are available in [64].

Once the kernel is configured an important step is the configuration of U-Boot [65]. This is an universal Boot loader very flexible that allows to boot from various type of storages (SD card, SATA, I²C, SPI, Ethernet, etc). This boot loader supports the ZedBoard, in fact, following this guide [66] is possible to build it using the XSDK IDE mentioned in paragraph 2.4, or the Petalinux tools environment [60]. U-Boot requires its boot commands to explicitly specify the physical memory addresses as destinations for copying data (kernel, ramdisk, device tree, etc.) An important parameter for U-boot is the *mem* parameter, that allows to create a free RAM zone as buffer area, reducing the RAM memory available for the OS to 386MB, while the other part is left free. U-boot parameters are the following:

```
bootargs=console=ttyPS0,115200 earlyprintk uio_pdrv_genirq.of_id="generic-uio"
mem=384MB root=/dev/mmcblk0p2 rw rootwait
```

Listing 4.3: U-boot parameters

Where the parameters are launched on boot and provide the following functionalities:

- *bootargs=console=ttyPS0,115200* configure the serial terminal interface and speed;
- *earlyprintk* shows early boot messages;
- *uio_pdrv_genirq.of_id="generic-uio"* enable the UIO driver;
- *mem=384MB* set the amount of memory used by the Operating System leaving a free zone that is used as buffer area.
- *root=/dev/mmcblk0p2 rw* specify the position of the Operating System root;
- *rootwait* Wait for root device to be recognized;

The buffer area is used to interchange data with the FPGA, and it needs a free zone to avoid conflict with the OS memory zone.

Free Zone	<ul style="list-style-type: none"> - FPGA Read/Write - Input Frames - Output Frames - Data Vector
OS Zone	<ul style="list-style-type: none"> - OS variables - OS processes - OpenCV Variables - etc

Figure 4.15: Ram Distribution and utilization on the ZedBoard.

In figure 4.15, it is possible to see that the RAM memory has been divided into two main parts: the Operating System zone, where all the Linux processes, variables, and the OpenCV data are stored, and the Free Zone, where all the memory buffers, that contains the input/output video frames and the vectors (*modeUsed*, *mean*, *variance* and *weight*) are stored.

The last step to have the Operating System installed and ready, it to create the SD card that contain it, for the ZedBoard. The card must have two partition, the first one of about 100MB with the fat16 or fat32 file system, that contains the boot files, while the second partition with the ext4 file system, that contains the Operating System files, of the SD card remaining space size. The first partition must have the following files:

- *BOOT.BIN* this file contains the U-boot bootloader and the FPGA bitstream file;
- *image.ub* this file is the kernel image;
- *plnx_arm_system.dtb* this file is the compiled device tree;

4.4 Driver Implementation

The type of driver described in this paragraph, is the UIO (Userspace I/O) driver [67]. It is a simple and convenient way to implement a driver for an embedded system peripheral.

Figure 4.16: Userspace I/O driver structure from [7]

Figure 4.16 shows a UIO driver structure that resume its operating mode. This typology of driver is developed in the userspace with the application that use it and all the tools and libraries needed by the developer. The UIO driver is connected with a small kernel module through the interface `/dev/uioX`, it calls only a few internal kernel functions and is easy to write and maintain. Most of the functionality is handled in a generic way by the UIO framework, avoiding to the developer the direct interaction with the internal kernel modules, reducing the development complexity including its debugging, because the bugs in the driver do not crash the kernel.

The driver is written in C++, the same language of the algorithm. It works in the two memory zones, and should permits the complete interaction between the FPGA and the algorithm that runs on the CPU. An example about the needed driver functionality is following: the CPU receives the video sequence, writes it on the input memory buffers in the Free Zone and launches the Background subtractor. The FPGA reads the input memory buffers, processes the data, writes the output and asserts a done signal. The CPU reads the output data from output memory buffer and provides it to the following part of the algorithm, that performs the tracking to final provide the JSON data as system output.

After detailing the idea about the driver functionality, an interface analysis is needed to organize the data interchange between CPU and FPGA. According to the previous paragraphs, where the hardware addresses and the device tree has been specified, the Operating System see the FPGA as a peripheral and is possible to communicate with VDMA and the Background Module using the following interfaces and channels:

- `/dev/uio0` is the Operating System interface connected to the Background Subtractor module, over AXI lite that has the address `43c00000`;
- `/dev/uio1` is the Operating System interface connected to the VDMA block, over AXI lite that has the address `43000000`;
- the VDMA read AXI stream channel connected with the Free Zone input frame buffer in RAM;
- the VDMA write AXI stream channel connected with the Free Zone output frame buffer in RAM;
- the Background Subtractor read/write AXI master channel connected with the Free Zone data vector buffer in RAM;

The in/out image and vectors data exchange, is performed using the DMA (Direct Memory Access). It is a method which allows to the peripheral direct communicate with the main memory to send or receive data independently from the CPU.

Figure 4.17: Direct Memory Access architecture from [8]

In fact, as depicted in figure 4.17, the input/output devices are directly connected with the memory through the DMA controller. This architecture allows to speed up memory operations reducing the CPU overhead. More information are available in [68].

In the specific case of the VDMA core [6], it has several registers, which needs to be set with data to control the mode of operation.

In this driver we are using the Control and the Status register, respectively to control and debug the VDMA.

4.4.1 VMDA configuration

The *MM2S_CR* and *S2MM_CR* are the control registers with same fields for controlling the read and write channels respectively. The fields of the control register are as shown in the figure 4.18. The bits of position 0 and 1 of the control register are of interest to set the channels in the required mode. Bit zero, RS- Run/Stop, starts and stops the channel from

accessing the memory and hence the data transfer. Bit one, Circular_Park, configures the channels to work in circular buffering mode or park mode.

Figure 4.18: VDMA control register layout from [6]

The *MM2S_SR* and *S2MM_SR* are the status registers with similar fields for monitoring the read and write channels respectively. The fields that capture errors, resulting in halting of the channel, are of interest. These fields of the registers are as shown in the figure 4.19. The bits six through four of the status register corresponds to DMADecErr, DMAStvErr and DMAIntErr, which results in channel halt. It is crucial to monitor these errors so that, when the channel halts, a soft reset is issued to clear the error flags and restart the channels.

Figure 4.19: VDMA status register layout from [6]

The content of the registers are available for the PS to access using the AXI Lite Bus. To access the same content from the Linux user space, the physical address of the VDMA registers must be mapped. This mapping of the physical address to memory is done in the user space to reduce the complexity of the software using the `mmap` function [69, 70] as shown in the code 4.4. The `mmap` function performs the mapping of the physical address to the virtual address and returns the virtual address after successful mapping, providing access to the VDMA registers.

```
static void configureVDMA(void){
    int size;
    int fd = open( "/dev/uiol", O_RDWR);
    size=get_memory_size("/sys/class/uio/uiol/maps/map0/size");
    unsigned char* base_address;

    base_address = (unsigned char*)mmap(NULL, size, PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, 0);
    //check if mapping was successful
    if(base_address == MAP_FAILED)
    {
        std::cout<<"VDMA Mapping memory for absolute memory access failed.\n";
        return;
    }
    streamDevHandler.vdmaDevFD= fd;
    streamDevHandler.vdmaDevVirtAddr=base_address;
    streamDevHandler.vdmaMemLength=size;
}
```

Listing 4.4: Memory mapping VDMA registers

```
static void configureVDMAInBuffers(int num_frame){
    int map_len = IN_FRAME_WIDTH * IN_FRAME_HEIGHT * IN_BYTES_PER_PIXEL;
    for (int i=0; i<num_frame; i++ )
    {
        int fd = open( "/dev/mem", O_RDWR);
        unsigned char* base_address;
        unsigned long int PhysicalAddress = In_PhysicalAddress[i];
        base_address = (unsigned char*)mmap(NULL, map_len, PROT_READ |
            PROT_WRITE, MAP_SHARED, fd, (off_t)PhysicalAddress);
        //check if mapping was successful
        if(base_address == MAP_FAILED)
        {
            std::cout<<"Input Frame " << i << " mapping memory for absolute
                memory access failed.\n";
            return;
        }
        videoFrameHandler[i].mem2VdmaFrameFD = fd;
        videoFrameHandler[i].mem2VdmaFrameVirtAddr = base_address;
        videoFrameHandler[i].mem2VdmaFrameMemLength = map_len;
    }
}
```

Listing 4.5: Memory mapping VDMA input buffer

The VDMA channels need to be mapped, to send the input/output images, the function is similar, but it is mapped to the system direction `/dev/mem`. The code showed in 4.5 represents the image input buffer mapping. As in the Free zone more frames are stored, the *for cycle* maps one address par frame. The Output buffer mapping is similar to 4.5.

4.4.2 Video Stream Data Read and Write transactions

Since the read and write channels are configured to run in continuous mode, the bits for starting and stopping the frame data transfers need not be polled. However, the following registers must be set with appropriate data to initialize the VDMA core for memory access [71].

- MM2S Vertical Size (MM2S_VSIZE) and S2MM Vertical Size (S2MM_VSIZE) registers are a 13-bit long field for specifying the height of the frame. They also have a second purpose of starting the channels on writing to these registers. It must be noted that for any channel this field must be written at the end to prevent premature start of the data transfer.
- MM2S Horizontal Size (MM2S_HSIZE) and S2MM Horizontal Size (S2MM_HSIZE) register is a 16-bit long field for specifying the width of the frame in terms of number of bytes per pixel. If a pixel is a 24-bit (3 bytes) data, as in case of an RGB image of 8-17 bit depth, then the horizontal size must be width of the frame multiplied by the number of bytes per pixel. For example, for an RGB 8-bit depth image of frame width 480 pixels, HSIZE value should be set to $480 \times 3 = 1440$.
- MM2S Frame Delay and Stride (MM2S_FRMDLY_STRIDE) and S2MM Frame Delay and Stride (S2MM_FRMDLY_STRIDE) registers are used to set the number of address bytes between the first pixels of each video line. The stride value is calculated similar to the horizontal size.
- MM2S Start Addresses and S2MM Start Addresses are the registers for specifying the physical address of the image buffers. VDMA read channel accesses the image frame from this physical address in the memory and the write channel transfer the image frame to this physical address. Mapping of buffers in the Linux user space has been described in the previous paragraph.

Configuration of the VDMA is done as an initialization step, which sets the required parameters and starts the channels. It is important that the setting of the vertical size is done as the last step in configuring a channel. It is important to note the stream data width between the VDMA and the Background Subtractor module. This width is equal to the depth of the image. For the RGB version the input image with is 8 bits pixel value for each component, the stream width should be 24 bits, while the output Gray-scale image having only one component, has a stream width of 8 bits. For the Grayscale version the input an the output images have the same components, and the stream width is 8 bits for both channels. Therefore, the width of the interface between the Background Subtractor core and the VDMA core must match. This depth value must also be considered when setting the stride and horizontal size values in the VDMA core registers [71].

4.4.3 Background Subtractor Configuration

As the VDMA, the Background Subtractor module needs to be mapped to be available from the system. In this case is not complex as VDMA because from one side, the VDMA provides the video data to process performing the frame management, while for the vectors it needs a values copy. The vectors hold the Background Subtractor history, the operation needed is to copy data from the RAM buffer before starting the processing, and save the data back at the end of the processing. The mapping needed is to the `/dev/uio0` and to the `/dev/map` interfaces.

```
static void configureImageFilter(void){
    int map_len = get_memory_size("/sys/class/uio/uio0/maps/map0/size");
    int fd = open( "/dev/uio0", O_RDWR);
    unsigned char* base_address;
    std::cout<<"Map Length: "<<map_len<<std::endl;

    base_address = (unsigned char*)mmap(NULL, map_len, PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, 0);
    //check if mapping was successful
    printf("Background Subtractor address buffer:  %p \n", base_address);
    if(base_address == MAP_FAILED)
    {
        std::cout<<"Background Subtractor Mapping memory access failed.\n";
        return;
    }
    imageFilterHWHandler.imageFilterHWFD = fd;
```



```

imageFilterHWHandler.imageFilterHWVirtAddr = base_address;
imageFilterHWHandler.imageFilterHWLength = map_len;
}

```

Listing 4.6: Memory mapping for Background Subtractor registers

```

static void configureVectorsBuffer(void){
    int map_len = MAX_WIDTH * MAX_HEIGHT * sizeof(t_data);
    int fd = open( "/dev/mem", O_RDWR);
    unsigned char* base_address;
    unsigned long int PhysicalAddress = Vectors_BUFFER_BASEADDR;
    base_address = (unsigned char*)mmap(NULL, map_len, PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, (off_t)PhysicalAddress);
    //check if mapping was successful
    if(base_address == MAP_FAILED)
    {
        std::cout<<"Vectors Buffer mapping memory access failed.\n";
        return;
    }
    printf("Vectors address buffer:  %p \n", base_address);
    vectorsDMAHandler.vectorsFD = fd;
    vectorsDMAHandler.vectorsAddr = base_address;
    vectorsDMAHandler.vectorsLength = map_len;
}
}

```

Listing 4.7: Memory mapping Background Subtractor vectors buffer

The code depicted in 4.6 represents the Background Subtractor mapping that make the control interface available thorough AXI lite, while the code shown in 4.7 connects the Background Subtractor vectors port to the RAM Vectors buffer over the AXI master interface.

4.4.4 Video Stream Data Read and Write transactions

The Background Subtractor module consists of control registers for configuring the core, controlling the behavior of the algorithm and generation the output depending on the information read and written from/to the vectors.

```
void apply_fpga(AXI_STREAM_IN& src_in, AXI_STREAM_OUT& dst_out,
    float *weight_window, float *variance_window, float *mean_window, int *modesUsed_window,
    int num_gaussian, int numchannels, int nmixtures, float alphaT, float Tb, float TB,
    float Tg, float varInit, float varMin, float varMax, float prune, float tau,
    bool detectShadows, unsigned char shadowVal, int cols, int rows)
{
```

Listing 4.8: Background Subtractor header function parameters

The code 4.8 is the header of the synthesized function that calculate the Background Subtractor into the FPGA. The parameters in green represents the AXI stream interface, the parameters in red are the AXI master interface, while the parameters in orange are the AXI lite interface. While the AXI streaming and AXI master has been mapped for separate interface as detailed in the previous paragraphs, the AXI lite are mapped using the `/dev/uio0` Operating System interface. The algorithm parameters must be updated each time the processing is launched because they could vary their value and in some case do not only pass some value for calculation, but also information such as the number of column or rows. An other important register that is received through the AXI lite is the AP_CTRL (Control signals) for controlling the core. AP_CTRL registers have the following fields:

- bit 0 - ap_start (Read/Write/COH): Setting the bit to 1 starts the conversion and is cleared on completion.
- bit 1 - ap_done (Read/COR): The bit is set to 1 when the conversion is complete.
- bit 2 - ap_idle (Read) is set to 1 in idle.
- bit 3 - ap_ready (Read) bit is set when the image processing core is ready for the conversion.
- bit 7 - auto_restart (Read/Write). Setting the bit to 1 restarts the conversion after completion of the current frame.

The Background Subtractor calculation is started by setting the ap_start bit of the AP_CTRL register and the ap_done bit is polled for completion. On completion, processing of the next frame is started. For more information about the Xilinx drivers and on how to connect the Vivado HLS IP is possible to consult [72, 73].

4.4.5 Parameters Calculation and Automating

As the design works with more formats, a lot of parameters need to be calculated to arrange the data interchange between CPU and FPGA. These parameters are useful to automate for example the RAM memory allocation needs by buffers, the number of pixels to interchange with the FPGA etc.

Frame Width	Frame Height	VDMA Height	Internal FPGA Memory elements	Frames to buffer In RAM
320	240	48	15360	15
480	480	32	15360	15
512	512	32	16384	15
640	480	24	15360	15
768	768	16	12288	15
1056	1056	12	12672	10

Table 4.1: RGB parameters for the FPGA data interchange, based on video resolution.

Frame Width	Frame Height	VDMA Height	Internal FPGA Memory elements	Frames to buffer In RAM
320	240	96	30720	15
480	480	60	28800	15
512	512	64	32768	15
640	480	48	30720	15
768	768	32	24576	15
1056	1056	24	25344	15

Table 4.2: Gray-scale parameters for the FPGA data interchange, based on video resolution.

Table 4.1 and 4.2 contains a fundamental set of parameters calculated according to the FPGA and RAM memories available. The parameters are detailed as follows:

- *Frame Width* and *Frame Height* represent the resolution of the image.

- *VDMA Height* represent the max resolution of the internal image possible to process into the FPGA.
- *Internal FPGA Memory Elements* is the max number of pixels and vectors components that is possible to store into the FPGA.
- *Frames to buffer in RAM* is the max number of Frame that are stored in RAM. More details about this parameters are detailed in the paragraph 5.3.

It represent a good idea to introduce these parameters into a static vector to load them when the image format is defined or detected. An other important operation to automatize is the Ram addresses for vectors and Frame Buffers calculation. The base concept is that the virtual memory is divided in pages and the addresses must take into account this in the calculation. For more details please consult [74].

```
if IN_FRAME_WIDTH == 1056 && IN_FRAME_HEIGHT == 1056

#define Vectors_BUFFER_BASEADDR                0x1707D000

#define MEM2VDMA_FRAME01_BASEADDR              0x18182000 /* 404234240 */
#define MEM2VDMA_FRAME02_BASEADDR              0x18293000 /* 405352448 */
#define MEM2VDMA_FRAME03_BASEADDR              0x183A4000 /* 406470656 */
#define MEM2VDMA_FRAME04_BASEADDR              0x184B5000 /* 407588864 */
#define MEM2VDMA_FRAME05_BASEADDR              0x185C6000 /* 408707072 */
#define MEM2VDMA_FRAME06_BASEADDR              0x186D7000 /* 409825280 */
#define MEM2VDMA_FRAME07_BASEADDR              0x187E8000 /* 410943488 */
#define MEM2VDMA_FRAME08_BASEADDR              0x188F9000 /* 412061696 */
#define MEM2VDMA_FRAME09_BASEADDR              0x18A0A000 /* 413179904 */
#define MEM2VDMA_FRAME10_BASEADDR              0x18B1B000 /* 414298112 */

#define VDMA2MEM_FRAME01_BASEADDR              0x1AC2A000 /* 448962560 */
#define VDMA2MEM_FRAME02_BASEADDR              0x1AD3B000 /* 450080768 */
#define VDMA2MEM_FRAME03_BASEADDR              0x1AE4C000 /* 451198976 */
#define VDMA2MEM_FRAME04_BASEADDR              0x1AF5D000 /* 452317184 */
#define VDMA2MEM_FRAME05_BASEADDR              0x1B06E000 /* 453435392 */
#define VDMA2MEM_FRAME06_BASEADDR              0x1B17F000 /* 454553600 */
#define VDMA2MEM_FRAME07_BASEADDR              0x1B290000 /* 455671808 */
#define VDMA2MEM_FRAME08_BASEADDR              0x1B3A1000 /* 456790016 */
#define VDMA2MEM_FRAME09_BASEADDR              0x1B4B2000 /* 457908224 */
#define VDMA2MEM_FRAME10_BASEADDR              0x1B5C3000 /* 459026432 */

#endif /* 1056x1056 definitions */
```

Listing 4.9: Ram Addresses configuration example

The portion of code shown in 4.9 represents an example of address configuration that the driver load in case the input frame has a resolution of 1056x1056. The addresses have been calculated using a page size of 4096 bytes according to the ZedBoard CPU ARM Cortex-A9. The address has been calculated simply adding to the start address, the amount of memory that permits to contain the frame. This operation follows n times up to complete the frames buffers. In listing 4.9 for example it is repeated ten times for the input frames (MEM2VDMA), and other ten times for the output frames (VDMA2MEM). An important consideration is that each address must be divided by 4096 to avoid crashing the system during the mapping process (shown in listing 4.5).

Chapter 5

Test and Validation

In this chapter the test and validation of the system is described. As the system realized in this work runs on a hybrid platform, its verification focuses on two main areas, the hardware area consists in all the tests that permits to analyze the FPGA and debug its functionalities, while the software area includes all the test that allows to evaluate the algorithmic part.

5.1 Hardware

The programmable System-on-Chip platforms combine FPGAs with general purpose processors. SoC are suitable for running operating systems and to also accelerate algorithms at the hardware level as already detailed. As the complexity and size of these devices is increasing, the verification techniques play an important role to validate the design concreteness and to reduce the hardware errors and the time to market. This represents a big challenge for design and verification engineers. In fact in the design process an entire team deals with verification, as the complexity of the design increase, the time dedication to the verification arise as detailed in this study [9]. As an example, figure 5.1 depict the time dedication of the engineers to the various verification phases. Considering a real application this is further emphasized, because the design needs to be verified according the operating conditions. In fact the test must includes the communications between the various part of the design, any synchronization with external interfaces, checking the design correctness across various levels.

To analyze the correctness of a design from a functional point of view, an additional portion of code called test-bench [75] is written, that allows to introduce certain signal stimuli into the design and compare the output generated with the expected ones.

Figure 5.1: Engineers time dedication performing Verification from [9]

The verification process also has been automated to determine when the design is ready, using a metric-driven implementation such as code coverage that helps to determine its functionalities analyzing the code of the design. Furthermore many methodologies have been developed to verify the hardware, that can vary from formal verification [76], simulation [77], fault injection [78], debugging [79], fault mitigation [80], Universal Verification Methodology (UVM) [81] etc. These methodologies, available in the cited references, can be applied depending on the design requirements and at different stages in the design flow. It is important to select the right strategy depending on design complexity and required fault coverage. This could affect the verification methodology and plan that will be followed for each design, to ensure the functional correctness of the entire system.

In this thesis a mix of these methodologies is applied to debug the hardware at various levels. An interesting approach is the one presented in our previous work [21], that will be adapted to debug a part of the design. In fact the key concept depicted in [21], allows to perform the verification through an interface where the direct introduction of a stimulus could be a challenge stage by stage. In this case, the stimulus signal that drives the hardware test, will be introduced by the CPU through the AXI interfaces, that will return the result of the FPGA processing to compare it with the expected one.

5.1.1 Vivado HLS pre and post synthesis

As first step, the verification of the algorithm starts in Vivado HLS. In fact a Test-bench has been created to check the functionality of the Background Subtractor algorithm, modified as described in the previous paragraphs. In this case, that is a software simulation debug, the tool allows to open a video test from a file, that is impossible to do into the FPGA, and provide all the function parameters and data, simulating the communication busses (AXI Lite, AXI Streaming and AXI Master).

Figure 5.2: BackGround Subtractor algorithm debug on Vivado HLS

This verification test is very important because it allows to check the variable type data change explained in paragraph 3.4, but also it permits to validate the algorithm porting from software to hardware. Figure 5.2 shows the debug stages that are explained as follows:

- 5.2 *a)* represents a frame of the input video that is received by the algorithm to process the Background Subtractor. In the blue dashed rectangles has been manually marked the zone where the people are present, that should be identified by the algorithm.
- 5.2 *b)* is the output image processed by the algorithm as reference. This image is provided by the algorithm implementation using the Eclipse IDE before the porting to Vivado HLS.
- 5.2 *c)* is the first version ported to Vivado HLS that present some algorithm implementation errors visible as additional white dots noise.
- 5.2 *d)* represents the final version of the algorithm where the porting has been successfully completed.

The algorithm debug started in C++ using the Eclipse IDE that has been used to apply the modifications explained in paragraph 3.4, where has been changed the data structures from dynamic to static and all the code instruction not supported by Vivado HLS has been modified. Furthermore the Background Subtractor function call has been updated to have the same parameters as in the FPGA, preparing the parameters interchange to be performed using the AXI buses. Taking the image 5.2.*b* as reference and starting point, all the new versions of the algorithm must give the same results to avoid algorithmic errors before the porting to the hardware. This is done using the Eclipse IDE because Vivado HLS is not the correct tool for this operations since it is very slow. In fact the simulation take time and in this case it corresponds to a normal software debugging process that can be easily made with a software tool such us the Eclipse IDE.

Once the algorithm is ready to be introduced in Vivado HLS the verification is still a debug, but it uses this latter tool to follow implementing the further modification described in 3.4. These changes are closest to the hardware as for example the variable types to meet the memory requirements, removing the divisions, variable splitting into more buffers, etc. Figure 5.2.*c* represents the debugging stage of these changes that includes the algorithm acceleration. In fact the noise that is possible to appreciate, is produced by the errors that occurs applying the changes. The algorithm needs to be verified each time is modified to check its correctness. The errors occurs due to wrong buffers assignation, wrong formulas implementation, wrong *for*

cycle parameters optimization etc. An example is the case of the equation 3.4 where to avoid the division, it has been approximated as commented in paragraph 3.4. This approximation plays an important role on the algorithm decision threshold that introduce the noise. In particular the noise generated as white dots visible in figure 5.2.c will produce an error for the following stages, that could localize people in a wrong position. According to the *Original frame* and to the *BGS reference* (figure 5.2.a,b) in the bottom part of the image there aren't people, while in the 5.2.c in the bottom part, the error will make the system localizing people in that zone. The debugging of the algorithm permits to correct the errors improving all the parameters up to reach the final stage represented in figure 5.2.d where the noise has been considerably reduced. At this point the algorithm correctness is proved and it has been accelerated to be packed as Vivado HLS IP. The last verification post synthesis is done using Vivado HLS C/RTL Cosimulation that it is correct as produce the same result shown in figure 5.2.d. In case the output is not correct the synthesis process should be investigated for RTL translation errors.

5.1.2 Vivado FPGA synthesis verification

As the IP verification success, the next stage is direct to its functionality into the FPGA. The Background Block has been connected as depicted in paragraph 4.1, where the hardware has passed all the Vivado tests that includes the Design Rule Check (DRC), timing check and the bitstream file is ready to be loaded on the FPGA. A possible check is to route the internal signals to the board output pin and analyze it with a logic analyser as described in [82], but due to the high data flow, this technique has been used only to debug some control signal. As the module verification has been performed in the previous paragraph, the test focuses on the communication using the AXI buses and verify the correctness of the data interchange to perform the communication with the rest of the algorithm. This test also involve the driver testing, in fact the CPU during the execution of the algorithm, will send the data to the FPGA that will start its processing using the commands mentioned in the paragraph 4.4.4. Basically the CPU send the video data and the vectors data in the RAM input buffers, launches the FGPA using the AXI lite. The FPGA reads the vectors data using the AXI master bus and the video data using the AXI stream bus over the VDMA, that coordinate the video transfers. Once received the data the Background Subtractor processing starts. When the Processing ends, the FPGA execute all data transfer to the RAM output buffers and notify to the CPU that the processed data is ready. The CPU read the data provided in the RAM output buffer and follows with the tracking algorithm. After describing globally how the processing happens, the verification strategy needs to be adapted to this

system to prove the algorithms execution correctness on this hybrid platform. It is necessary to perform the test from the CPU that is executing the algorithm, introducing the data over the three AXI buses into the FPGA, and debugging the manner in which the FPGA responses to external stimuli.

The method adopted in this case is to replace the Background Subtractor module with a testing block that performs elementary tasks to simplify the debugging process. To verify the video streaming a simple task is performed using a color converter from RGB to Gray scale, as shown in [83] using the function `hls::CvtColor<HLS_BGR2GRAY>(src, dst)` offered by Vivado HLS video library [42]. Another simple task to verify the vectors AXI master transfer rate, is to introduce the input vectors having known values into the FPGA, as internal operation to add a constant and provide the vector output as a sum between the input and the constants. The control signals and the parameters sent over AXI lite are not complex to debug, because for the control signals is sufficient that the FPGA responds to the input commands as expected (functioning in the same way described in 4.4.4), while for the parameter, it is enough to introduce their values as FPGA input, and read the same value as FPGA output.

Figure 5.3: Vivado Hardware Test Diagram

A module that performs the operations previously described has been realized in Vivado HLS, It has been synthesized, packed into an IP and replaced to the Background Subtractor in the

Vivado project, as shown in figure 5.3. In fact the connections are the same as figure 4.6, but a test module called *Grayscale converter & Sum Block Test* that performs the operations described, has been introduced replacing the original Background Subtractor module. The hardware is ready to test, a C++ code is needed to run the algorithm and create a log activity with the FPGA inputs/outputs, to analyze the data transferred and debug the system.

Figure 5.4: FPGA AXI stream and VDMA verification

In the AXI stream and VDMA case, the test is performed introducing a video sequence into the FPGA and saving the video output sequence to analyze the data transfer. The input stream is a RGB format and as output a Gray scale format is expected. Figure 5.4 shows the

test and a brief explanation is following:

- 5.4 *a)* is the input RGB frame sent from the CPU to the FPGA.
- 5.4 *b)* represents a communication error about the VDMA parameters. The error depends on the wrong parameter selection such as: internal buffer size, Read/Write burst size, Input Input/Output stream width, synchronization signals etc. This generate the artifacts and the image is not clear.
- 5.4 *c)* reproduces a synchronization error. In fact most of parameters are set in the correct way, but a the synchronization with the Start of Frame (Sof) is failing. This generate a little shift on the image.
- 5.4 *d)* All the parameters have the right setting and the output image is as expected.

This method has been used to set the VDMA parameters ad depicted in figure 4.8 of paragraph 4.1, repeating the test and adapting the parameters described in [6] until the correct configuration for this project has been founded.

The AXI master verification is carried out with the sum test algorithm shown in figure 5.5.

Figure 5.5: Vectors Sum Test algorithm for AXI master debug

The test consists into generate a known values vectors, introduce them into the FPGA that sums an established constant for each vector as follows: +1 for the *Variance* vector, +2 for the *Weight* vector, +3 for the *modesUsed* vector and +4 for the *Mean* vector. As already mentioned, the FPGA has a limited internal memory, it is necessary to divide the vectors in more parts and process them part by part. For this reason in figure 5.5 there is a loop that repeat the process *N buffer* times. Also to understand which part of the input vectors is under process, it has been created with more values divided by zones following this pattern: Zone 1 vectors values 1000, Zone 2 vectors values 2000, Zone 3 vectors values 3000, Zone n vectors values n000. Using this method helps to create a known condition to check the system as depicted in table 5.1.

Vector	Constant to add	Input value Zone 1	Expected Out value Zone 1	Input value Zone n	Expected Out Zone n
Variance	1	1000	1001	n000	n001
Weight	2	1000	1002	n000	n002
modesUsed	3	1000	1003	n000	n003
Mean	4	1000	1004	n000	n004

Table 5.1: Vectors Debug Values

Running this test algorithm allows to verify the AXI master bus connection between is the FPGA and the CPU and the data interchange. All the transaction are saved in a log file to analyze and debug the system. A part of this log file is showed in listing 5.1.

The text in blue shows the peripherals mapping of the Background Subtractor module, the vectors buffers and the VDMA register. These addresses showed are a possible output about the example codes available in the driver section depicted in paragraph 4.4, where the function *mmap* is employed to assign a virtual address to each device.

Following the log file analysis there are three zones: the *Writing Zone* copies the vectors data into the RAM buffer, while the *Reading Zone* performs the opposite operation of copying data from the RAM buffer. The *Running FPGA Filter* represents the zone where the hardware is launched to perform the data transfers and the processing tasks, that consists into summing the constants as already described.

The portion of logfile is taken when the FPGA is processing the data and it is giving the output as expected in table 5.1. The debug has been performed correcting the errors (such as read/write wrong zones, wrong use of the communication bus, etc) until prove that the AXI master bus is working properly.

```

Map Length: 65536
Background Subtractor address buffer: 0xb33c1000
Variance address buffer: 0xb326f000
Weight address buffer: 0xb311d000
modesUsed address buffer: 0xb30e4000
Mean address buffer: 0xb1a29000
VDMA MM2S Control Register:65537
VDMA S2MM Control Register:65537
***** Writing Zone = 1 *****
Copy Variance vector to FPGA RAM buffer to Address 421527552, Value: 1000
Copy Weight vector to FPGA RAM buffer to Address 432013312, Value: 1000
Copy modesUsed vector to FPGA RAM buffer to Address 442499072, Value: 1000
Copy Mean vector to FPGA RAM buffer to Address 446693376, Value: 1000
***** Running FPGA Filter *****
Running FPGA Filter Time us = 796
***** Reading Zone = 1 *****
reading Variance from Ram Address 421527552: 1001
reading Weight Ram Address 432013312: 1002
reading modesUsed Ram Address 442499072: 1003
reading Mean Ram Address 446693376: 1004
***** Writing zone = 2 *****
Copy Variance vector to FPGA RAM buffer to Address 421573632, Value: 2000
Copy Weight vector to FPGA RAM buffer to Address 432059392, Value: 2000
Copy modesUsed vector to FPGA RAM buffer to Address 442506752, Value: 2000
Copy Mean vector to FPGA RAM buffer to Address 446831616, Value: 2000
***** Running FPGA Filter *****
Running FPGA Filter Time us = 796
***** Reading Zone = 2*****
reading Variance from Ram Address 421619712: 2001
reading Weight Ram Address 432105472: 2002
reading modesUsed Ram Address 442514432: 2003
reading Mean Ram Address 446969856: 2004

```

Listing 5.1: Vectors copy and modification log

Furthermore the same log is useful to check the buffer memory addresses in RAM. In fact as the FPGA needs to copy parts of the vectors buffers to process data, it is possible to verify if the reading / writing process are moving over the RAM using the right memory amounts, that are summed each time to jump to the next position.

The AXI lite verification task is performed sending the commands and the parameters form the CPU and analyzing the FPGA response as shown in listings 5.2 and 5.3

```
*****
VDMA MM2S status before vsize init: 0x1000
VDMA MM2S status after vsize init: 0x1001
VDMA MM2S status before running FPGA: 0x1001
CRTL_BUS before running command: 0x4
CRTL_BUS entering function: 0x4
CRTL_BUS launching command: 0x1
CRTL_BUS after launching command: 0x6
CRTL_BUS leaving function: 0x6
VDMA S2MM status before leaving function: 0x1001
VDMA S2MM status after leaving function: 0x1001
S2MM status register (00000000): running frame-count:0 delay-count:0
MM2S status register (00000000): running frame-count:0 delay-count:0
VDMA MM2S status before running FPGA: 0x1001
CRTL_BUS before running command: 0x4
*****
```

Listing 5.2: AXI lite control command verification log.

```
*****
num_gaussian: 3
num_channels: 3
nmixtures: 3
alphaT: 0.00195313
Tb: 16
TB: 0.899902
Tg: -7
varInit: 15
varMin: 4
varMax: 75
prune: 0
tau: 0.5
detectShadows: 1
shadowVal: 127
cols: 480
rows: 16
*****
```

Listing 5.3: AXI lite parameters verification log.

The listing 5.2 is a part of a log employed to verify the commands detailed in paragraph 4.4

where the the APCTRL signal to control the core and the VDMA status registers have been explained. The log file is written during the code execution, entering into its function and saving the status of the register to know the FPGA internal status. Concretely to understand the log, the registers status value is detailed as follows:

- FGPA Control register: 0x4 idle;
- FGPA Control register: 0x1 start;
- FGPA Control register: 0x6 done + idle;
- VDMA status register: 0x1000 MM2S synchronized to frame input, VDMA in stopped;
- VDMA status register: 0x1001 MM2S synchronized to frame input, VDMA running;

The log depicts the configuration and running states of the system. At the begin the VDMA is waiting (status=0x1000) to be programmed with the *vsize* parameter before start the operation. Once it has been programmed, the VDMA receives (MM2S) and transmits (S2MM) the video stream (status=0x1001). The FPGA is in idle (status=0x4) waiting to be launched. The FPGA receives the running command (status=0x1) and process the data. Once it finish a ready signal is asserted and the FPGA return into idle state (status=0x6). The processed data is outputted (as the VDMA S2MM status is running) and the process is repeated in loop until the tracking algorithm is ended by the user. The VDMA from this moment is running and streaming the data, the process is controlled only by the FPGA Control register.

The listing 5.3 is a dump about the algorithm parameters form the FPGA at the end of the processing. These parameters has been introduced at the beginning of the process into the FPGA. This log allows to verify the correctness of parameters avoiding errors during their transmission from CPU to FPGA over the axi lite bus (as for example wrong bus use, casting error, etc).

5.2 Software

Once the hardware verification has been performed and the FPGA side check is complete, the testing and validation about the algorithmic part from the CPU side, are required. The first analysis performed is on the data integrity, which consists of debugging and analysis to check if the data exchanged between the CPU and the FPGA on the free zone are correct, and if it is

desired to implement the tracking in the next stage. This is performed by running a short test sequence to save images and log files that contain the values of the corresponding variables. In order to generate a reference, initially the algorithm runs only on the CPU. During the verification stage, the same test sequence is launched on the Zynq system, storing debug data from the memory buffers in the free zone, before and after the execution of background subtraction on the FPGA. The data (images and log file) are compared with the reference to check the data coherency, and to determine if the data transfer between the CPU and the FPGA has been successfully performed. If the data integrity analysis is satisfactory, the next step is to test the time analysis to evaluate the performance. All the execution times are saved to a log file, to observe and investigate any eventual losses in performance. The first bottleneck that emerges from the time analysis is the detriment to RAM speed. Unfortunately, the DDR3 memory of the ZedBoard is slow because the interface speed is governed by a 533 MHz clock, allowing data transfers up to 1066 Mbs. The FPGA internal memory resources are limited, and the amount of data that can be processed is around 15,000 RGB pixels. This means that to process an image of 640×480 pixels, it needs to divide it into around twenty sub-images and process them part by part. The quantity of data to transfer also requires analysis, because it could slow down the entire data (vectors and in/out sub images) transfer process, especially when this copy should be repeated several times (e.g., 20 times for 640×480 resolution).

Upon analysis of how the data is used in MoG, around 85% of the memory is used by the vectors that are holding all changes between the various frames. The strategy used in this study, to optimize the data copies, is to create a special buffering system where a number of frames is stored in a buffer and it is processed part by part from frame buffer 1 to n transferring the vectors information only at the beginning and at the end of processing. Figure 5.6 depicts the proposed method. The number in yellow indicates the chronology of the operations described as follows:

1. Read vector data from RAM.
2. Read image in part 1(2.. n) from input RAM buffer and process background Subtractor, updating FPGA internal vectors' data.
3. Write output image part 1(2.. n) to the RAM output buffer.
4. After processing the N frames part 1(2.. n), write the updated information the vectors RAM buffer.

Figure 5.6: Frame Buffering strategy used to move data in this work.

Steps 2 and 3 are repeated n times until part 1 has been processed, and then continues with part 2, 3, ..., n until the n frames in the buffer have been processed. The system is then ready to process the ensuing frames of the stream. This strategy improves the algorithm execution time because it drastically reduces the amount of information to copy from/to RAM. The next bottleneck to address arises in the CPU and the time that it is waiting during the data copy from/to RAM. The Zynq 7Z020 has a dual-core ARM Cortex-A9 CPU at 667 MHz, and

it is possible to use C++ multi threading algorithm implementation to reduce the bottleneck represented from the data copy waiting time.

Figure 5.7 represents the processing strategy adopted by the multi-threading algorithm using a ping-pong buffer pattern[84]. The frame buffer in RAM has been divided into two zones, allowing the new version of the system to copy new input data while the FPGA is processing the current frames. There are two synchronized threads in the system: `thread_cp` and `thread_ps`. First, `thread_cp` copies the input frames in the zone 0 input buffer. Then, `thread_ps` controls the FPGA to elaborate the data on zone 0 output buffer, while `thread_cp` is copying the following frames in the zone 1 input buffer. When `thread_ps` finishes in zone 0, it starts again in zone 1, while `thread_cp` is saving the output data and preparing the new input frames on the zone 0 input and output buffers. The subsequent executions, `thread_cp` and `thread_ps` alternate the region of execution in a sequential way processing the data. Optimizing the multi-threading algorithm allows the CPU to improve the processing time by coordinating the Zynq resources and reducing the waiting times between the various stages.

Figure 5.7: Multi threading Frame Buffering.

5.3 Experimental Setup

In this section, the experiment conditions are specified. For reference, the tests have been launched using a personal computer, an embedded system and a SoC using only the CPU. The specifications of the three systems are detailed below:

- Personal computer with an Intel i7-4790 CPU and 8GB of RAM;
- Raspberry Pi 3 with a 64 bit quad-core ARM Cortex A53 (ARMv8) and 1 GB of RAM [85];
- ZedBoard with SoC XC7Z020 (that includes Dual-Core ARM Cortex-A9 32 bit CPU and an Artix-7 FPGA) and 512 MB of RAM [4].

The experiment was performed by processing some video sequences using the three architectures, and focusing on the execution times. The times were measured at two critical points of the chain: the background subtractor and the Kalman Tracker. The results of the test are shown in Table 5.2.

Most of the video sequences used for the experiment are recordings of Parkinson's patients at the *Asociación Parkinson Madrid* day hospital center, recorded legally and as part of the ICT4Life project. To provide a wider spectrum of scenarios, the testing set also includes two video sequences recorded in a shop and two other video sequences recorded in a university laboratory.

It is important to note that the two columns of the ZedBoard's times in Table 5.2 represent only CPU processing mode without FPGA HW acceleration. The algorithm used in this test is the same as the one launched on the personal computer and Raspberry Pi 3.

Sequence	Resolution	Average Num of People/Frame	i7 BS (ms)	i7 KT (ms)	Rpi3 BS	Rpi3 KT	ZedB BS	ZedB KT
shop1.mp4	640 × 480	3.5	13.3	0.04	8.84	12.5	16.2	16.5
shop2.mp4	640 × 480	8	13.6	0.08	8.87	14.5	16.6	20.75
hosp1.mp4	1056 × 1056	4.3	48.9	0.045	8.02	11.11	17.2	14.88
hosp2.mp4	1056 × 1056	3	48.4	0.051	7.6	8.82	17.14	15.5
hosp3.mp4	1056 × 1056	3.3	48.5	0.042	7.56	9.3	17.13	15.48
hosp4.mp4	1056 × 1056	2.3	49.3	0.049	5.62	7.75	17.68	15.10
hosp5.mp4	1056 × 1056	5	48.4	0.056	7.8	6.96	16.98	10
lab1.mp4	854 × 480	2	17.5	0.038	8.82	12.37	16.69	17.37
lab2.mp4	1920 × 1080	3	88.7	0.05	8.93	12.2	17.03	15.4

Table 5.2: Time measurements (in ms) to process one frame depending on the video sequence and the resolution for the i7. Rpi3 and ZedB are presented in i7 slower ratio. BS stands for background subtraction, KT stands for Kalman Tracking, i7 represents the personal computer, Rpi3 stands for Raspberry pi 3 and ZedB stands for ZedBoard.

Table 5.2 shows the time measurements of the experiment. It takes as reference the i7 times in milliseconds and presents the Rpi3 and ZedBoard measurements as a ratio, which expresses how much slower they are compared to the i7. Each measurement represents the time needed to process one frame. As a first consideration, the number of people per frame increases the complexity and the processing time (more people = more time needed to process the frame). The important consideration that emerges from the data is that the background subtraction is the slowest part of the algorithm; i.e., it is the module demanding more resources. The Kalman tracking, even if its complexity increases with a higher number of people per frame, has a minor impact on the total processing time. To guarantee real-time conditions, in this case, the processing time should not exceed 66.67ms per frame.

$$T_{FRAME} = BS + KT \leq 66.67ms \quad (5.1)$$

Only the personal computer could reach real time for video resolutions less than 1920 x 1080. The project aims to use an embedded low power device, but even Raspberry Pi 3 is not able to meet the timing requirements. For that reason a hardware acceleration is needed on the ZedBoard.

Having established the starting conditions, the parameter that is missing is the optimum number of frames to buffer in the RAM free zone. To find this parameter, some tests with the same video sequence and a different number (N) of frames buffered in memory must be carried out. The results are shown in Figure 5.8, wherein the blue curve represents the achieved frame rate and the latency is represented by the red curve; the principal vertical axis (on the left) shows the frames per second (FPS) speeds that the system can reach, the secondary vertical axis (on the right) shows the latency time in seconds and the horizontal axis shows the number of frames buffered in RAM. This simple experiment, which already uses the FPGA acceleration and a resolution of 480 x 480, makes it possible to choose the number of frames to buffer. In fact, the evolution of the blue curve shows that after six frames buffered in RAM, the system is able to surpass twenty FPS. The number of frames chosen was fifteen, which was the same frame rate of the input video, which represents a realistic compromise between memory usage and processing speed. The latency in this case represents the time that is needed by the system to process the video stream; in other words, how much time has passed since the first frames were introduced, processed and reached the output of the system. The latency observed was less than one second, which for all intensive purposes represents an acceptable value.

Figure 5.8: Evolution of the Number of frames to buffer in RAM.

5.4 Testing Environment in APM

The developed prototype embedded system has been physically validated in a real environment in APM. The requirements for a customised solution were developed in conjunction with the APM medical personnel. A very important consideration is that this thesis is dealing with Parkinson's patients in a medical area. Thus the sensors (in this case the camera sensors) and all the necessary equipment must be installed in a less invasive manner. Furthermore, the layout of the equipment must be tidy, no misplaced cables or objects that can obstruct the passage of patients or staff, or cause them any harm. Another important aspect is represented by the camera position, because it should be placed in a location that permits a wide field of vision. A strategic camera position allows for the monitoring of a larger area and the tracking of a greater number of people. For this reason, an accurate analysis about the day hospital center premises has also been realized, to establish where the electronic instrumentation should be installed for optimal performance.

a) Triangular Support

b) Installation View 1

c) Installation View 2

d) Installation View 3

Figure 5.9: Camera Installation in APM.

Figure 5.9 depicts the details of the camera installation, where a triangular support has been designed and produced to install the camera. An important consideration is to make the patients feel comfortable and avoid feeling observed, and thus the camera installation is discrete and attempts to blend in with the hospital setting as far as possible. As a consequence the patients behave normally.

Figure 5.10 shows the scene captured by the camera. As is possible to note, the camera is located in a high transit zone: the entrance corridor.

Figure 5.10: Camera live view.

This position is strategic because people frequently pass by this intersection, using the corridor to access the central areas of the hospital like the lunchroom and the bathrooms. All the rest of the equipment (server, Zedboard, etc) has been installed in a separate room not accessible to the public, respecting all the requirements for patients and medical personnel.

Proper equipment installation that takes into consideration the patients well-being first of all, resolves one of the most sensitive dilemmas in this project. It's imperative that the system works and monitors patients without disturbing them. Thanks to the proposed solution, it is possible to perform all necessary tests needed to verify the algorithms and the embedded system. The results are commented on in the next chapter.

Chapter 6

Results

This section presents the results obtained by our prototype system, including the FPGA background subtraction module. The experimental setup is similar to the one described in the previous section, processing the same video sequences in order to maintain the same complexity (number of people, light condition, etc). However, as each video sequence has a different resolution, the video resolution of all the sequences have been normalized to generate different sets of video sequences. The selected resolutions are those available to the camera. Furthermore, we use two video format color spaces: RGB and gray-scale. Even if the original version of the algorithm is RGB, the gray-scale is added to evaluate the speed differences compared to RGB, as the footprint in memory of gray-scale video sequence is lower, enabling more pixels to be stored in FPGA memory simultaneously.

Tables 6.1 and 6.2 show the execution time, in milliseconds, taken from different video sequences with different resolutions. Table 6.1 presents the results obtained using video sequences with a squared aspect ratio, while Table 6.2 presents the results with rectangular aspect ratio video sequences in order to compare them with state of the art systems.

Finally, Table 6.3 sums up the average timing performances of the different formats processed, aggregated by the color space used. In line with previous studies referenced above, the measurements are expressed in FPS. Table 6.3 has been converted from milliseconds to FPS to use the same unit of comparison depicted in Table 6.4.

Sequence	480×480 RGB (ms)	480×480 gray (ms)	512×512 RGB (ms)	512×512 gray (ms)	768×768 RGB (ms)	768×768 gray (ms)	1056×1056 RGB (ms)	1056×1056 gray (ms)
shop1.mp4	45.8	43.05	52.9	48.9	121.3	115.9	234.9	219.9
shop2.mp4	45.1	43.6	52.2	50.6	121.5	116.5	237.5	221.9
hosp1.mp4	47.4	44.6	53.5	49.05	123.9	123.9	242.2	228.7
hosp2.mp4	46.9	45.1	51.4	50	125.1	117.2	243.3	227.3
hosp3.mp4	47.2	44.6	52.5	51.8	124.3	124.3	245.8	229.9
hosp4.mp4	48.5	45.4	53.8	49.8	125.8	119.1	244.9	230.4
hosp5.mp4	46.5	43.8	53.5	50.5	124.2	117.9	242.9	229.8
lab1.mp4	44.6	43.3	50.9	49.2	121.9	115.2	233.2	222.2
lab2.mp4	44.9	43.2	50.3	48.8	118.7	115.4	232.1	219.9

Table 6.1: Time (in ms) to process the background subtraction on one frame depending on the video sequence and the resolution (square aspect ratio), using the ZedBoard (CPU+FPGA).

Sequence	160×120 RGB (ms)	160×120 gray (ms)	320×240 RGB (ms)	320×240 gray (ms)	640×480 RGB (ms)	640×480 gray (ms)
shop1.mp4	3.9	3.6	15.3	13.7	62.2	59.04
shop2.mp4	4.1	3.8	15.5	13.9	63.6	59.7

Table 6.2: Time (in ms) to process the background subtraction on one frame depending on the video sequence and the resolution (rectangle aspect ratio), using the ZedBoard (CPU+FPGA).

Sequence	160×120 FPS	320×240 FPS	480×480 FPS	512×512 FPS	640×480 FPS	768×768 FPS	1056×1056 FPS
RGB	250	64.93	21.58	19.11	15.9	8.13	4.17
Grayscale	277.8	72.46	22.7	20.07	16.84	8.45	4.43

Table 6.3: System average time measures in FPS, using the ZedBoard (CPU+FPGA).

Work	Hardware	Connectivity	Resolutions	Frame Rate	Power Consumption	Remarks
		IN/OUT		FPS	W	
Chen et al. 2017 [27]	Zynq 7020	Eth/NS	640×480	30	1.82	In/Out times not considered
Guo et al. 2017 [28]	Zynq 7020	PMOD/NS	320×240 resized to 160×120	89.2	2.99	Out system times not specified
Cocorullo et al. 2019 [29]	Zynq 7020 Virtex 6	NS	HD resized to 160×128 and 320×240	up to 74	NS	In/Out times not considered and not specified
Sajjanar et al. 2016 [31]	Zynq 7020	PMOD/VGA	640×480	81.57	NS	The chain process only the BGS
Zhang et al. 2018 [32]	Zynq 7020	NS	VOT2015 dataset	18.6	1.284	In/Out times not considered
Liu et al. 2015 [34]	Zynq 7020	NS/HDMI	1137×686	1.30	NS	In system times not specified
Cho et al. 2019 [37]	Virtex 5	NS/HDMI	640×480	30	NS	In system times not specified
Our Work	Zynq 7020	Eth/Eth	up to 1056×1056 see Table 6.3	up to 277.8 see Table 6.3	2.025 Gray 2.297 RGB	Complete chain analyzed, In/Out times considered

Table 6.4: Main characteristics of literature tracking systems and differences with related work. NS stands for not specified, Eth stands for Ethernet, BGS stands for background subtractor.

The first observation of note can be made about the timing results shown in Table 6.1, Table 6.2 and Table 6.3, where we see that processing an RGB sequence introduces just a little overhead when compared to processing a gray-scale sequence, about 6% slower. The difference in overhead is primarily due to the memory copy operation, as the amount of memory required is triplicated. On the contrary, the overhead in the HW processing pipeline of the FPGA is negligible, as the HLS creates a similar internal circuit that could perform the background subtraction on the image pixels, regardless the number of components per pixel (one for gray-scale and three for RGB).

Another important observation that emerges from the results is that the number of people in the video sequences has a lesser impact on the processing time of the CPU+FPGA version than on the CPU-only version. In fact, Table 5.2 shows just CPU times, and you can see that the time processing increases when there are more people, as is expected. On the contrary, Table 6.1 and Table 6.2, which show the CPU+FPGA solution, show a negligible difference amongst all sequences. The generated background subtractor HW module has a data independent execution time, which is lower due to the HLS optimizations that generate a processing pipeline to implement parallelism.

The processing acceleration of the ZedBoard (CPU+FPGA) compared to ZedBoard (only CPU) is between 3.4X and 3.8X faster, while the processing acceleration of the ZedBoard (CPU+FPGA) compared to Raspberry Pi 3 is between 1.6 and 2X faster. Compared with the Intel i7 CPU the ZedBoard (CPU+FPGA) processing times are closer, but are still slower than Intel CPU. However, the ZedBoard, or other boards with CPU+FPGA SoC are embedded systems, which have inherent advantages in terms of cost, power consumption (that is less than 10% of the i7) and space.

Table 6.4 compares the performance of our system with other related systems, which lightly outperform our system. However, there are two factors not considered in the table: our proposal is a complete HW/SW chain that stores the data in main memory for other modules to use it, and the background subtractor module is automatically generated from the SW implementation. Our solution is more flexible, and its modification or integration in a SW processing chain is simpler and faster. The performance of the proposed system, with the bottlenecks included, is valid for real time Parkinson's patient monitoring allowing its use in hospitals.

As this work is describing a real implementation that works in a biomedical context, the measure are evaluated according the complete chain, from the input data acquisition to the final output data submit, that includes the bottleneck and the optimizations.

The remarks column of the comparison gathered within Table 6.4, highlights the fact that the other works do not specify input/output latencies, that have considered because this is a real project that receives data in one format and submits the data in another format in order to respect patient privacy, a mandatory requirement. Thus the entire process has been measured, not only MoG FPGA performance. Moreover, the Multi-tasking implementation uses one thread to manage the input buffers and the FPGA; and the other thread for the tracking and the output which are done in parallel.

Figure 6.1: Process measurement comparison.

The image 6.1 gives a more visual explanation of the more complete measurements done by our work with respect to what is measured in other works, specifically that not all works perform the tracking. Also most of the other works resize the video stream while in this work the stream is taken as it is, and processed.

Table 6.5 compares only the performance of the background subtractor module, eliminating the effects of data input and output to main memory, as done in the related work. Our MoG hardware/software implementation achieves excellent performance marks.

Previous Works			Our Work			Remarks
Work	Real Resized Resolution	FPS	Resolution	FPS complete chain	FPS MoG only FPGA	
Chen et al. 2017 [27]	640×480	30	640×480	15.9(RGB) 16.84(Gray)	162	I/O data exchange associated times are not considered
Guo et al. 2017 [28]	160×120	89.2	160×120	250 (RGB) 277.8 (Gray)	2600	The Input is resized, the Output data exchange associated times are not specified
Cocorullo et al. 2019 [29]	160×128 and 320×240	74	160×120 and 320×240	250 (RGB) 277.8 (Gray) 64.93(RGB) 72.46 (Gray)	2438 650	The Input is resized, I/O data exchange associated times are not considered, the I/O interfaces are not specify
Sajjanar et al. 2016 [31]	640×480	81.57	640×480	15.9 (RGB) 16.84 (Gray)	162	The complete chain is specified, but no Tracking is performed, only MoG
Zhang et al. 2018 [32]	from 320×180 to 1280×720	18.6	320×240 and 1056×1056	64.93 (RGB) 72.46 (Gray) 4.17(RGB) 4.43 (Gray)	650 44.8	I/O data exchange associated times are not considered
Liu et al. 2015 [34]	1137×686	1.30	1056×1056	4.17 (RGB) 4.43 (Gray)	44.8	The Input data exchange associated times are not specified
Cho et al. 2019 [37]	640×480	30	640×480	15.9 (RGB) 16.84 (Gray)	162	Only FPGA implementation, the time associated to input interface is not specified

Table 6.5: FPGA mixture of Gaussian performance comparison.

Once the timing results are discussed, it is important to highlight another positive aspect: the reliability of the background subtractor implemented on the FPGA. This is important to analyze because of the approximations done, changing the variable types, as already mentioned and shown in Table 3.3. Figure 6.2 makes it possible to evaluate the results of the background subtractor module. It is arranged as follows: in the even rows the foreground masks are exposed, while in the odd rows the results of the tracking are shown. The images are also organized in columns where the gray-scale column represents the images processed with the FPGA using the gray-scale algorithm version, whereas the RGB column represents the images processed with the FPGA using the RGB algorithm version, and the reference column represents the images processed with the CPU using the original algorithm, to be used as a reference to compare and evaluate the FPGA processing capabilities. The analysis is done by capturing a frame for all the scenarios investigated (the hospital, the shop and the laboratory). It is important to note that in each scenario, the frame is the same for gray-scale, RGB and reference.

Evaluating the images, independently of the conversion from float to fixed, the people in the foreground masks are located exactly in the original position. Regarding their shapes, you can observe less sharpness in the figures, and in some zones the shape has been deleted. This could become an issue in the case that the Kalman tracker does not interpret the foreground correctly. In fact, in the Figure 6.2a and 6.2d the bounding box of the gray-scale algorithm has a minor area, and in Figure 6.2d there is a false positive, where in fact one person has been detected as two people. The same effect happens in Figure 6.2g, 6.2l and in Figure 6.2o, 6.2r. The RGB version displays in some cases a minor bounding box area, but does not generate false positives (detecting one person as two people), and in other cases, helps to reduce false negatives, as seen in Figures 6.2b, 6.2e, whereas in the reference images (Figure 6.2c and 6.2f) the tracking has marked the door, but there is no person at that position.

gray-scale

RGB

Reference

hosp1 Output Mask

a

b

c

hosp1 Tracking Result

d

e

f

shop1 Output Mask

g

h

i

shop1 Tracking Result

l

m

n

Figure 6.2: Images comparison (a–t).

After the image comparison, a more complete analysis of the results was done using two performance indicators—*precision* and *recall*. In this case, *precision* represents the probability that a detected pedestrian is relevant, while the *recall* indicator is the probability that a relevant pedestrian is detected in a scene. The two indicators are defined as follows:

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN} \quad (6.1)$$

Where:

- TP (true positive) represents the bounding box estimated by the algorithm where a human is located in the sequence at specific time t .

- *FP* (false positive) represents the bounding box estimated by the algorithm where a human is not located in the sequence at specific time t .
- *FN* (false negative) represents the bounding box not estimated by the algorithm where a human is located in the sequence at specific time t .

Tables 6.6 and 6.7 contain the respective results, the tracking precision and recall comparison for the algorithm, where i7 represents the personal computer and ZB stands for ZedBoard.

Finally, in Table 6.8 the FPGA resource utilization is depicted. As it is clear to observe, the RGB version of the algorithm needs more resources because each pixel is represented with three components, while the gray-scale is represented only with one component per pixel.

Sequence	TP%	FP%	FN%	TP%	FP%	FN%	TP%	FP%	FN%
	i7 RGB	i7 RGB	i7 RGB	ZB gray	ZB gray	ZB gray	ZB RGB	ZB RGB	ZB RGB
hosp1	81.78	9.98	4.23	48.94	38.80	12.26	72.01	18.13	9.86
hosp2	69.78	14.10	16.11	40.87	31.01	28.12	66	15	19
shop1	84.87	4.52	10.61	54.28	32.21	13.51	73.57	15.80	10.63
shop2	60.23	22.59	17.18	22.62	7.60	69.78	39.91	6.04	54.05
lab1	87.82	2.78	9.39	46.67	34.36	18.97	79.83	8.41	11.76
lab2	74.01	12.77	13.21	39.66	34.18	26.16	67.41	11.24	21.35

Table 6.6: Tracking algorithm results comparison (represented in percentages).

Sequence	Precision %	Recall %	Precision %	Recall %	Precision %	Recall %
	i7 RGB	i7 RGB	ZB gray	ZB gray	ZB RGB	ZB RGB
hosp1	89.58	93.22	55.78	79.97	79.89	87.95
hosp2	83.19	88.89	56.85	59.24	81.48	77.65
shop1	94.94	89.89	62.76	80.08	82.32	87.38
shop2	72.72	77.81	74.84	24.48	86.86	42.48
lab1	96.92	90.33	57.59	71.09	90.48	87.16
lab2	85.27	84.91	53.71	60.26	85.71	75.95

Table 6.7: Comparison of precision and recall for the tracking algorithm.

Resource	Available	Utilization RGB	Utilization % RGB	Utilization gray	Utilization % gray
LUT	53200	24170	45.43	6892	12.95
LUTRAM	17400	11589	66.60	328	1.89
FF	106400	11588	10.89	9397	8.83
BRAM	140	135	96.43	111	79.29
DSP	220	39	17.73	24	10.91
BUFG	32	1	3.13	1	3.13

Table 6.8: FPGA resources utilization (in number of elements).

Chapter 7

Conclusions

A complete tracking solution dedicated to biomedical purposes has been presented in this thesis. The system has been tested in several scenarios: a shop, a laboratory and a day hospital, specifically the APM Day Hospital center in Madrid where the system has been tested with Parkinson's patients. The goal of this study was to create a solution that permits real-time tracking of patients while guaranteeing their privacy protection. To achieve this objective, a hybrid solution that uses the combination of hardware and software customization is proposed. The system runs on the novel Zynq platform that incorporates a CPU and an FPGA on the same chip. Moreover, a deep integration between hardware and software resources has been adopted, which allows us to combine the software libraries needed to track people, identify the most demanding parts of the software chain, and take advantage of the FPGA to accelerate them. The system developed could grant real time processing acquiring data from a commercial Zenith camera (@ 15 FPS), starting from the base resolution (192×192) up to 640×480 . For applications that need higher resolutions, it is still possible to process in real time, but the Zenith camera should be configured to send a slower frame rate (8 FPS for 768×768 and 4 FPS for 1056×1056), as detailed in Table 6.3. This study also outlines the analysis of the complete system reliability and testing of the two hardware implementations (that use RGB and gray-scale color spaces respectively) and the analysis of the algorithm that involves type conversion from float to fixed. The results achieved are compared with the reference results obtained by processing the same test sequences with the original version of the algorithm as the ground-truth. As detailed in the previous section, the reader can observe that the RGB version of the algorithm obtains more accurate results,

although its efficiency is lower than that of the gray-scale version. This is confirmed in Table 6.6 and Table 6.7 which show that the RGB implementation is the closest to the reference, even better in some cases. The RGB version is therefore the solution chosen for this project because it represents the best compromise between processing speed and accuracy.

This thesis improves the state of art due to the thorough analysis performed during its evolution which is highlighted as follows:

- The analysis is more complete because it is realized considering RGB and Gray scale color spaces while other works only consider Gray scale, which also represents an improvement in terms of memory management.
- A consequence of the previous point, the precision of the proposed solutions is evaluated, comparing the results with the original algorithm, not implemented on the FPGA, used as the reference that hasn't been performed in other works of the state of art.
- The complete processing chain has been specified describing all the bottle necks and all the modules timing not always explicit in the other works of the state of art.
- This work perform a complete speed and timing comparison. The various phases developed includes pre-acceleration and post-acceleration of the different modules comparing the timing of both software and hardware implementation. Furthermore once the hardware acceleration is ready, a timing measurement is done analyzing the complete solution and only FPGA solution.
- This work has also been tested using a set of verification methodologies that examine the hardware zone, the software zone and the complete solution.
- This work is also unique because it has been tested in an real biomedical environment with actual Parkinson's patients and the medical personnel of APM.
- Another unique feature not mentioned in the state of art is represented by the solutions proposed to meet the demands of privacy for patients, which was one of the main problems resolved in this research project.

The system presented is developed on a ZedBoard, an embedded system that allows real time processing, and additionally, it is a low-power consumption system. In fact, the board power consumption is lower than 20 W which represents a great savings in energy consumption when compared to a PC that will always have a much higher power demand (minimum 300 W).

This study could be further improved in several aspects, focusing on applications that could generate short and long-term strategies.

From a short-term plan, this study could be improved as follows:

- A Microblaze soft CPU could be added to the FPGA to reduce the interaction between ARM CPU and FPGA. This would allow for the introduction of a part of the code, that in this moment is running on the ARM CPU, into the Microblaze to simplify the system relays between CPU and FPGA.
- Improve the Hardware and Software scheduling by simplifying the data exchanged.
- Another future development foreseen in this study is the porting of the entire system to a newer and faster Zynq board. The bottleneck in the ZedBoard is represented in the RAM that operates at 533 MHz and has a low bandwidth and low data transfer rate. The entire system could easily be ported to a higher performance Zynq board, which would enable a faster CPU and a higher RAM bandwidth with a higher data transfer rate. Thus, the system could deal with higher image resolutions, further improving the overall quality and efficiency.

Once the system is updated to a more powerful device with a larger memory capacity, in the long term, it would make the following improvements possible:

- Due to greater memory availability, it would be possible to use float type instead of fixed type. This could be a great improvement of precision, achieving results close to the original algorithm reference.
- Implement the Kalman filter on the FPGA improving the acceleration of the overall system. This also allows for a reduction in the RAM memory exchanges because the image could enter into the FPGA Background subtractor and instead of sending the output to the RAM, it could be introduced in the Kalman filter FPGA implementation. In this way the output of the system consists only of the coordinates of the people being tracked from the output of the Kalman filter.
- As a consequence of the previous point, reducing the memory exchanges, it could be possible to increase the FPGA internal component pipeline, and a new resource schedule should be implemented to optimize the entire system.

- Another research possibility is the integration of the chip with the camera; in fact, many cameras already do this, streamlining the ability to acquire and process the data in real time using FPGAs. It is possible to integrate our background subtraction IP directly in the FPGA that acquires the data to provide parallel processed streaming that could be analyzed with an external tracker using less resources.
- A further research option would be to construct the entire Zynq chip inside the camera. This would allow the central control server to receive already-processed data with the tracking information from the camera. This would represent a great improvement looked at from an energy efficiency aspect, because the single chip power demand of the entire application is only around 2 W. Additionally, a control server that previously could only handle a limited number of cameras, thanks to this solution would be able to control more cameras and cover a larger physical area. This also represents a great efficiency improvement in terms of energy savings.
- Another opportunity for future research would be to study a hybrid distributed processing architecture. This allows for local calculations of the cameras and the management of all the architecture from a central control server. This could improve the patients care improving the use of data generated in the local processing points, introducing additional control server features such as falls detection, patient pain deduced from facial expression recognition, Parkinson patients freezing detection, etc.

Bibliography

- [1] Technical Specifications, available online: <https://www.ni.com/es-es/innovations/white-papers/08/fpga-fundamentals.html>, accessed on end 2017.
- [2] Technical Specifications, available online: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>, accessed on end 2017.
- [3] Technical Specifications, available online: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, accessed on end 2017.
- [4] Technical Specifications, available online: <http://zedboard.org/product/zedboard>, accessed on end 2017.
- [5] Technical Specifications, available online: https://www.xilinx.com/html_docs/xilinx2019_1/SDK_Doc/SDK_concepts/concept_Xilinxsystemdebugger.html, accessed on summer 2018.
- [6] Technical Specifications, available online: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_3/pg020_axi_vdma.pdf, accessed on summer 2018.
- [7] H. J. Koch and H. L. Gmb, “Userspace i/o drivers in a realtime context,” in *The 13th Realtime Linux Workshop*, 2011.
- [8] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, “Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 174–187, 2015.

- [9] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference, DAC '15*, (New York, NY, USA), Association for Computing Machinery, 2015.
- [10] Technical Specifications, available online: <https://www.xilinx.com/applications/broadcast/cameras.html>, accessed on end 2019 .
- [11] Technical Specifications, available online: https://wiki.apertus.org/index.php/AXIOM_Beta/Camera_Structure, accessed on end 2019.
- [12] Technical Specifications, available online: <https://www.tattile.com/vision-solutions/s100-s50/>, accessed on end 2019.
- [13] A. Ibrahim, W. Simon, D. Doy, E. Pignat, F. Angiolini, M. Arditi, J. . Thiran, and G. De Micheli, "Single-fpga complete 3d and 2d medical ultrasound imager," in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1–6, Sep. 2017.
- [14] F. Alvarez, M. Popa, N. Vretos, A. Belmonte-Hernández, S. Asteriadis, V. Solachidis, T. Mariscal, D. Dotti, and P. Daras, "Multimodal monitoring of parkinson's and alzheimer's patients using the ict4life platform," in *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–6, Aug 2017.
- [15] F. Alvarez, M. Popa, V. Solachidis, G. Hernández-Peñaloza, A. Belmonte-Hernández, S. Asteriadis, N. Vretos, M. Quintana, T. Theodoridis, D. Dotti, and P. Daras, "Behavior analysis through multimodal sensing for care of parkinson's and alzheimer's patients," *IEEE MultiMedia*, vol. 25, pp. 14–25, Jan 2018.
- [16] A. Belmonte-Hernández, V. Solachidis, T. Theodoridis, G. Hernández-Peñaloza, G. Conti, N. Vretos, F. Álvarez, and P. Daras, "Person tracking association using multimodal systems," in *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–6, Aug 2017.
- [17] G. Hernández-Peñaloza, A. Belmonte-Hernández, M. Quintana, and F. Álvarez, "A multi-sensor fusion scheme to increase life autonomy of elderly people with cognitive problems," *IEEE Access*, vol. 6, pp. 12775–12789, 2018.
- [18] A. Belmonte-Hernández, G. Hernández-Peñaloza, F. Álvarez, and G. Conti, "Adaptive fingerprinting in multi-sensor fusion for accurate indoor tracking," *IEEE Sensors Journal*, vol. 17, pp. 4983–4998, Aug 2017.

- [19] Technical Specifications, available online: <http://www.vivotek.com/fe8180/#specifications>, accessed on end 2017.
- [20] G. Conti, M. Quintana, P. Malagón, and D. Jiménez, “An fpga based tracking implementation for parkinson’s patients,” *Sensors*, vol. 20, p. 3189, Jun 2020.
- [21] G. Conti, C. Kyrkou, T. Theocharides, G. Hernández-Peñaloza, and D. Jiménez, “A novel fpga-based test-bench framework for sdi stream verification,” *EURASIP Journal on Image and Video Processing*, vol. 2020, p. 31, Jul 2020.
- [22] A.-H. A. El-Shafie and S. E.-D. Habib, “A survey on hardware implementations of visual object trackers,” *IET Image Processing*, vol. 13, pp. 863–876, 2017.
- [23] B. Bulat and M. Kryjak, Tomaszand Gorgon, “Implementation of advanced foreground segmentation algorithms gmm, vibe and pbas in fpga and gpu – a comparison,” *Computer Vision and Graphics*, vol. 8671, pp. 124–131, 2014.
- [24] N. C. Rao and P. G. Hegde, “Design and implementation of robust object detection using ip core creation by background subtraction algorithm on fpga,” *IJETCSE*, vol. 14, pp. 369–372, 2015.
- [25] H. Tabkhi, M. Sabbagh, and G. Schirner, “An efficient architecture solution for low-power real-time background subtraction,” in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 218–225, July 2015.
- [26] Technical Specifications, available online: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, accessed on summer 2018.
- [27] W. Chen, Y. Ma, Z. Chai, M. Chen, and D. He, “An fpga-based real-time moving object tracking approach,” in *Algorithms and Architectures for Parallel Processing* (S. Ibrahim, K.-K. R. Choo, Z. Yan, and W. Pedrycz, eds.), (Cham), pp. 65–80, Springer International Publishing, 2017.
- [28] S. Guo, S. Pan, X. Li, L. Shi, P. Zhang, P. Guo, and Y. He, “A system on chip-based real-time tracking system for amphibious spherical robots,” *International Journal of Advanced Robotic Systems*, vol. 14, no. 4, pp. 1–19, 2017.
- [29] G. Cocorullo, P. Corsonello, F. Frustaci, L.-d.-l.-A. Guachi-Guachi, and S. Perri, “Multimodal background subtraction for high-performance embedded systems,” *Journal of Real-Time Image Processing*, vol. 16, pp. 1407–1423, Oct 2019.

- [30] D. Li, ed., *HSV Color Space*, pp. 793–793. Boston, MA: Springer US, 2008.
- [31] S. Sajjanar, S. K. Mankani, P. R. Dongrekar, N. S. Kumar, Mohana, and H. V. R. Aradhya, “Implementation of real time moving object detection and tracking on fpga for video surveillance applications,” in *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, pp. 289–295, Aug 2016.
- [32] B. Zhang, X. Li, J. Han, and X. Zeng, “Minitracker: A lightweight cnn-based system for visual object tracking on embedded device,” in *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, pp. 1–5, Nov 2018.
- [33] Technical Specifications, available online: <https://www.votchallenge.net/vot2015/dataset.html>, accessed on end 2017.
- [34] W. Liu, H. Chen, and L. Ma, “Moving object detection and tracking based on zynq fpga and arm soc,” in *IET International Radar Conference*, pp. 1–4, 01 2015.
- [35] M. Cheema, L. Lacassagne, and O. Hammami, “System-platforms-based systemc tlm design of image processing chains for embedded applications,” *EURASIP Journal on Embedded Systems*, vol. 2007, p. 071043, 01 2007.
- [36] R. Domingo, R. Salvador, H. Fabelo, D. Madroñal, S. Ortega, R. Lazcano, E. Juárez, G. Callicó, and C. Sanz, “High-level design using intel fpga opencl: A hyperspectral imaging spatial-spectral classifier,” in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, 2017.
- [37] J. Cho, Y. Jung, D.-S. Kim, S. Lee, and Y. Jung, “Moving object detection based on optical flow estimation and a gaussian mixture model for advanced driver assistance systems,” *Sensors*, vol. 19, no. 14, 2019.
- [38] Technical Specifications, available online: <http://zedboard.org/content/zedboard-0>, accessed on end 2017.
- [39] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, “A survey and evaluation of fpga high-level synthesis tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.
- [40] Technical Specifications, available online: <https://www.xilinx.com/products/design-tools/vivado.html>, accessed on end 2017.

- [41] Technical Specifications, available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug893-vivado-ide.pdf, accessed on end 2017.
- [42] Technical Specifications, available online: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>, accessed on summer 2018.
- [43] Technical Specifications, available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf, accessed on summer 2018.
- [44] Technical Specifications, available online: <https://www.xilinx.com/products/design-tools/embedded-software/sdk.html>, accessed on summer 2018.
- [45] Technical Specifications, available online: <https://opencv.org/>, accessed on summer 2018.
- [46] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, “A brief introduction to opencv,” in *2012 Proceedings of the 35th International Convention MIPRO*, pp. 1725–1730, 2012.
- [47] Z. Zivkovic and F. van der Heijden, “Recursive unsupervised learning of finite mixture models,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 651–656, May 2004.
- [48] Z. Zivkovic and F. van der Heijden, “Efficient adaptive density estimation per image pixel for the task of background subtraction,” *Pattern Recogn. Lett.*, vol. 27, p. 773–780, May 2006.
- [49] P. Power and J. A. Schoonees, “Understanding background mixture models for foreground segmentation,” 2002.
- [50] R. Mehra, “On the identification of variances and adaptive kalman filtering,” *IEEE Transactions on Automatic Control*, vol. 15, pp. 175–184, April 1970.
- [51] Technical Specifications, available online: https://docs.opencv.org/3.4.0/d7/dff/tutorial_how_to_use_OpenCV_parallel_for_.html, accessed on summer 2018.
- [52] G. Zhang, R. K. H. Ngan, and J. Y. B. Lee, “Emustream—an end-to-end platform for streaming video performance measurement,” *IEEE Access*, vol. 8, pp. 669–680, 2020.

- [53] Technical Specifications, available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_3/ug994-vivado-ip-subsystems.pdf, accessed on beginning 2018.
- [54] Technical Specifications, available online: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf, accessed on summer 2018.
- [55] Technical Specifications, available online: https://www.xilinx.com/support/documentation/ip_documentation/xilinx_com_ip_xlconcat/v2_1/pb041-xilinx-com-ip-xlconcat.pdf, accessed on summer 2018.
- [56] Technical Specifications, available online: https://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf, accessed on summer 2018.
- [57] Technical Specifications, available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug906-vivado-design-analysis.pdf, accessed on May 2018.
- [58] Technical Specifications, available online: <https://www.yoctoproject.org/>, accessed on end 2017.
- [59] Technical Specifications, available online: <https://www.xilinx.com/content/xilinx/en/products/design-tools/embedded-software/petalinux-sdk.html>, accessed on Q2 2018.
- [60] Technical Specifications, available online: <https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0016-petalinux-tools-hub.html>, accessed on Q2 2018.
- [61] Technical Specifications, available online: https://elinux.org/Device_Tree_Reference, accessed on Q2 2018.
- [62] T. Petazzoni, *Device Tree for Dummies*. 2013.
- [63] F. Reghenzani, G. Massari, and W. Fornaciari, “The real-time linux kernel: A survey on preempt_rt,” *ACM Comput. Surv.*, vol. 52, Feb. 2019.
- [64] Technical Specifications, available online: https://rt.wiki.kernel.org/index.php/Main_Page, accessed on Q2 2018.

- [65] Technical Specifications, available online: <https://www.denx.de/wiki/U-Boot/>, accessed on Q2 2018.
- [66] Technical Specifications, available online: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841973/Build+U-Boot>, accessed on Q2 2018.
- [67] Technical Specifications, available online: <https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html>, accessed on September 2018.
- [68] D. Aarno and J. Engblom, “Chapter 7 - dma: A concrete modeling example,” in *Software and System Development using Virtual Platforms* (D. Aarno and J. Engblom, eds.), pp. 211 – 236, Boston: Morgan Kaufmann, 2015.
- [69] M. Gorman, *Understanding the Linux Virtual Memory Manager*. USA: Prentice Hall PTR, 2004.
- [70] G. K.-H. Jonathan Corbet, Alessandro Rubini, *Linux Device Drivers, 3rd Edition*. O’Reilly Media, Inc., 2005.
- [71] Technical Specifications, available online: https://www.xilinx.com/support/documentation/application_notes/xapp742-axi-vdma-reference-design.pdf, accessed on summer 2018.
- [72] Technical Specifications, available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug871-vivado-high-level-synthesis-tutorial.pdf, accessed on September 2018.
- [73] Technical Specifications, available online: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841873/Linux+Drivers>, accessed on September 2018.
- [74] I. A. Dhotre, *Operating Systems*. O’Reilly Media, Inc., 2009.
- [75] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. USA: Kluwer Academic Publishers, 2003.
- [76] S. Singh and C. J. Lillieroth, “Formal verification of reconfigurable cores,” in *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00375)*, pp. 25–32, 1999.
- [77] R. Salemi, *FPGA Simulation: A Complete Step-By-Step Guide*. USA: Boston Light Press, 2009.

- [78] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, “A fast, flexible, and easy-to-develop fpga-based fault injection technique,” *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000 – 1008, 2014.
- [79] H. Angepat, G. Eads, C. Craik, and D. Chiou, “Nifd: Non-intrusive fpga debugger – debugging fpga ‘threads’ for rapid hw/sw systems prototyping,” in *2010 International Conference on Field Programmable Logic and Applications*, pp. 356–359, 2010.
- [80] A. Kourfali, D. M. Codinachs, and D. Stroobandt, “Superimposed in-circuit fault mitigation for dynamically reconfigurable fpgas,” in *2017 17th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pp. 1–6, 2017.
- [81] R. Salemi, *The Uvm Primer: A Step-By-Step Introduction to the Universal Verification Methodology*. Boston Light Press, 2013.
- [82] P. Graham, B. Nelson, and B. Hutchings, “Instrumenting bitstreams for debugging fpga circuits,” in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’01)*, pp. 41–50, 2001.
- [83] Technical Specifications, available online: <https://projects.digilentinc.com/adam-taylor/using-hls-on-an-fpga-based-image-processing-platform-8f029f>, accessed on January 2019.
- [84] K. Swaminathan, G. Lakshminarayanan, and S. Ko, “High speed generic network interface for network on chip using ping pong buffers,” in *2012 International Symposium on Electronic System Design (ISED)*, pp. 72–76, 2012.
- [85] Technical Specifications, available online: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>, accessed on end 2017.