People’s Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University M’Hamed BOUGARA – Boumerdès

Institute of Electrical and Electronic Engineering

Department of Electronics

Final Year Project Report Presented in Partial Fulfilment of
The Requirements for the Degree of

MASTER

In **Electronics**

Option: **Computer Engineering**

Title:

Performance Evaluation of OpenCV-based Applications on the Xilinx ZedBoard

Presented by:

- **ZIDELMAL Cherif**
- **DAHOUMANE Brahim**

Supervisor:

Dr. MAACHE Ahmed

Registration Number:……./2018

Dedication

I would like to dedicate my humble effort to my parents AbdelMadjid and Zina, my twin sisters, my grandparents, my uncles and aunts, to my friends Mokrane Meziane and Hadjab Farid without forgetting my partner Zidelmal Cherif, and my high school teacher Mrs. Akroum Hassiba.

Dahoumane Brahim

I would like to dedicate this work to my dear parents, to my brothers and sister, to my grandmother, aunts, uncles and to all their family members, to my friends, to my beloved Lily, and for all people who helped me through my academic journey.

Zidelmal Cherif

Acknowledgements

Ultimate Thanks and Praise to Allah.

As we present the results of our master project, we would like to thank the many people who made this success possible.

To begin with, we would like to express my heartfelt gratitude to Dr. Ahmed MAACHE for providing us with the opportunity to contribute towards the research project. The frequent suggestions and reviews provided by him were critical for the success of this dissertation. He took keen interest in our work and motivated us all along to meet the goals of this project. We feel lucky to have been supervised by him. Without his invaluable guidance and direction, we would have been surely lost.

We would like also thank the system administrators of the Institute for providing all the necessary infrastructure for the project expeditiously. We are also thankful to all the staff members of the institute for making our project at the institute an enjoyable experience.

Finally, I would like to thank all my friends and colleagues who were with us during all the ups and downs of the Master project. Going through all of this without their presence would have been difficult.

Abstract

This project report presents the performance evaluation of executing a number of OpenCV (Open Computer Vision) algorithms on the Xilinx ZedBoard. The latter contains Zynq-7000 FPGA, a System on a Chip (SoC) from Xilinx, which integrates dual-core ARM Cortex A9. The main goal of the project is to compare the execution times of a number of OpenCV-based image processing applications on the ZedBoard running a Linux kernel with the execution times of the same applications on a Personal Computer running Linux operating system.

The OpenCV library was selected for evaluation in this context due to its popularity in computer-vision based embedded systems and its availability for Zynq-7000 devices. The version of OpenCV used in this study is purely a software implementation running on a Linux kernel without any hardware acceleration.

The comparison results showed that, on average, the PC runs the selected applications 7 times faster than the Zedboard. The latter achieved an average image processing speed of approximately 5 fps, which is sufficient for a number of real-time computer vision applications. These results also suggest that xfOpenCV will provide better performance (i.e. faster execution time) given the hardware acceleration.

Contents

List of Figures

List of Tables

Nomenclature

ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FF	Flip-Flop
FPS	Frame Per Second
FPGA	Field-Programmable Gate Array
IDE	Integrated Development Environment
I/O	Input Output
JTAG	Joint Test Action Group
LUT	Look-Up Table
OS	Operating system
OTG	ON-The-Go
RAM	Random Access Memory
RGB	Red Green Blue
RTL	Register transfer language
SoPC	System on Programmable chip
UART	Universal Asynchronous Receiver Transmitter
VDMA	Video Direct Memory Access

Introduction

Motivation

Ever since their invention, reconfigurable architectures have been evolving continuously. Their popularity can be associated to the fact that they combine the flexibility of software with the performance of hardware. Field Programmable Gate Arrays (FPGAs) are perhaps the most popular and widely used class of reconfigurable architectures. With the research community devoting significant resources for developing these architectures, it is hardly surprising that they are growing larger and becoming faster by the day. Xilinx Inc. and Intel FPGA, world's leading vendors of FPGAs, have been introducing larger and faster products with every passing generation along with additional features which enhance their usability.
The main aim of this project is to evaluate the utilization of Xilinx ZedBoard FPGA in real-time computer-vision based on OpenCV library. A performance study is to be conducted where the execution times of ten computer-vision algorithms on this Board are compared to their counterpart on a PC.
The comparison will be performed between Xilinx Zedboard which contains a Dual-core ARM PS operating at up to 1GHz, and a Computer with an Intel Core i7 Processor operating up to 2.6GHz.

Goals

With this master project, we aim to:

- Implement and test OpenCV-based example applications on the ZedBoard platform.
- Conduct a performance evaluation of these implemented example applications by comparing their execution times to a PC.
- Assess the suitability of the Zedboard platform for real-time video processing based on OpenCV

- Ultimately, implement a real-time computer-vision platform using the hardware accelerated OpenCV library on the ZedBoard. In this platform, live images are read from a camera, processed inside the FPGA and the results are shown on a monitor.

Report Organisation

The Organisation of the remainder of this thesis is presented in this section. Chapter 1 gives a general overview of Open Source Computer Vision Library and general overview about Computer Vision.

Chapter 2 explores the different tools used in this projects starting with the introduction of the hardware and then software tools.

Chapter 3 covers the practical parts and implementations done in this project. These include the Hardware design, the Boot procedure of Linux on the ZedBoard and the steps to build and execute algorithms in the Xilinx SDK. We conclude the chapter with descriptions of the tested OpenCV Algorithms.

Chapter 4 outlines the performance evaluation results in addition to discussion.

Finally, conclusions and future work are presented.

Chapter 1

Background

1.1 Computers and Vision

1.1.1 Computer Vision

Computer Vision is a broad field that seeks to enhance and simplify processing of understanding images and videos easier for the computer. It deals with modeling and replicating human vision using computer software and hardware.

Computer Vision overlaps significantly with the following fields:

- **Image Processing** − It focuses on image manipulation.
- **Pattern Recognition** − It explains various techniques to classify patterns.
- **Photogrammetry** − It is concerned with obtaining accurate measurements from images.

1.1.2 Computer Vision Vs Image Processing

Image processing is the field that deals with image-to-image transformation where the input and output are both images. However, Computer vision is the construction of explicit, meaningful descriptions of physical objects from their image. The output of computer vision is a description or an interpretation of structures in 3D scene.

1.1.3 Applications of Computer Vision

Here we have listed down some of major domains where Computer Vision is heavily used.

Robotics

- Localization − Determine robot location automatically
- Navigation
- Obstacles avoidance
- Assembly (peg-in-hole, welding, painting)
- Manipulation (e.g. PUMA robot manipulator)
- Human Robot Interaction (HRI) − Intelligent robotics to interact with and serve people

Medicine

- Classification and detection (e.g. lesion or cells classification and tumor detection)
- 2D/3D segmentation
- 3D human organ reconstruction (MRI or ultrasound)
- Vision-guided robotics surgery

Industrial Automation

- Industrial inspection (defect detection)
- Assembly
- Barcode and package label reading
- Object sorting
- Document understanding (e.g. OCR)

Security

- Biometrics (iris, finger print, face recognition)

- Surveillance − Detecting certain suspicious activities or behaviors

Transportation

- Autonomous vehicle

- Safety, e.g., driver vigilance monitoring

1.2 Open Source Computer Vision Library

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. The library is used extensively in companies, research groups and by governmental bodies.

OpenCV contains different modules that are dedicated to different areas of computer vision. The images are stored in matrices defined by the Mat class, and one very helpful feature of Mat is its dynamic memory management, which allocates and deallocates memory automatically when image data is loaded or goes out of scope respectively. Also, if an image is assigned to other variables, the memory content stays the same, as the new variables get the reference passed on to them. This can save resources since algorithms often process smaller parts of the image at a time, and the data need not be copied every time. This requires OpenCV to keep count of how many variables are using the same block of memory as to not deallocate it the moment one of the variables are deleted.

Features of OpenCV Library

Using OpenCV library, you can

- Read and write images
- Capture and save videos
- Process images (filter, transform)
- Perform feature detection
- Detect specific objects such as faces, eyes, cars, in the videos or images.
- Analyze the video, i.e., estimate the motion in it, subtract the background, and track objects in it.

1.3 Xilinx Fast OpenCV Library

OpenCV library is available for several processor platforms including the Zedboard. The version evaluated in this project is a pure software implementation compiled for the ARM Cortex A9 processor running on a Linux Kernel.

In September 2017, Xilinx released a newer version of OpenCV library called xfOpenCV (Xilinx Fast OpenCV) which is a set of 50+ kernels, optimized for Xilinx FPGAs and SoCs, based on the OpenCV computer vision library. The kernels in the xfOpenCV library are optimized and supported in the Xilinx SDx Tool Suite [1]. The main improvement in this version is that it contains hardware acceleration blocks (functions) which subsequently boost performance. The following figure shows the architecture of a computer vision system based on xfOpenCV Kernel [1].

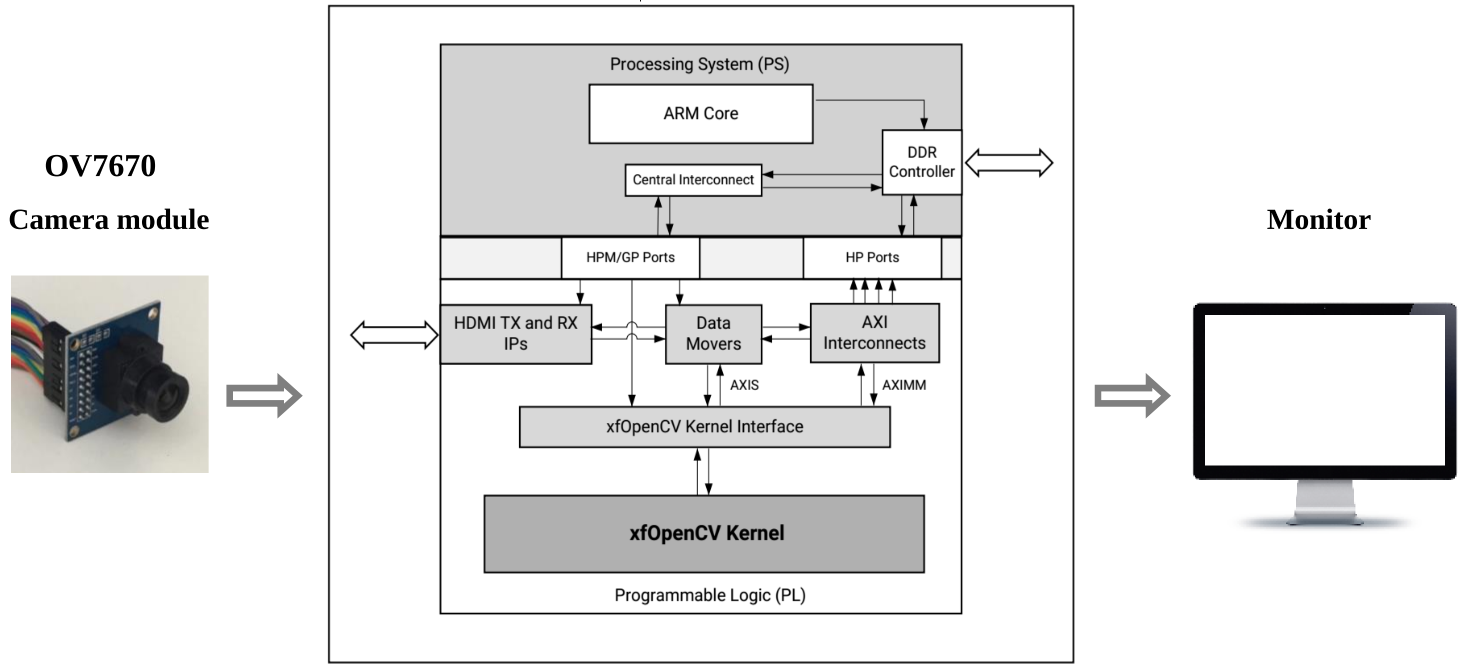
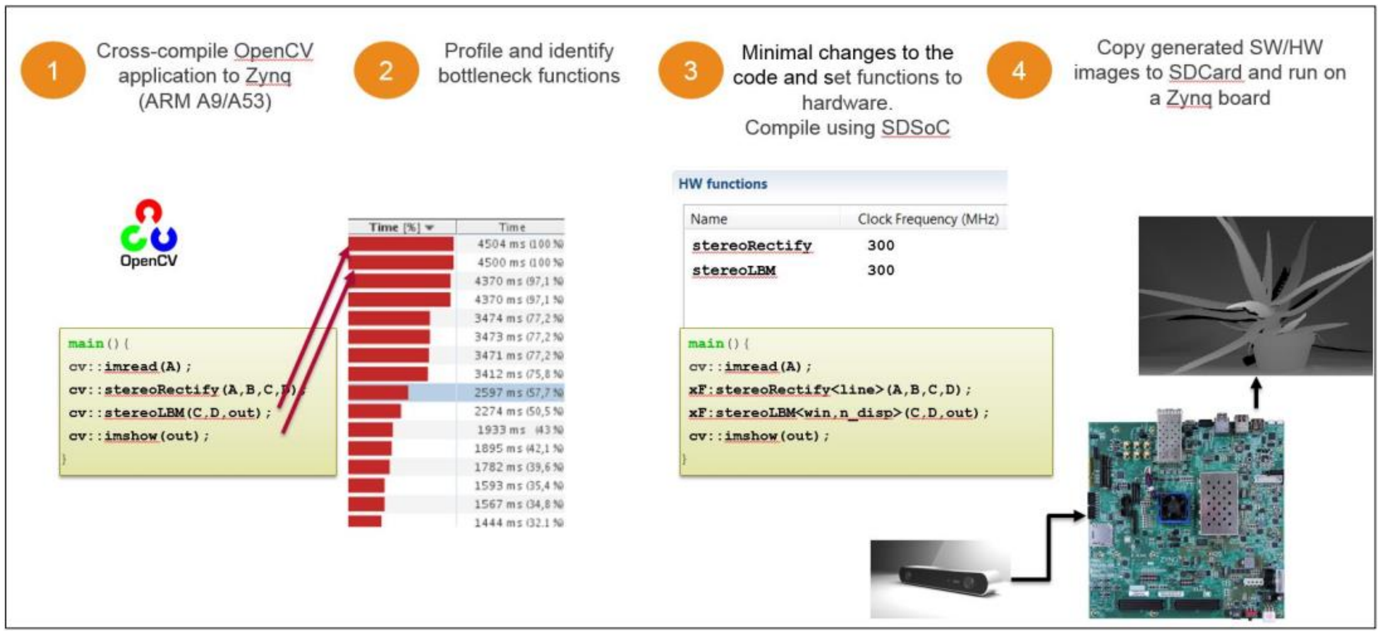


Figure 1. 1: The architecture of computer vision system based on xfOpenCV kernel [1]

The figure below demonstrates a typical design flow for accelerating OpenCV applications using the hardware functions available in the xfOpenCV library. The main idea is to replace the OpenCV functions that take the longest time to execute with their xfOpenCV counterpart (hardware functions) [1]. In this project, the first and the second steps were successfully accomplished. The remaining two steps are suggested as future work.


1
Cross-compile OpenCV application to Zynq (ARM A9/A53)
2
Profile and identify bottleneck functions
3
Minimal changes to the code and set functions to hardware.
Compile using SDSoC
4
Copy generated SW/HW images to SDCard and run on a Zynq board
OpenCV
HW functions

```
main() {
cv::imread(A);
cv::stereoRectify(A,B,C,D);
cv::stereoLBM(C,D,out);
cv::imshow(out);
}
```

Time [%]	Time
	4504 ms (100 %)
	4500 ms (100 %)
	4370 ms (97,1 %)
	4370 ms (97,1 %)
	3474 ms (77,2 %)
	3473 ms (77,2 %)
	3471 ms (77,2 %)
	3412 ms (75,8 %)
	2597 ms (57,7 %)
	2274 ms (50,5 %)
	1933 ms (43 %)
	1895 ms (42,1 %)
	1782 ms (39,6 %)
	1593 ms (35,4 %)
	1567 ms (34,8 %)
	1444 ms (32.1 %)

Name	Clock Frequency (MHz)
stereoRectify	300
stereoLBM	300

```
main() {
cv::imread(A);
xF:stereoRectify<line>(A,B,C,D);
xF:stereoLBM<win,n_disp>(C,D,out);
cv::imshow(out);
}
```

Figure 1. 2: The process of accelerating OpenCV algorithms using xfOpenCV library [1]

Chapter 2

Hardware and Software Tools

In this chapter, the hardware and software tools used in the performance evaluation are introduced.

2.1 Hardware:

2.1.1 FPGA

Conventional microchips and Central Processing Units (CPUs) are static and their internal structure cannot be changed. They are specifically manufactured to process software as efficient and fast as possible, but always one single instruction at a time. An FPGA on the other hand, contains many Configurable Logic Blocks (CLBs) that can be connected to each other in different ways as defined by a HDL. This gives the freedom to process many input data in parallel with the possibility of speeding up computation time by orders of magnitude. If the architecture is faulty or outdated, it can simply be reprogrammed, while a static chip such as an Application-Specific Integrated Circuit (ASIC) would have to be replaced completely.

Resource Type	Description
Flip-Flop (FF)	A small element of gates able to store one data bit between cycles
Look-Up Table (LUT)	An N-bit table of pre-defined responses for each unique set of inputs
Digital Signal Processor (DSP)	A block with built-in computational units such as adder, Substractor and
Block RAM (BRAM)	A block of dual port RAM memory very close to the FPGA fabric

Table 2. 1: Some of the logical block types in the FPGA

The internals of an FPGA are made up of many instances of a couple of logic units such as multiplexers, Flip-Flops (FFs), Look-Up Tables (LUTs), and memory blocks called BRAM. There are also Digital Signal Processors (DSPs), often in the hundreds, which can assist in floating point operations. The HDL provides a way for the user to specify the behavior of the system. The code is then used to synthesize an actual digital circuit that is implemented for the FPGA hardware to determine which of its resources are connected to which.

2.1.2 ZedBoard

ZedBoard is a low-cost development board for enabling hardware and software developers to create or evaluate Zynq SoC designs. It supports many target applications that include video processing, software acceleration, motor control, Linux/Android/RTOS development, embedded ARM processing, and general Zynq-7000 AP SoC prototyping. It is chosen for this project because real-time image processing can take advantage of its onboard Xilinx Zynq-7020 All Programmable SoC and its capability for a comprehensive set of port expansions. Figure 4.2 depicts the ZedBoard evaluation board.

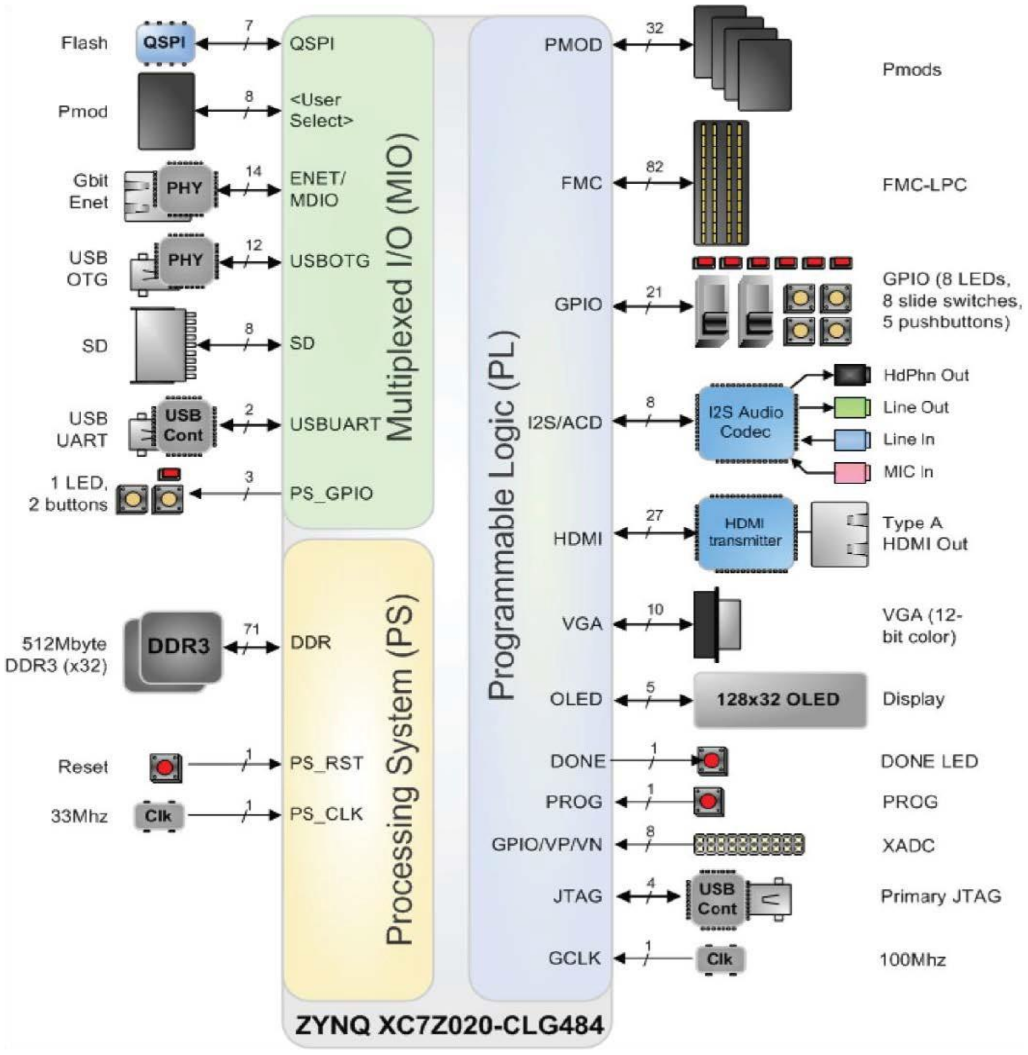


Figure 2. 1:*ZedBoard Hardware Block Diagram*

Following is the most important interface features that ZedBoard support:

- Zynq-7000 All Programmable SoCXC7Z020-CLG484-1
- Memory: 512 MB DDR3, 256 Mb Quad-SPI Flash, 4 GB SD card
- Onboard USB-JTAG Programming
- 10/100/1000 Ethernet
- USB OTG 2.0 and USB-UART
- PS & PL I/O expansion (FMC, Pmod Compatible, XADC)
- Multiple displays (1080p HDMI, 8-bit VGA, 128 x 32 OLED)
- I2S Audio CODEC

The Zynq 7020 device is composed of a dual ARM Cortex-A9 CPU based PS (Processing System) as well as Xilinx's hardware PL (Programmable Logic), with both on the same chip. The ARM cores are the Cortex A9 MPcore which can run at a peak frequency of 1 GHz. The PS also includes Level-2 cache and enables embedded computing capability by using DDR2 and DDR3 SDRAM memory, Flash memory, Gigabit Ethernet, general purpose I/O, and UART technologies. The Xilinx AXI Interface also provides high-speed memory-access between the PS and the PL.

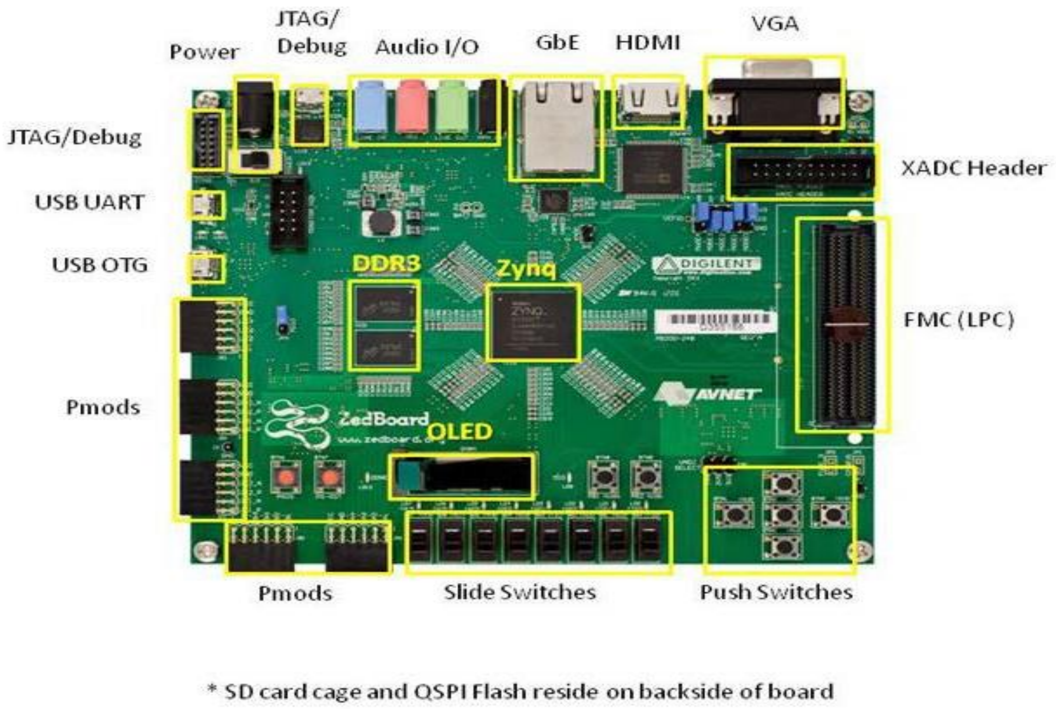


Figure 2. 2: Top side of the Xilinx ZedBoard platform

2.1.3 Zynq7000

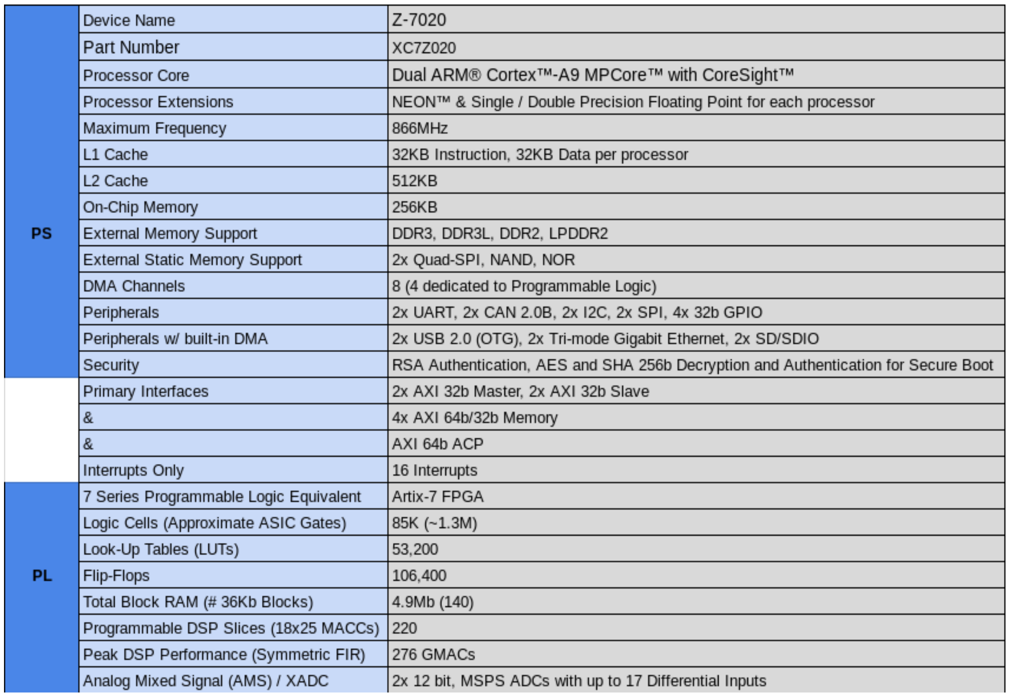
The FPGA device used in this project is Zynq 7020 (*XC7Z020)*. Figure 2.3 show the architectural parameters of the specific Zynq device.

The main advantage of this device is the high bandwidth of AXI4 communication ports available between the FPGA and the embedded processors.

ARM (PS)

There are two embedded ARM Cortex-A9 dual-core PS operating at up to 1 GHz and supporting cache hierarchy, memory controllers and I/O peripherals. It represents a complete programmable embedded platform that is fit for use without any FPGA programming. The two ARM processor cores come with a single instruction multiple data (SIMD) extension called NEON that provides a 128-bit-wide data path.

Figure 2.4 shows how PL, PS and peripherals are organized inside the Zynq FPGAs. The PS of Zynq contains a dual-core A9 processor with a set of peripheral connectivity interfaces and interconnection between PL and PS sections. PL operates as FPGA and it uses the AXI bus to communicate with ARM processor.



	Device Name	Z-7020
	Part Number	XC7Z020
	Processor Core	Dual ARM® Cortex™-A9 MPCore™ with CoreSight™
	Processor Extensions	NEON™ & Single / Double Precision Floating Point for each processor
	Maximum Frequency	866MHz
	L1 Cache	32KB Instruction, 32KB Data per processor
	L2 Cache	512KB
	On-Chip Memory	256KB
PS	External Memory Support	DDR3, DDR3L, DDR2, LPDDR2
	External Static Memory Support	2x Quad-SPI, NAND, NOR
	DMA Channels	8 (4 dedicated to Programmable Logic)
	Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
	Peripherals w/ built-in DMA	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO
	Security	RSA Authentication, AES and SHA 256b Decryption and Authentication for Secure Boot
	Primary Interfaces	2x AXI 32b Master, 2x AXI 32b Slave
	&	4x AXI 64b/32b Memory
	&	AXI 64b ACP
	Interrupts Only	16 Interrupts
	7 Series Programmable Logic Equivalent	Artix-7 FPGA
	Logic Cells (Approximate ASIC Gates)	85K (~1.3M)
	Look-Up Tables (LUTs)	53,200
PL	Flip-Flops	106,400
	Total Block RAM (# 36Kb Blocks)	4.9Mb (140)
	Programmable DSP Slices (18x25 MACCs)	220
	Peak DSP Performance (Symmetric FIR)	276 GMACs
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs

Figure 2. 3: Zynq-7020 All Programmable SoC Overview

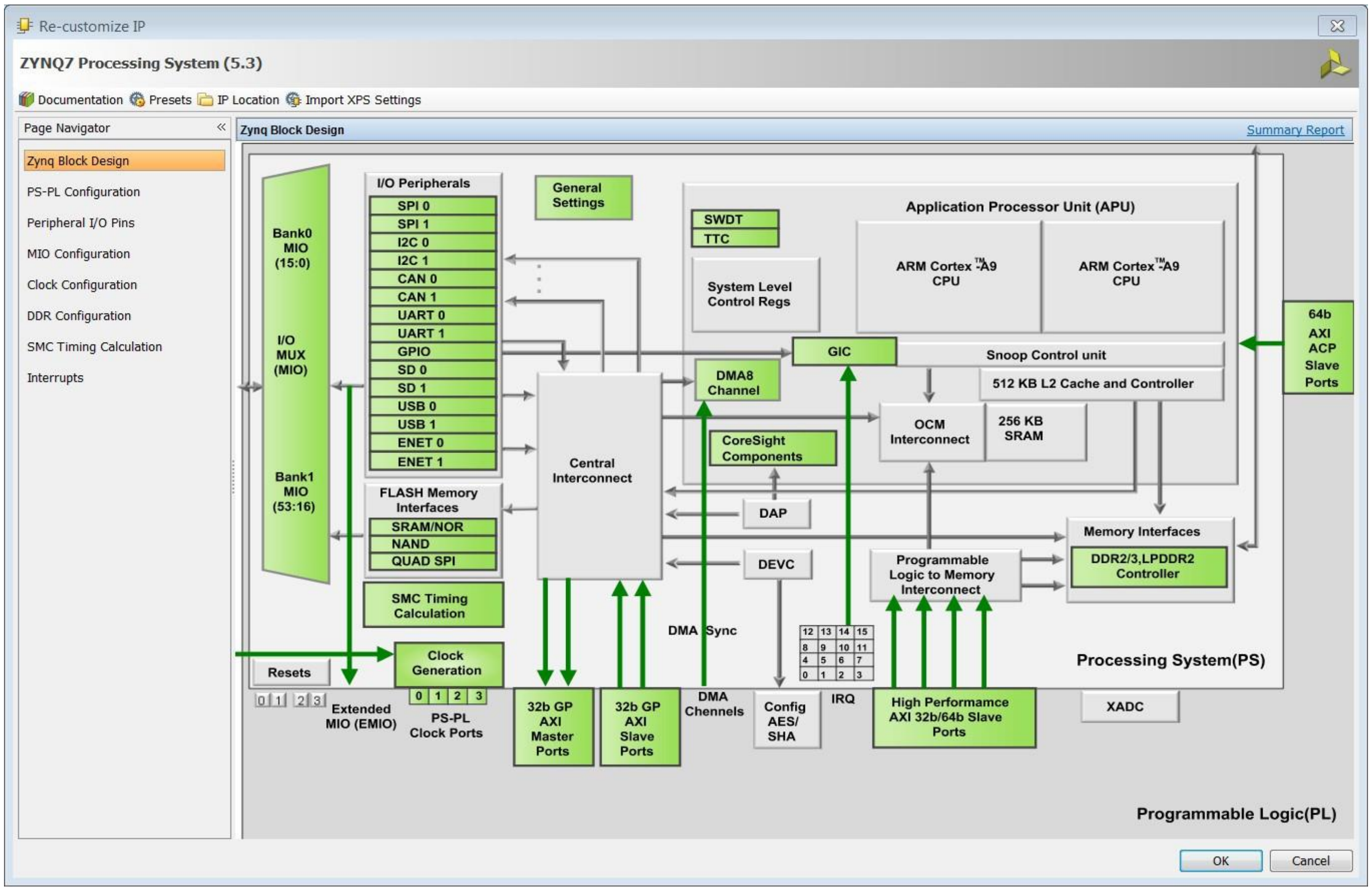


Figure 2. 4: Processing System Re-customize IP view

FPGA (PL)

FPGA (PL) is the Second major part of a hybrid computing platform in the Zynq SoC. FPGAs includes a set of reconfigurable resources that can be used to implement the individual function. The main resources parts of PL consist of Configurable Logic Blocks (CLB)'s, Block Ram (BRAM) Block, input-output blocks (IOBs), and Digital Signal Processing (DSP) slices. Figure 2.5 shows a high-level overview of Xilinx FPGA structure.

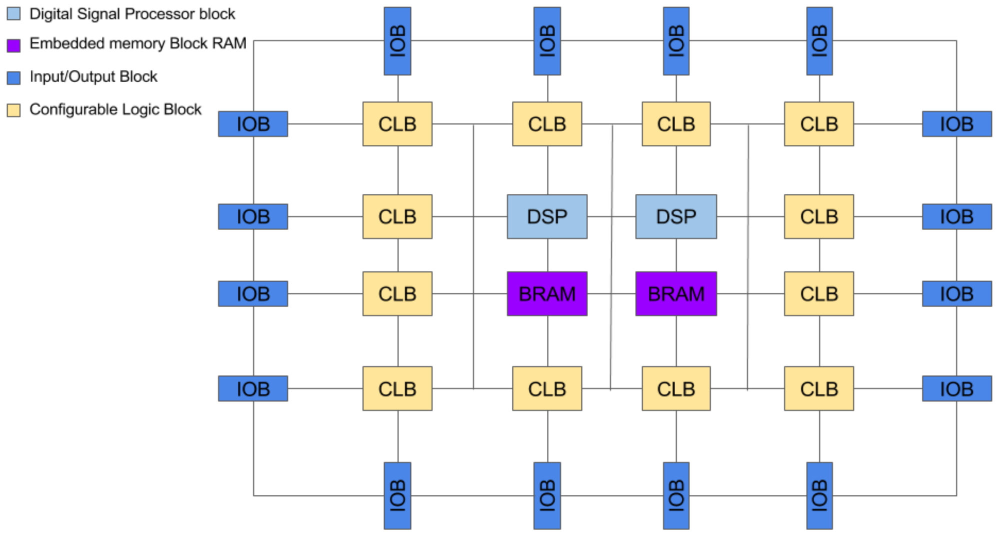


Figure 2. 5: Reconfigurable resources available overview on Xilinx FPGA

2.2 Software

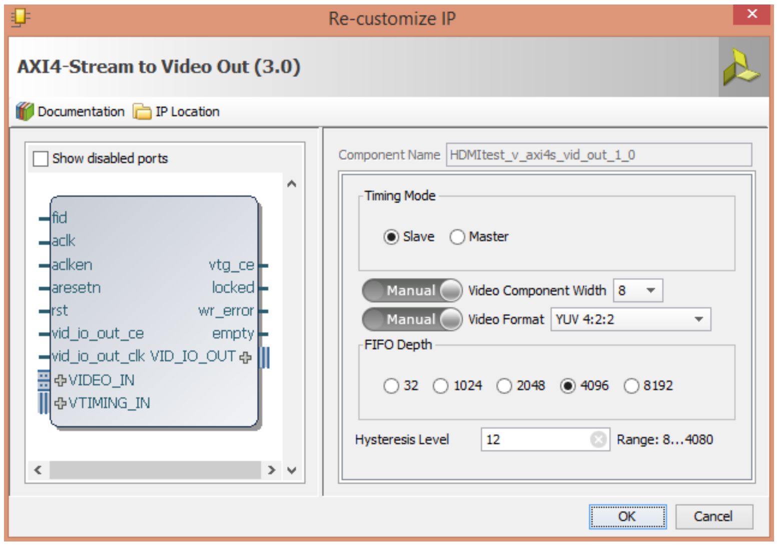
2.2.1 Vivado

Vivado is a new design program from Xilinx first released 2012. It is used to create the whole design of the RTL components to be synthesized and implemented on the chip. Note that this is not the same program as their HLST which is called Vivado HLS.

IP Blocks Vivado allows a user to create a design without forcing them to work with individual VHDL/Verilog code files, but instead uses complete IP Blocks that appears as boxes with pins for inputs and outputs used in a drag and drop fashion in the IP Integrator feature. This gives a good overview of the whole design and at the same time lets the user connect signals, buses and clocks without writing any code Most IP blocks can also be customized to some extent through a GUI accessed by double clicking the IP block in the IP Integrator. Depending on the IP there are different settings available, such as bit width of AXI4 stream, number of slave and/or master connections, color mode (i.e. YUV or RGB style etc.), timing mode and FIFO buffer depth to name a few.

A component generated in Vivado HLS would typically be one of those IP-Blocks. Vivado also comes with a library of a number of IP-Blocks from Xilinx ready to use, such as the Video Direct Memory Access (VDMA), Test Pattern Generator (TPG), and the essential PS IP used to interface any FPGA design with the PS of the Zynq. It is through the PS IP that the VDMA IPs access the external DDR memory using up to four High Performance (HP) ports. It is also where the clocks and other I/O are defined.

PS/PL Interface Many IP Blocks can communicate with the PS, as well as each other, through a bus interface called Advanced extensible Interface (which comes from the ARMAMBA protocol. AXI communication can be done in three different ways. An IP Block’s behavior and general settings are often controlled through an AXI4-Lite bus, as it is suited for low-throughput memory-mapped data transfers (i.e. reading/writing status registers).



Show disabled ports

Component Name: HDMItest_v_axi4s_vid_out_1_0

Timing Mode: [✓ Slave] Master

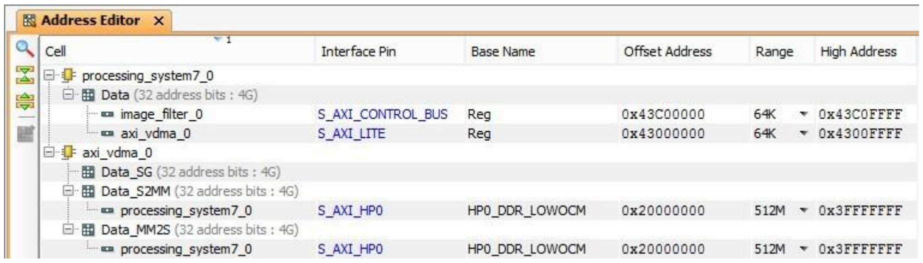
Manual — Video Component Width: 8

Manual — Video Format: YUV 4:2:2

FIFO Depth: 32 1024 2048 [✓ 4096] 8192

Hysteresis Level: 12 Range: 8...4080

Figure 2. 6: Customizable settings of the IP" AXI-4 Stream to Video Out.


Address Editor

Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
image_filter_0	S_AXI_CONTROL_BUS	Reg	0x43C00000	64K	0x43C0FFFF
axi_vdma_0	S_AXI_LITE	Reg	0x43000000	64K	0x4300FFFF
axi_vdma_0					
Data_SG (32 address bits : 4G)					
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x20000000	512M	0x3FFFFFFF
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x20000000	512M	0x3FFFFFFF

Figure 2. 7: The Address Editor, where each IP gets an address space defined for AXI communication.

Sending and/or receiving bigger data, such as an image, is often done through a standard AXI4 interface, which is more suited for high-performance memory- mapped operations. The third interface is AXI4-Stream which is used to continuously send a lot of data in high speed, for example streaming video.

Address space All the IP Blocks with an AXI4-Lite interface utilize the shared memory between the PS and PL to communicate. This means each such IP Block needs an address space which is assigned in Vivado before synthesis. Standard AXI4 bus data transfer is usually done through some sort of Direct Memory Access (DMA) which then feeds the consumer blocks. Control buses through AXI4-Lite normally gets a small 64 kB space for simple register I/O, while the AXI4 and AXI4-Stream interface can go up to gigabyte levels if needed. This address space can then be used from programs running on the PS to configure IP settings, or transfer images/data from/to the PL.

When the bitstream is finally generated after synthesis and implementation, Vivado can export a hardware specification file containing all addresses used in the design. This specification can then be imported and used in the SDK.

2.2.2 Xilinx SDK

The Xilinx SDK is used to write C/C++ drivers and programs to control the behavior of the FPGA. The hardware specification imported from Vivado is used to make a hardware platform project used to initiate the board. A Board Support Package (BSP) can then be created if Xilinx’s standalone Operating System (OS) is used, containing drivers to devices on the board such as I2C and USB etc.

Finally, a user can create their custom Application Project for their purposes. In this thesis, Linux was used as OS Platform, thus limiting the use of the BSP to mainly retrieving the AXI4 interface addresses. The SDK compiles the Application Project to *.elf* files which is then executed on the ARM processor in the Zynq.

2.2.3 Linux

Linux is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX. Linux is a Unix-like, open source and community-developed operating system for computers, servers, mainframes, mobile devices and embedded devices. It is supported on almost every major computer platform including x86, ARM and SPARC, making it one of the most widely supported operating systems.

Chapter 3

Experimental Setup

This chapter explains the experimental setup used to perform the performance evaluation of OpenCV applications. This includes installation of necessary drivers and software. We also provide the hardware/software setup of the target board used in implementation.

First, we give details about the host PC, and then we move to the boot procedure of Linux on Zedboard. We end up with general design flow by adding the required IP blocks using the IP integrator tool and build the Hardware Design, then running design Synthesis and Implementation and generate bit file. Finally, this Hardware Design, including the generated bit stream file, is exported to SDK tool.

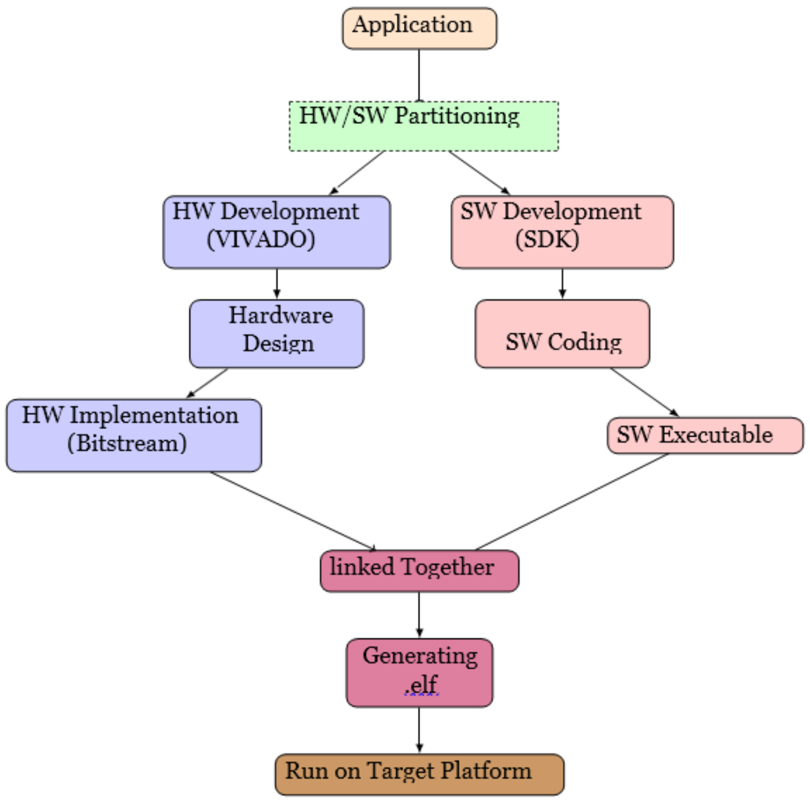


Figure 3. 1: HW/SW co-design overview of the design flow for Zynq SoC

SDK is then used to create an OpenCV-based software application that will use the customized board interface data and FPGA hardware configuration by importing the hardware design information from Vivado. The generated executable file (.elf) is finally copied to the SD card to run the OpenCV-based application on the ZedBoard. This flow is demonstrated in Figure 3.1

3.1 Host PC setup

The host PC that has been used in the experiments to implement the application and program the target board is based on a 64-bit Intel Core i7-6600U Quad Core CPU, 2.6 GHz using 8 GB of main memory running Linux Ubuntu 16.04.

The software tools that have been used for simulation/implementation are the Xilinx Vivado design suite (version 2014.2), Software Development Kit (SDK) and PuTTY terminal emulator. Xilinx Vivado is used to design the hardware blocks on which Linux is running on ZedBoard. Writing the OpenCV software executing on the ARM CPU was done using Xilinx SDK.

3.2 Linux Boot Procedure on ZedBoard

In order to boot Linux on the ZedBoard:

The Linux Kernel is downloaded from Xilinx Website [13]. A separate Linux desktop/laptop machine is needed to program the SD card. The machine should have an SD card slot that can be used and accessed by the Linux system.

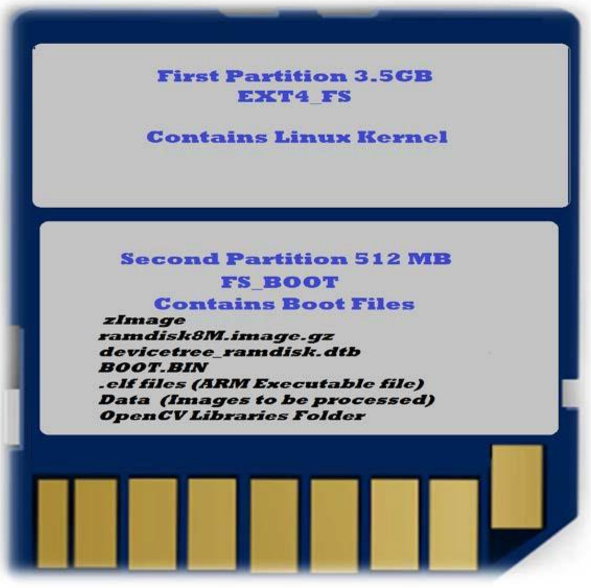


Figure 3. 2: SD card boot files

system.bit: this bitstream file contains the programmable logic part to initialize the Zynq FPGA Programmable Logic (PL) hardware zone.

FSBL.elf: this executable contains the First Stage Boot Loader. It performs several operations like configuring the Processing System PS at first, then loading the bitstream that initializes the PL. When this is done, it loads u-boot in memory and starts its execution.

u-boot.elf: u-boot (Universal BootLoader) is used as the Second Stage Boot Loader. Its role is to load the kernel image, the Device Tree blob as well as the file system and execute them.

BOOT.bin: In order to be executed from the SD card, these last three must be "Wrapped" in the same file: BOOT.bin. This is done from SDK. The three files must be placed in the boot order (FSBL, PL and u-boot) and the FSBL must have the flag "Bootloader", the others must have the flag "data”, it is necessary to specify the extension ***.elf*** for u-boot under Windows otherwise SDK will generate a file which does not work.

Device-tree.dtb: the kernel needs to know all the details of the hardware it is running on. For that, it must be given a device tree blob which contains the low-level hardware information of the ZedBoard (or at least the system (PS + PL) that we integrated in the ZedBoard).

zImage: this is the core of the system, the Linux kernel. It comes from the cross-compilation of sources of a standard kernel modified by Xilinx to adapt to the architecture of Zynq. These sources are retrieved from the Xilinx GIT repository [13].

RamDisk: in order to use the kernel, it is necessary to have a minimum of directories and basic programs (echo, insmod, mount, and so on). This is the role of the initrd (Initial Ram Disk) which is the minimalist file system stored in live memory. On a server or desktop machine, it is only a temporary file system to mount the real file system. In the case of an embedded system like ours, it is the final file system.

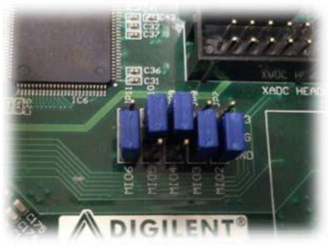
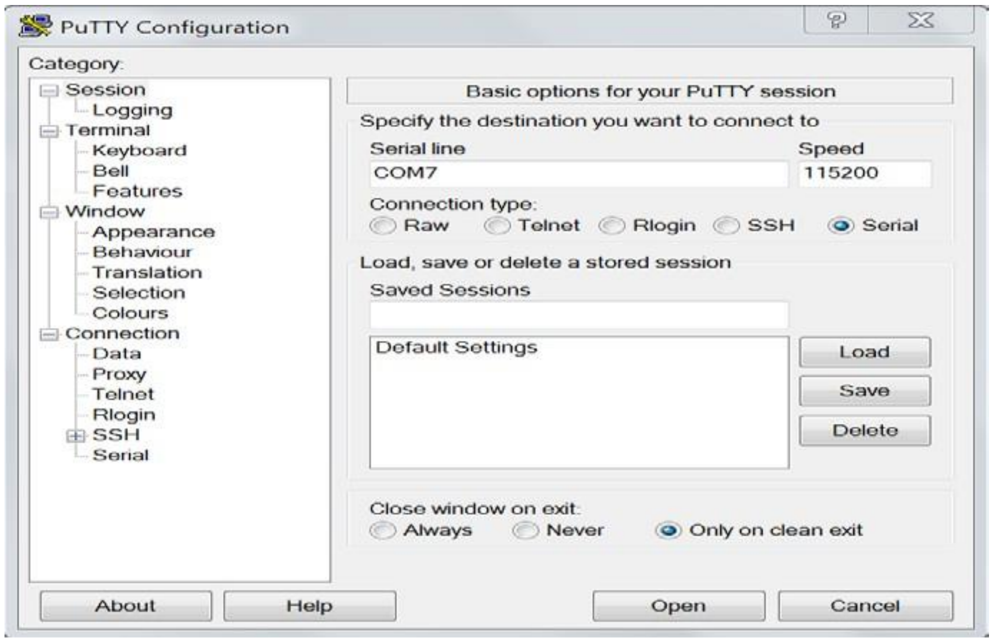


Figure 3. 3: Jumper Settings to boot from SD Card

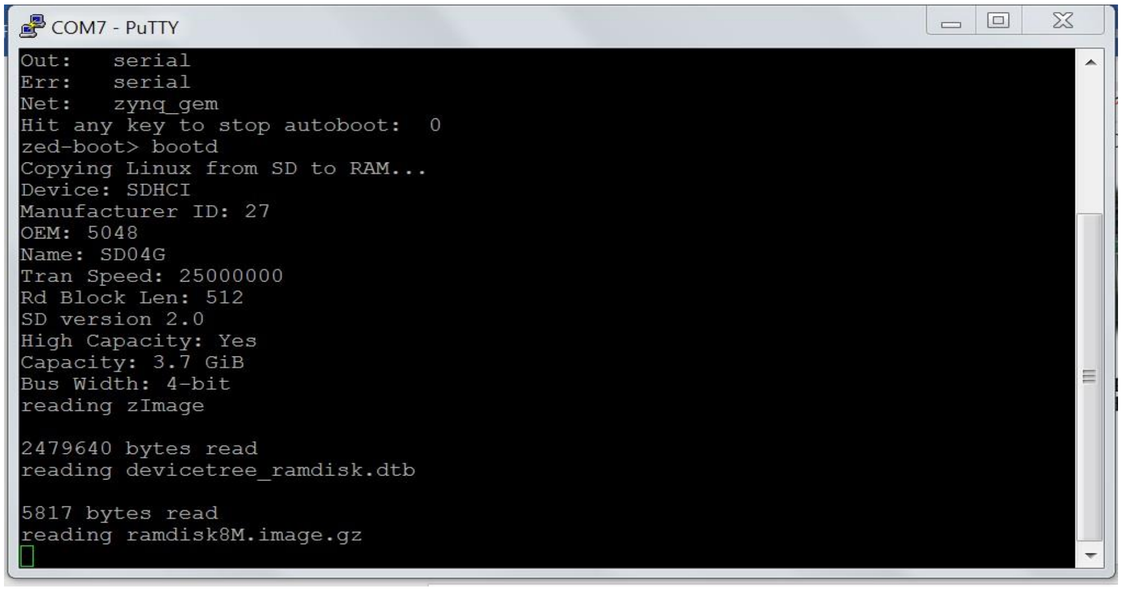
Once this is done, we connect the ZedBoard to the host PC through UART (micro-USB) for debugging. The following figure shows the configuration details of the Putty terminal emulator to connect to the board at 115200 bauds.



Specify the destination you want to connect to
Serial line: COM7
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Saved Sessions: Default Settings
Close window on exit: Always Never [✓ Only on clean exit]

Figure 3. 4: Putty Terminal Configuration

Once the ZedBoard is turned on, the booting process starts as shown in Figure3.2.4.

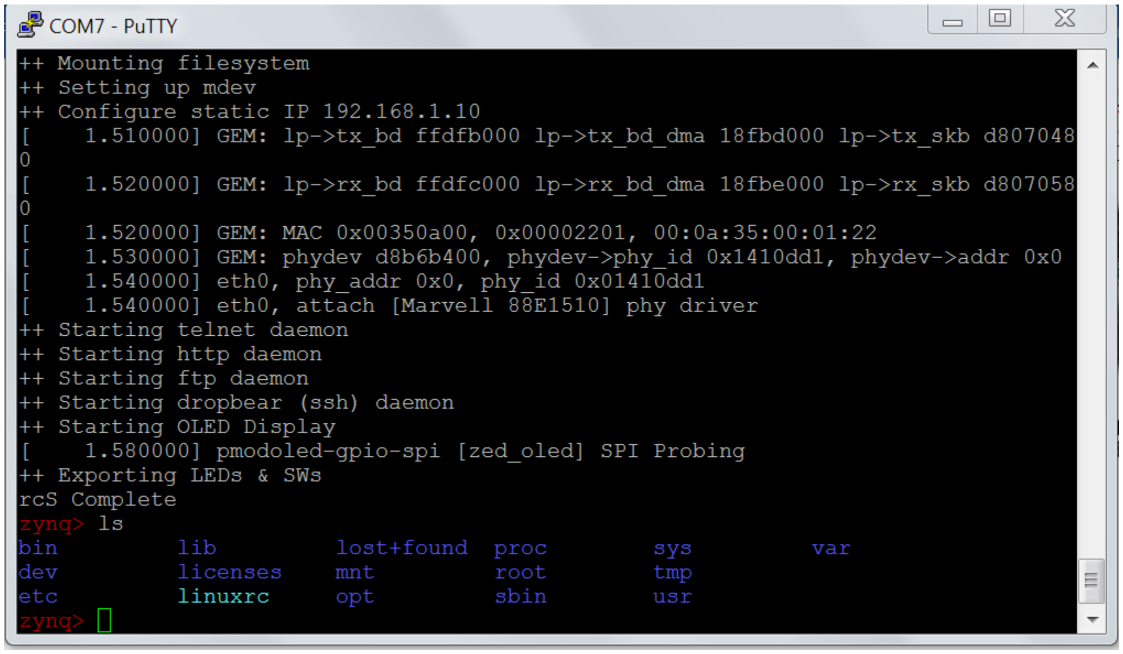


```
Out:    serial
Err:    serial
Net:    zynq_gem
Hit any key to stop autoboot:  0
zed-boot> bootd
Copying Linux from SD to RAM...
Device: SDHCI
Manufacturer ID: 27
OEM: 5048
Name: SD04G
Tran Speed: 25000000
Rd Block Len: 512
SD version 2.0
High Capacity: Yes
Capacity: 3.7 GiB
Bus Width: 4-bit
reading zImage

2479640 bytes read
reading devicetree_ramdisk.dtb

5817 bytes read
reading ramdisk8M.image.gz
```

Figure 3. 5: Linux Boot up as shown in Putty Terminal Window



```
++ Mounting filesystem
++ Setting up mdev
++ Configure static IP 192.168.1.10
[    1.510000] GEM: lp->tx_bd ffdfb000 lp->tx_bd_dma 18fbd000 lp->tx_skb d807048
0
[    1.520000] GEM: lp->rx_bd ffdfc000 lp->rx_bd_dma 18fbe000 lp->rx_skb d807058
0
[    1.520000] GEM: MAC 0x00350a00, 0x00002201, 00:0a:35:00:01:22
[    1.530000] GEM: phydev d8b6b400, phydev->phy_id 0x1410dd1, phydev->addr 0x0
[    1.540000] eth0, phy_addr 0x0, phy_id 0x01410dd1
[    1.540000] eth0, attach [Marvell 88E1510] phy driver
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
++ Starting OLED Display
[    1.580000] pmodoled-gpio-spi [zed_oled] SPI Probing
++ Exporting LEDs & SWs
rcS Complete
zynq> ls
bin         lib         lost+found  proc        sys         var
dev         licenses    mnt         root        tmp
etc         linuxrc     opt         sbin        usr
zynq> 
```

Figure 3. 6: Command prompt of the booted Linux on ZedBoard

In order to be able to execute the generated ***.elf*** files of the already built OpenCV applications under Xilinx SDK, The following commands are issued:

zynq> **cd dev** // Access the /dev directory in order to mount the sd card

zynq> **mount /dev/mmcblk0p1 /mnt** // command to mount the content of sd card to /mnt directory

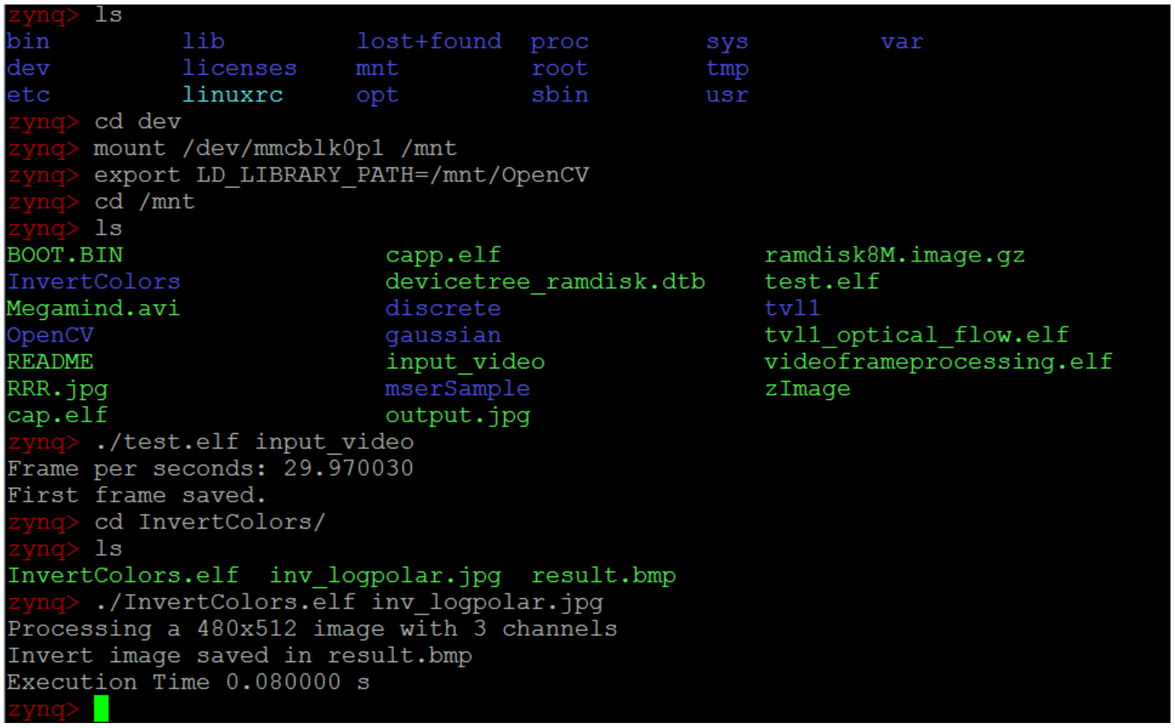
zynq> **export LD_LIBRARY_PATH=/mnt/OpenCV** // set the path of the OpenCV libraries

zynq> **cd /mnt** // Access the /mnt folder containing the content of the sd card

zynq> **ls** // list the content of the /mnt folder

zynq> **./InvertColors inv_logpolar.jpg** // execute an executable file

Once the OpenCV application finishes execution, its total execution time is printed to the terminal and the resulting images are saved in the SD card as shown in Figure bellow.



```
zynq> ls
bin         lib         lost+found  proc        sys         var
dev         licenses    mnt         root        tmp
etc         linuxrc     opt         sbin        usr
zynq> cd dev
zynq> mount /dev/mmcblk0p1 /mnt
zynq> export LD_LIBRARY_PATH=/mnt/OpenCV
zynq> cd /mnt
zynq> ls
BOOT.BIN                  capp.elf                  ramdisk8M.image.gz
InvertColors              devicetree_ramdisk.dtb    test.elf
Megamind.avi              discrete                  tvl1
OpenCV                    gaussian                  tvl1_optical_flow.elf
README                    input_video               videoframeprocessing.elf
RRR.jpg                   mserSample                zImage
cap.elf                   output.jpg
zynq> ./test.elf input_video
Frame per seconds: 29.970030
First frame saved.
zynq> cd InvertColors/
zynq> ls
InvertColors.elf  inv_logpolar.jpg  result.bmp
zynq> ./InvertColors.elf inv_logpolar.jpg
Processing a 480x512 image with 3 channels
Invert image saved in result.bmp
Execution Time 0.080000 s
zynq> 
```

Figure 3. 7: Example of successful execution of an algorithm on ZedBoard

3.3 On-chip Hardware Architecture

The following Figure 3.3.1 show the designed On-chip hardware that allows the Linux Operating System to run on the ZedBoard.

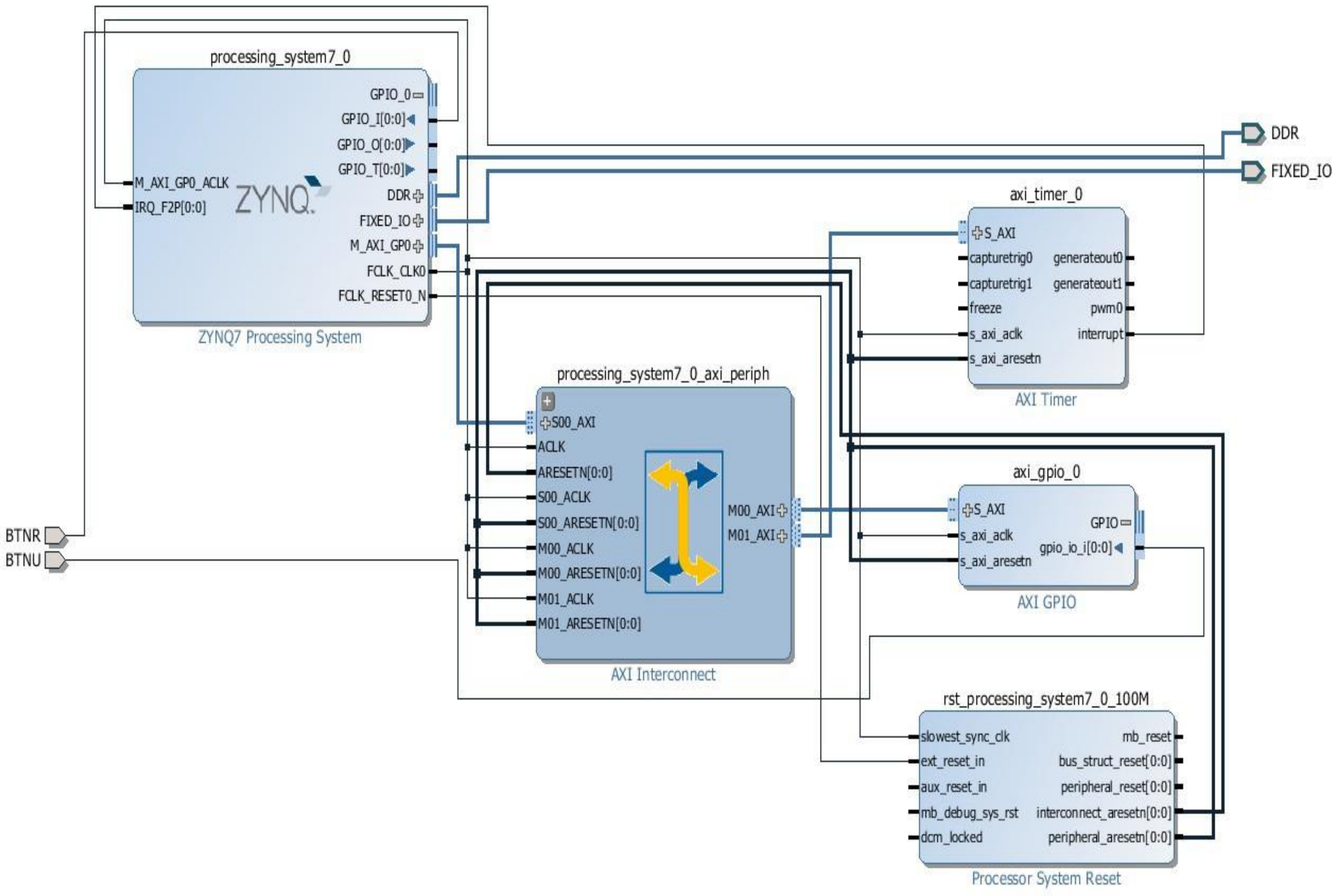
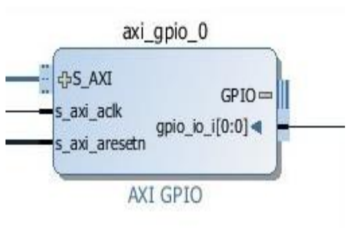


Figure 3. 8: On-chip Hardware design

The Zedboard needs to be configured in order for the Linux operating system to run as the above diagram shows. The brain of this configuration is the ZYNQ7 Processing System block which can be Re-customized as shown in (Figure 2.4). The interfaces between the processing system and programmable logic mainly consist of three groups: the extended multiplexed I/O (EMIO), programmable logic I/O, and the AXI I/O groups. The Zynq-7000 device configuration wizard configures the Processing System 7 core.

AXI GPIO:

The Xilinx LogiCORE IP AXI General Purpose Input/output (GPIO) core provides a general purpose input/output interface to the AXI interface. The AXI GPIO can be configured as either a



single or a dual-channel device. The width of each channel is independently configurable. This 32-bit soft Intellectual Property (IP) core is designed to interface with the AXI4-Lite interface.

Figure 3. 9: AXI GPIO IP Block

AXI Timer:

The AXI Timer/Counter modules provides an AXI4-Lite interface to communicate with the host. The timer/counter design has the following key modules: AXI4-Lite Interface, Timer Registers, 32-bit Counters, Interrupt Control, Pulse Width Modulation (PWM).

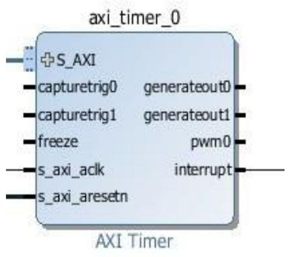


Figure 3. 10: AXI Timer IP Block

AXI Interconnect:

The AXI Interconnect IP connects one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices. The AXI interfaces conform to the AMBA AXI version 4 specifications from ARM®, including the AXI4-Lite control register interface subset. The Interconnect IP is intended for memory-mapped transfers only; AXI4-Stream transfers are not applicable. The AXI Interconnect IP can be used from the Vivado IP catalog as a pcore from the Embedded Development Toolkit (EDK) or as a standalone core from the CORE Generator IP catalog.

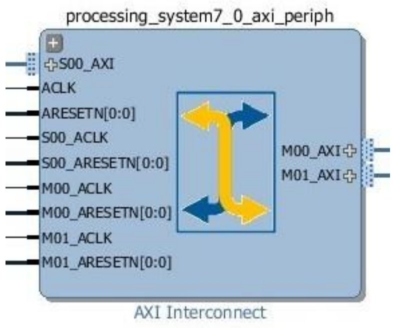


Figure 3. 11: AXI Interconnect IP

Processor System Reset:

The Processor System Reset Module provides a reset function.

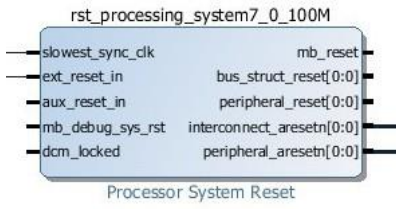


Figure 3. 12: Processor System Reset IP

Once the hardware is synthesized and implemented in Vivado, the hardware design is exported to the SDK using the Export Hardware function. For this design, the 'Include Bitstream' option is selected since there is a bitstream generated for the PL so that it will also be exported to SDK.

Summary

This chapter summarized and discussed the necessary background of hardware and software co-design system design and the experimental setup that has been used in the evaluation process. The hardware/software co-design is a fundamental process to divide the functions into the hardware and software in order to efficiently map applications into a heterogeneous platform.

Chapter 4

Results and Discussions

In this chapter, an overview of the implemented algorithms is given followed by the experimental results. The execution times for the executed OpenCV algorithms on both the host PC and ZedBoard are outlined. These results are then discussed.

4.1 Overview of the Evaluated OpenCV Algorithms

In this section we give a brief definition of the OpenCV algorithms that we tested. The execution time is measured for each algorithm, and compared later with the execution time on the host PC. More Detailed description of the algorithms are available in OpenCV website documentation section [7].

1. Object/features matching algorithm:

We used a queryImage, found some feature points in it, we took another trainImage, found the features in that image too and we found the best matches among them. In short, we found locations of some parts of an object in another cluttered image. This information is sufficient to find the object exactly on the trainImage, so we have a queryImage and a trainImage. We will try to find the queryImage in trainImage using feature matching.

Input Images:

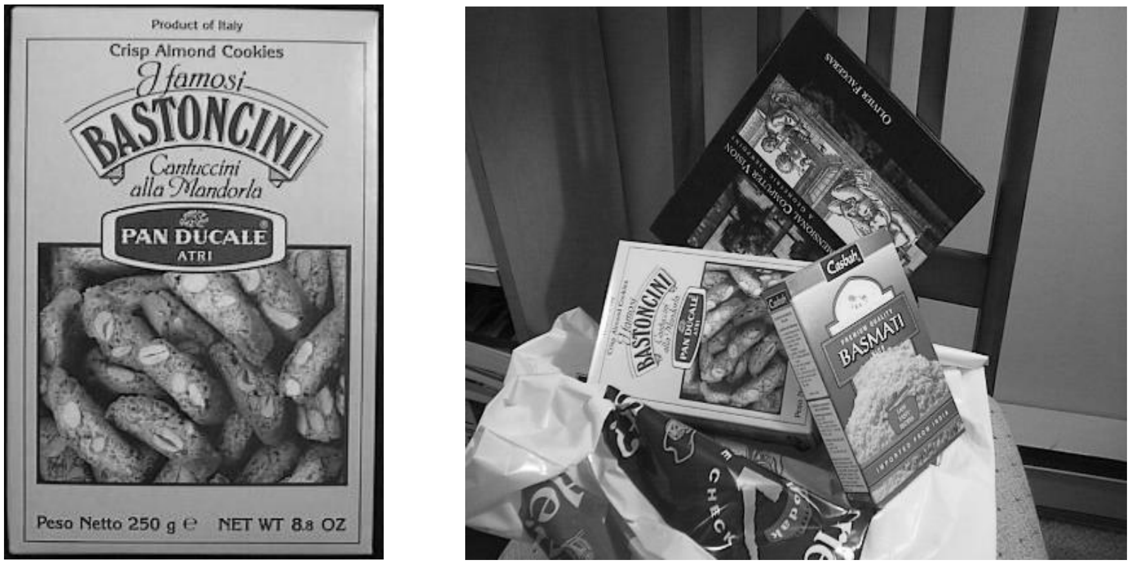


Figure 4. 1: queryImage and trainImage used for object matching algorithms

Output Images:

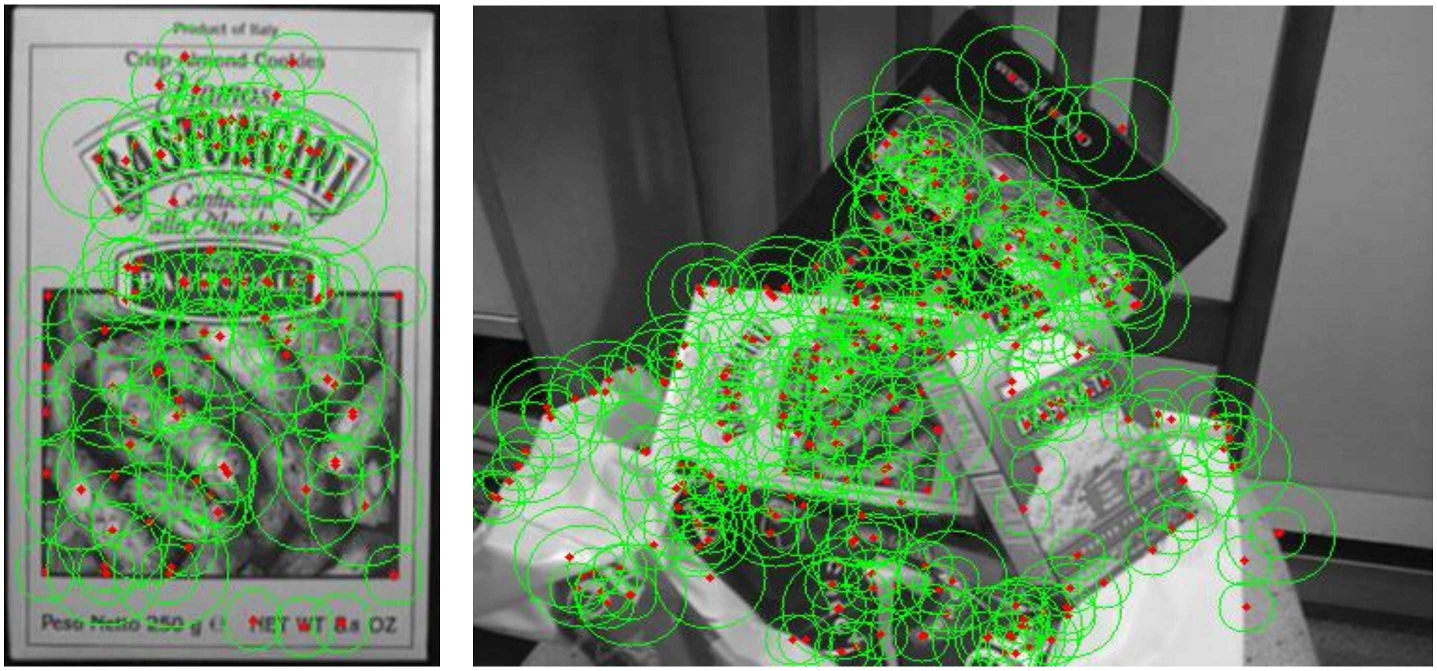


Figure 4. 2: detected features in queryImage and trainImage

Final Result: object matched

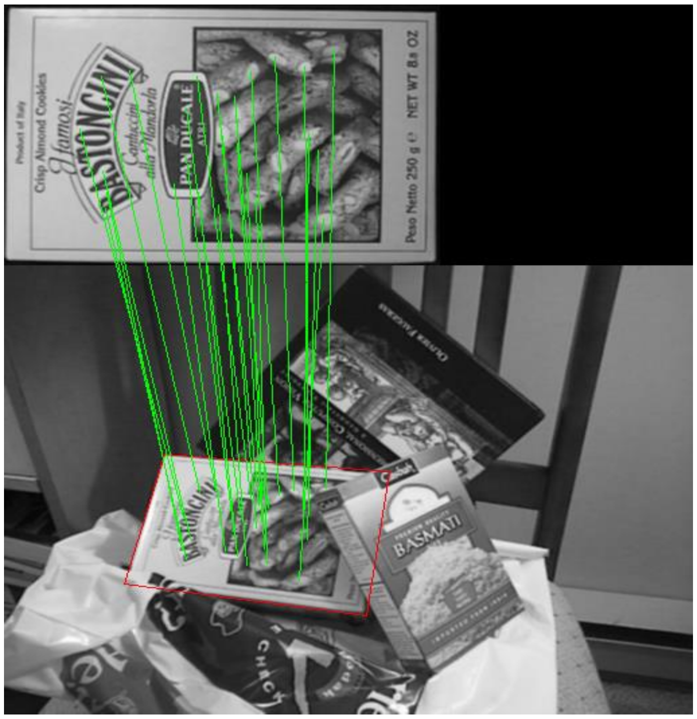


Figure 4. 3: Results of object matching algorithms

2. Optical Flow:

Optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by the movement of object or camera. It is 2D vector field where each vector is a displacement vector showing the movement of points from first frame to second [19].

Optical flow works on several assumptions:

1. The pixel intensities of an object do not change between consecutive frames.
2. Neighboring pixels have similar motion.

Consider a pixel I(x,y,t) in first frame (Check a new dimension, time, is added here. Earlier we were working with images only, so no need of time). It moves by distance (dx,dy) in next frame taken after dt time. So since those pixels are the same and intensity does not change, we can say,

$$I(x,y,t)=I(x+dx,y+dy,t+dt) \qquad (4.1)$$

Then take Taylor series approximation of right-hand side, remove common terms and divide by dt to get the following equation:

$$fxu+fyv+ft=0 \qquad (4.2)$$

where:

$$fx=\partial f\partial x; \quad fy=\partial f\partial y \qquad (4.3)$$

$$u=dxdt;v=dydt \qquad (4.4)$$

Above equation is called Optical Flow equation. In it, we can find fx and fy, they are image gradients. Similarly, ft is the gradient along time. But (u,v) is unknown. We cannot solve this one equation with two unknown variables. So several methods are provided to solve this problem and one of them is Lucas-Kanade.

Lucas-Kanade method

We have seen an assumption before, that all the neighboring pixels will have similar motion. Lucas-Kanade method takes a 3x3 patch around the point. So all the 9 points have the same motion. We can find (fx,fy,ft) for these 9 points. So now our problem becomes solving 9 equations with two unknown variables which is over-determined. A better solution is obtained with least square fit method. Below is the final solution which is two equation-two unknown problem and solve to get the solution.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}{}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}{}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix} \quad (4.5)$$

Optical Flow from two input images of the same size:

Input Images:

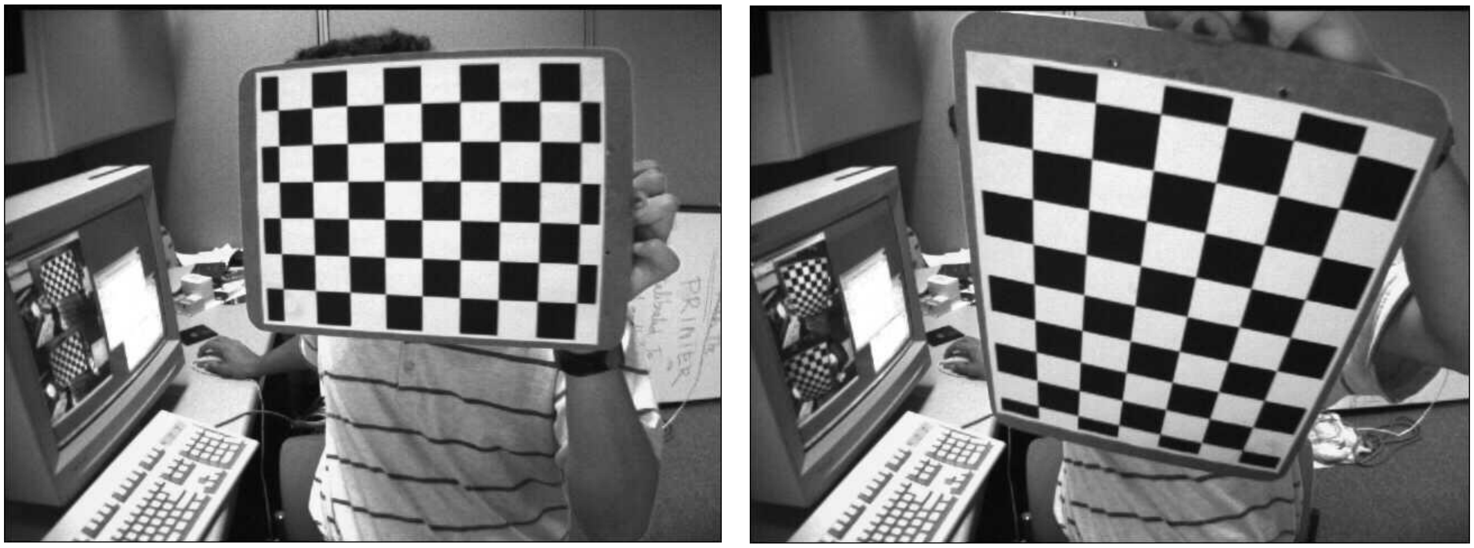


Figure 4. 4: Test images optical flow algorithms

Output File: The flow is detected

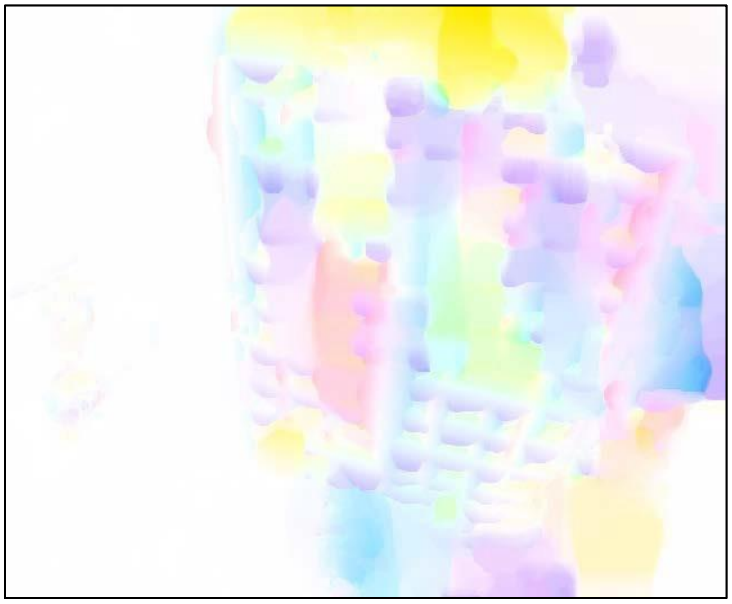


Figure 4. 5: Resulting Image of optical flow algorithms

Optical Flow from a video File:

It detects the flow of people walking on the street and displays the result.

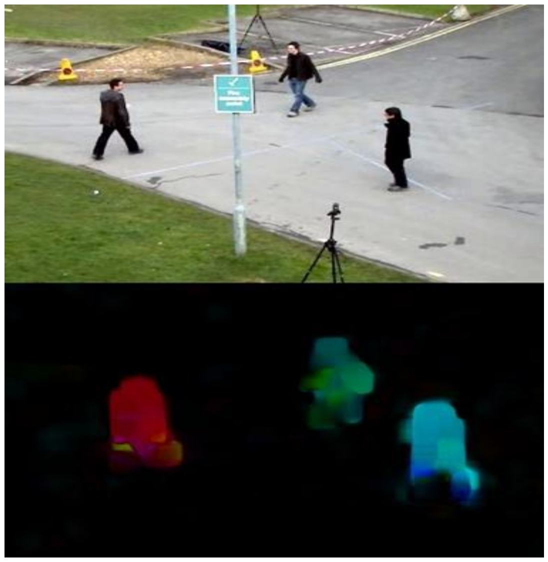


Figure 4. 6: Resulting optical flow algorithms from video file

3. Smoothing:

This is a simple algorithm that smooths a given image. It reduces the sharpness of edges and smooths out details in an image. OpenCV implements several of the most commonly used methods.

The main aim of image smoothing is to remove noise in digital images. It is a classical matter in digital image processing to smooth image. And it has been widely used in many fields, such as image display, image transmission and image analysis. Image smoothing has been a basic module in almost all the image processing software[2].

Figure 4. 7: Test Image of smoothing algorithm

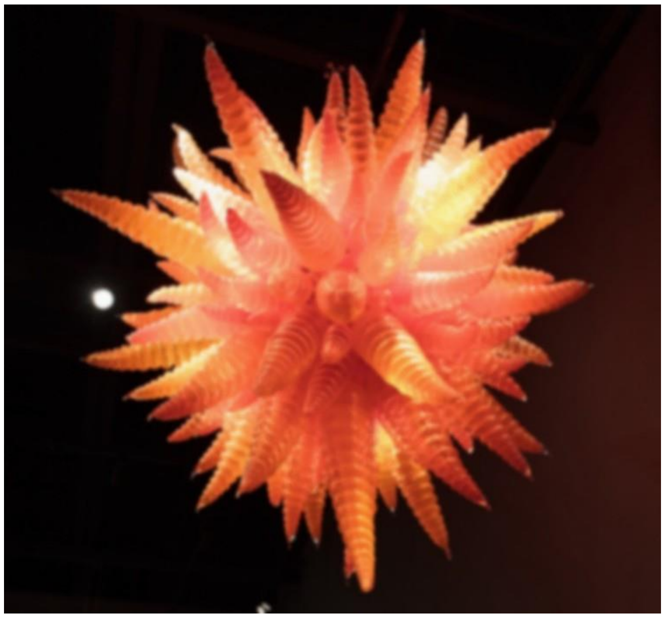


Figure 4. 8: Median blur image

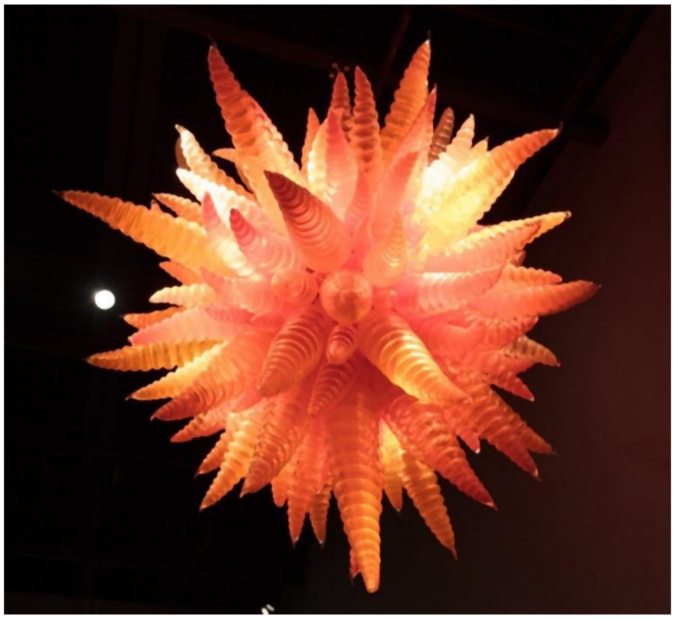


Figure 4. 9: Gaussian blur image

4. Edge Detection:

The Canny Edge detector was developed by John F. Canny in 1986. Also known to many as the optimal detector, Canny algorithm aims to satisfy three main criteria [2].

Low error rate: Meaning a good detection of only existent edges.

Good localization: The distance between edge pixels detected and real edge pixels have to be minimized.

Minimal response: Only one detector response per edge.

Steps:

1. Filter out any noise. The Gaussian filter is used for this purpose. An example of a Gaussian kernel of **size 5** that might be used is shown below:

$$K = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \qquad (4.6)$$

2. Find the intensity gradient of the image. For this, we follow a procedure analogous to Sobel:

a. Apply a pair of convolution masks (in X and Y directions:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$
$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \quad (4.7)$$

b. Find the gradient strength and direction with:

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan(\frac{G_y}{G_x}) \quad (4.8)$$

The direction is rounded to one of four possible angles (namely 0, 45, 90 or 135)

3. Non-maximum suppression is applied. This removes pixels that are not considered to be part of an edge. Hence, only thin lines (candidate edges) will remain.

4. Hysteresis*:* The final step. Canny does use two thresholds (upper and lower):

1. If a pixel gradient is higher than the *upper* threshold, the pixel is accepted as an edge
2. If a pixel gradient value is below the *lower* threshold, then it is rejected.
3. If the pixel gradient is between the two thresholds, then it will be accepted only if it is connected to a pixel that is above the *upper* threshold.

Edge detection Sobel Gray Image

Input Image:

Result:

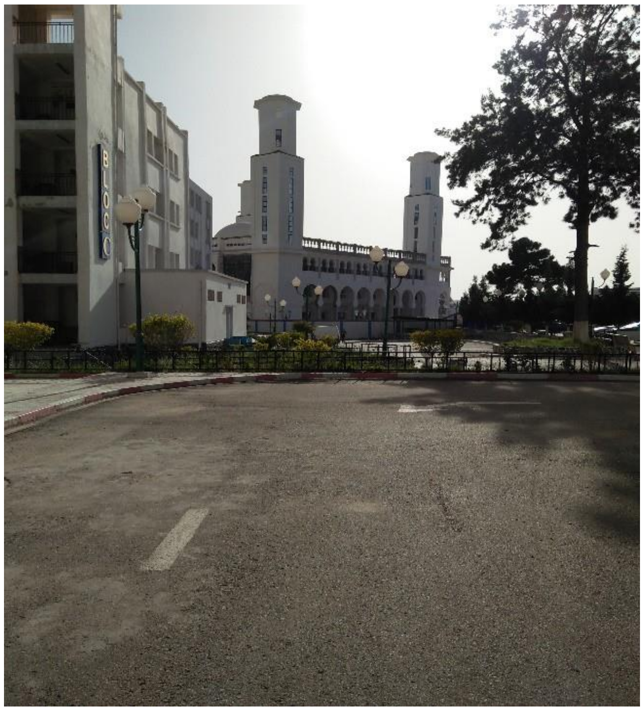


Figure 4. 10: *Test Image of Edge Detection*

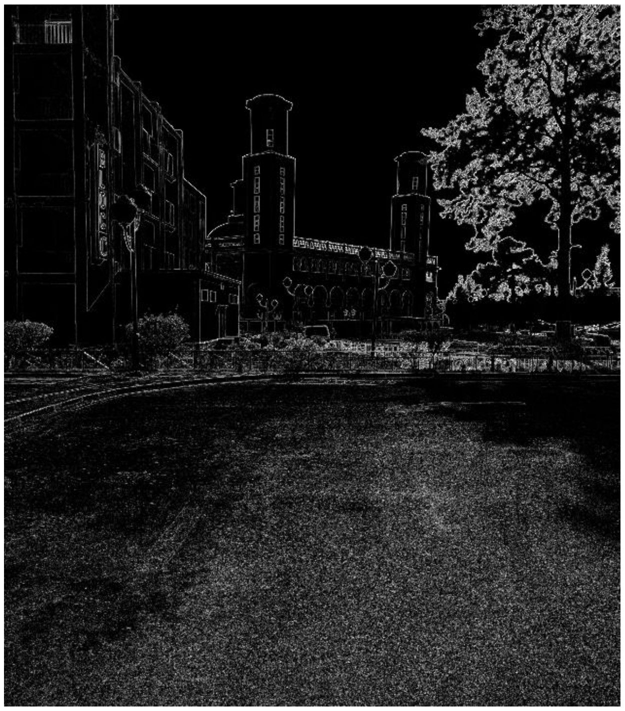


Figure 4. 11: *Result Image of Edge Detection*

Edge detection Sobel Color Image

Input Image:

Result:

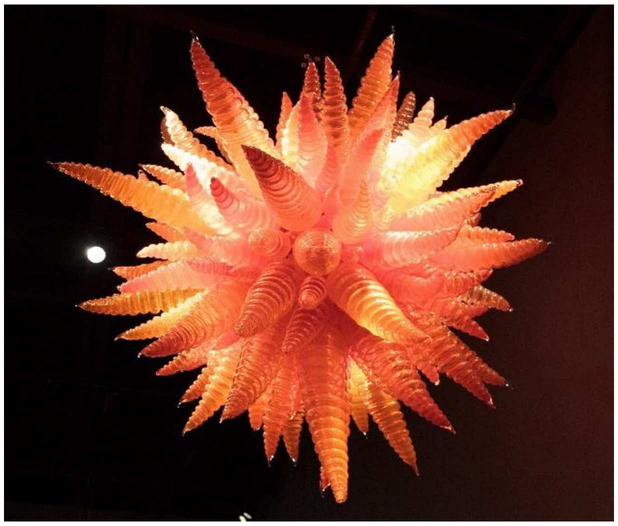


Figure 4. 12: Test Image of Edge Detection

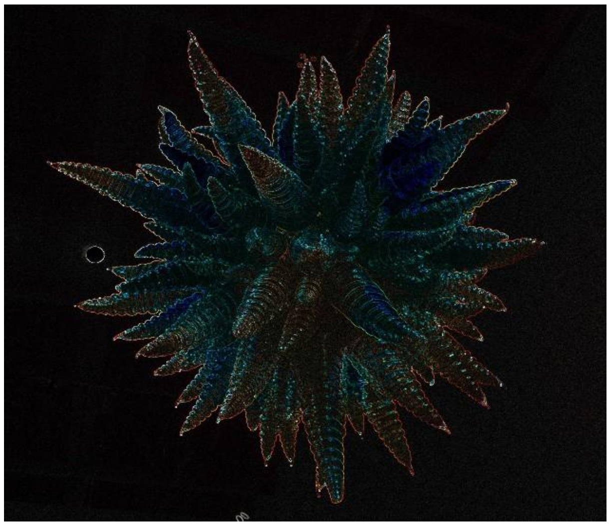


Figure 4. 13: Result image of Edge detection

5. Maximally Stable External Regions (MSER) Contour Extraction

MSER processing

The MSER extraction implements the following steps:

− Sweep threshold of intensity from black to white, performing a simple luminance thresholding of the image.

− Extract connected components ("Extremal Regions")

− Find a threshold when an extremal region is "Maximally Stable", i.e. local minimum of the relative growth of its square. Due to the discrete nature of the image, the region below /above may be coincident with the actual region, in which case the region is still deemed maximal.

− Approximate a region with an ellipse (this step is optional)

− Keep those regions descriptors as features. However, even if an extremal region is maximally stable, it might be rejected if:

− it is too big (there is a parameter MaxArea)

− it is too small (there is a parameter MinArea)

− it is too unstable (there is a parameter MaxVariation)

− it is too similar to its parent MSER Margin = the number of thresholds for which the region is stable

− Input image:

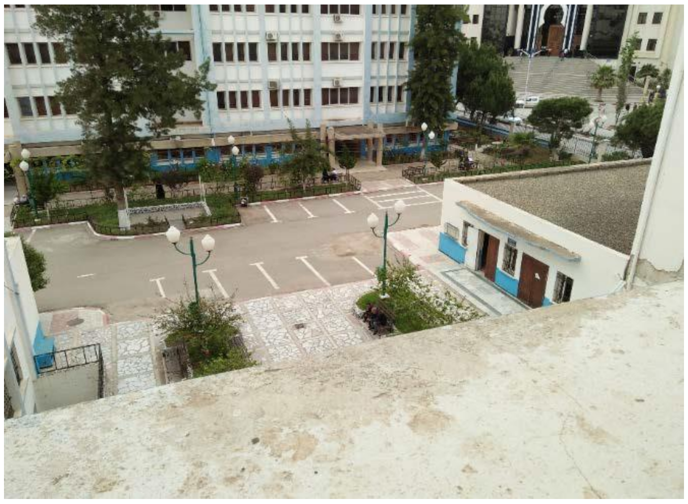


Figure 4. 14: Test Image of MSER algorithm

Output Images:

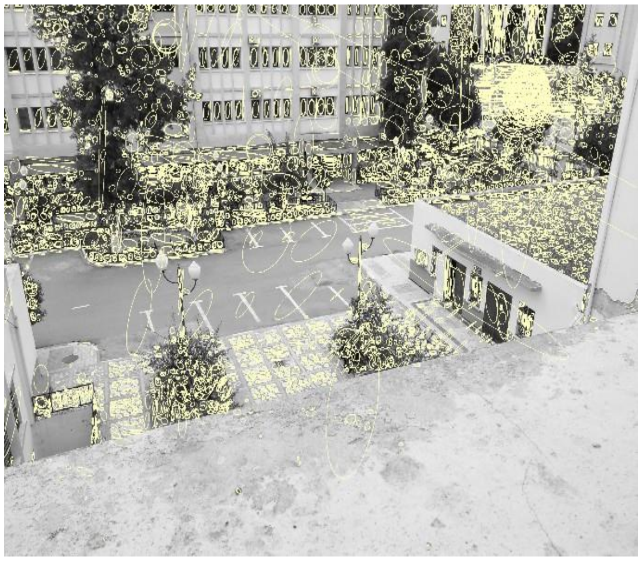


Figure 4. 15: MSER ellipses

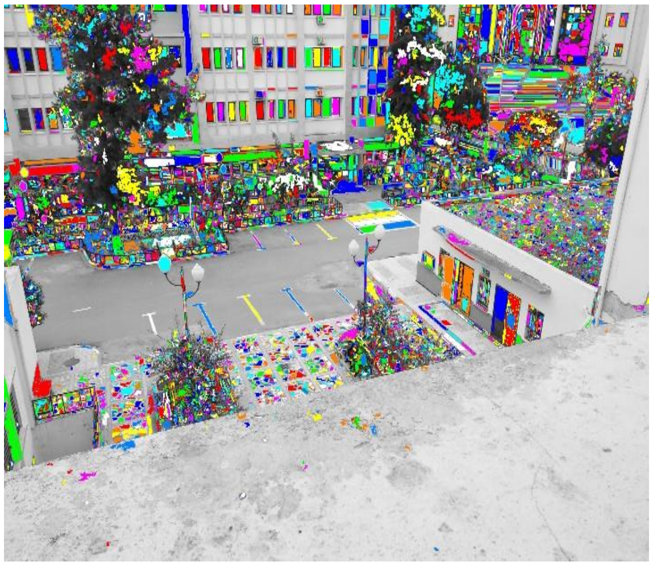


Figure 4. 16: Image Response

6. Discrete Fourier Transform

The Fourier Transform will decompose an image into its sinus and cosines components. In other words, it will transform an image from its spatial domain to its frequency domain. The idea is that any function may be approximated exactly with the sum of infinite sinus and cosines functions.

Figure 4. 17: Original test Image

Figure 4. 18: Gray Scale Image

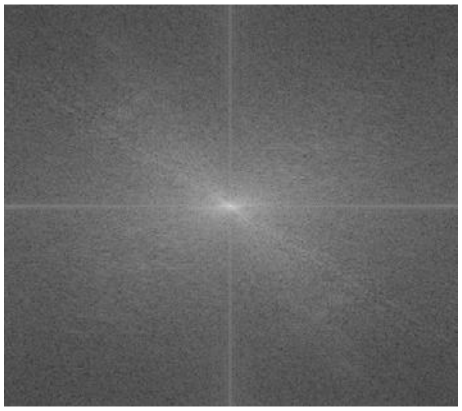


Figure 4. 19: Spectrum magnitude

7. Color Inversion:

Color inversion, also known as the negative effect, is one of the easiest effects to achieve in image processing. Color inversion is achieved by subtracting each RGB color value from the maximum possible value (usually 255). In pseudo-code performing the color inversion would go something like this [16]:

```
colour = GetPixelColour(x, y)
invertedRed   = 255 - Red(colour)
invertedGreen = 255 - Green(colour)
invertedBlue  = 255 - Blue(colour)
PutPixelColour(x, y) = RGB(invertedRed, invertedGreen, invertedBlue)
```

To demonstrate the color inversion effect we have below **Figure 4.1. 1** and **Figure 4.1. 2** showing both the original and inverted versions.

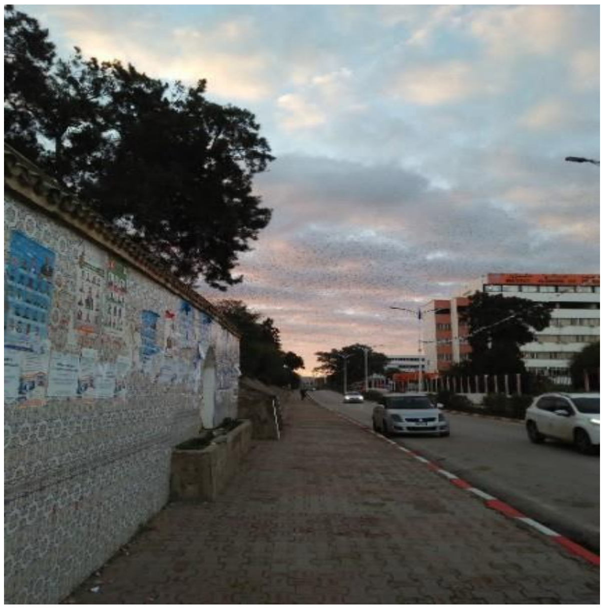


Figure 4. 20: Test Image

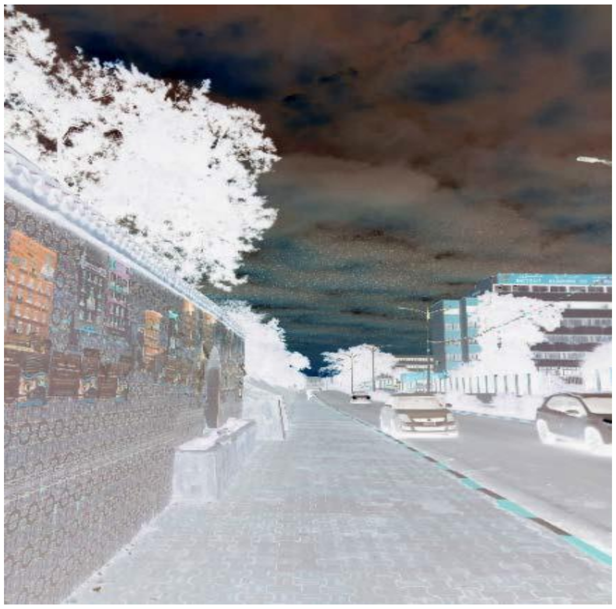


Figure 4. 21: Result of color inversion algorithm

8. Greyscale filter:

In this algorithm we will look at how to convert a color image to greyscale [16].

In order to convert a particular color into greyscale we need to work out what the intensity or brightness of that color is. A quick way of calculating the intensity is to obtain the mean value of the red, green and blue components. The formula for this is:

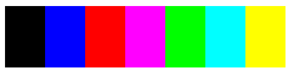
$$I = \frac{R + G + B}{3} \qquad (4.9)$$

Although it can produce reasonable results, this method is not perfect. The reason for this is that the human eye does not perceive reds, greens and blues at the same intensity level. The color green, for example, at maximum intensity looks brighter than blue at maximum intensity.

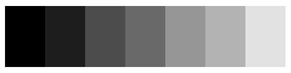
Another method of greyscale conversion which takes into account the human perception of color uses different weights for the red, green and blue components. This means that when you are calculating the value for the intensity each of the color components are multiplied by a weight value. There are a number of different values used for these weights but one of the most popular is the following:

$$I = 0.299R + 0.587G + 0.114B \qquad (4.10)$$

To illustrate the above points let’s have a look at an image of some colored bars:



Converting the image to greyscale using the weighted method will give us the following:



Let's have a look at the image which has been converted to greyscale:

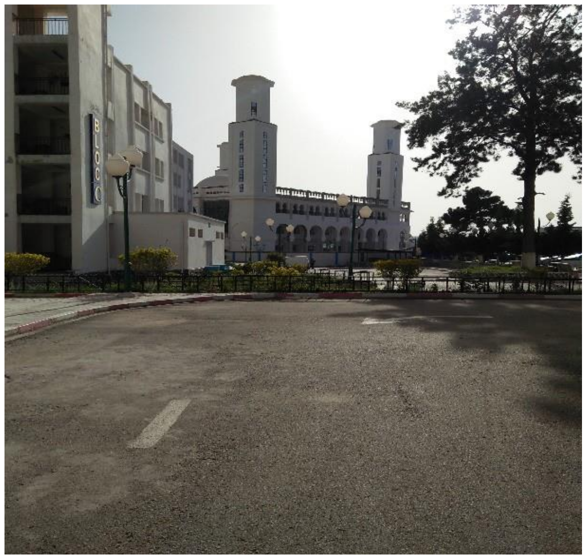


Figure 4. 22: Test Image

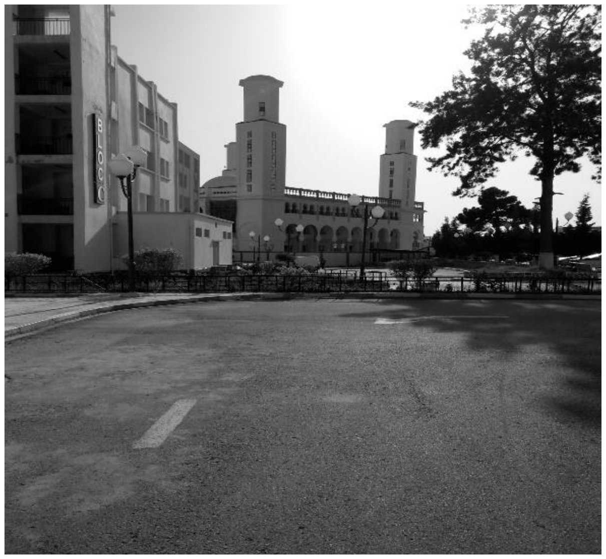


Figure 4. 23: Result of Grey filter algorithm

9. Background Substractor:

Background subtraction (BS) is a common and widely used technique for generating a foreground mask (namely, a binary image containing the pixels belonging to moving objects in the scene) by using static cameras. As the name suggests, BS calculates the foreground mask performing a subtraction between the current frame and a background model, containing the static part of the scene or, more in general, everything that can be considered as background given the characteristics of the observed scene [4].

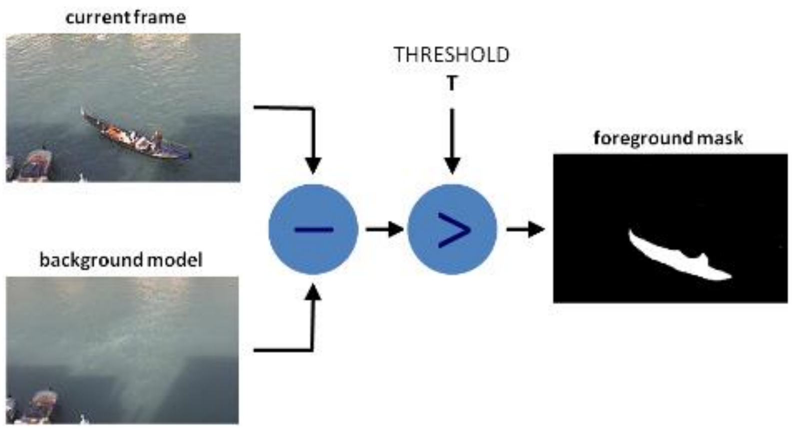


Figure 4. 24: Background Substractor Principle

Background modeling consists of two main steps: *Background Initialization* and *Background Update*. In the first step, an initial model of the background is computed, while in the second step that model is updated in order to adapt to possible changes in the scene.

Original color frame:

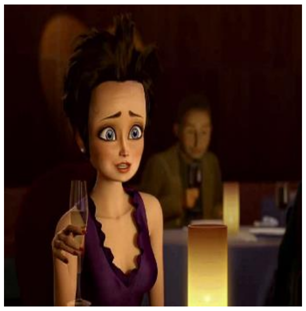


Figure 4. 25: Color Frame of a the video

Obtained background subtracted frames

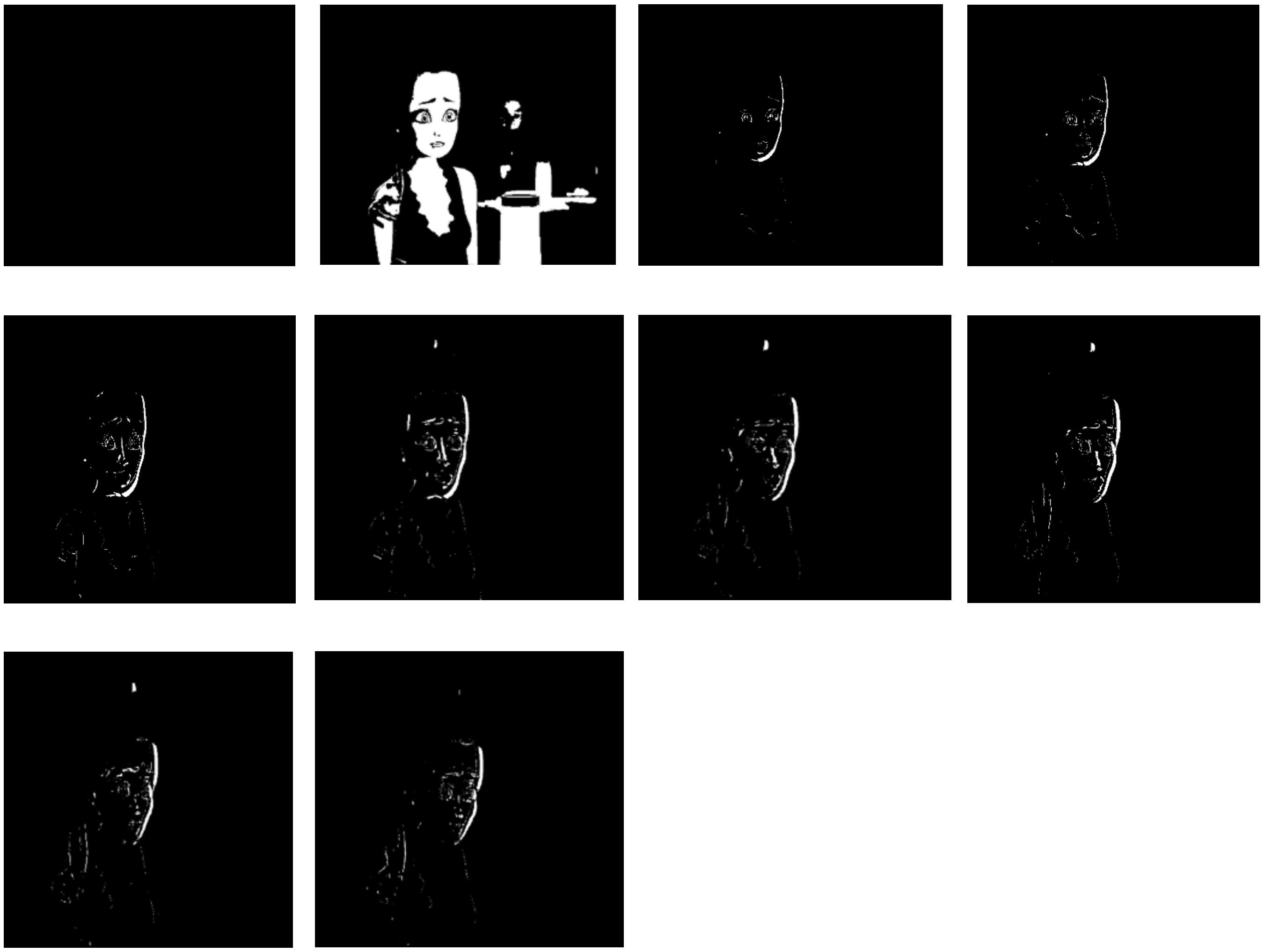
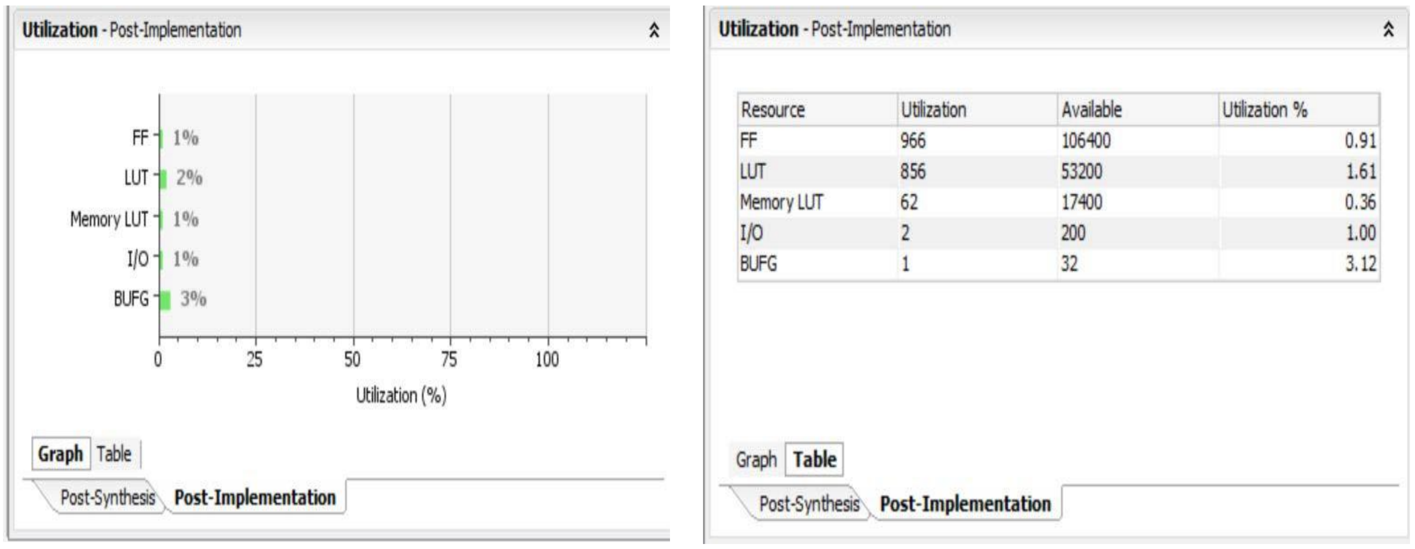


Figure 4. 26: Resulting Images from Background Substractor algorithm

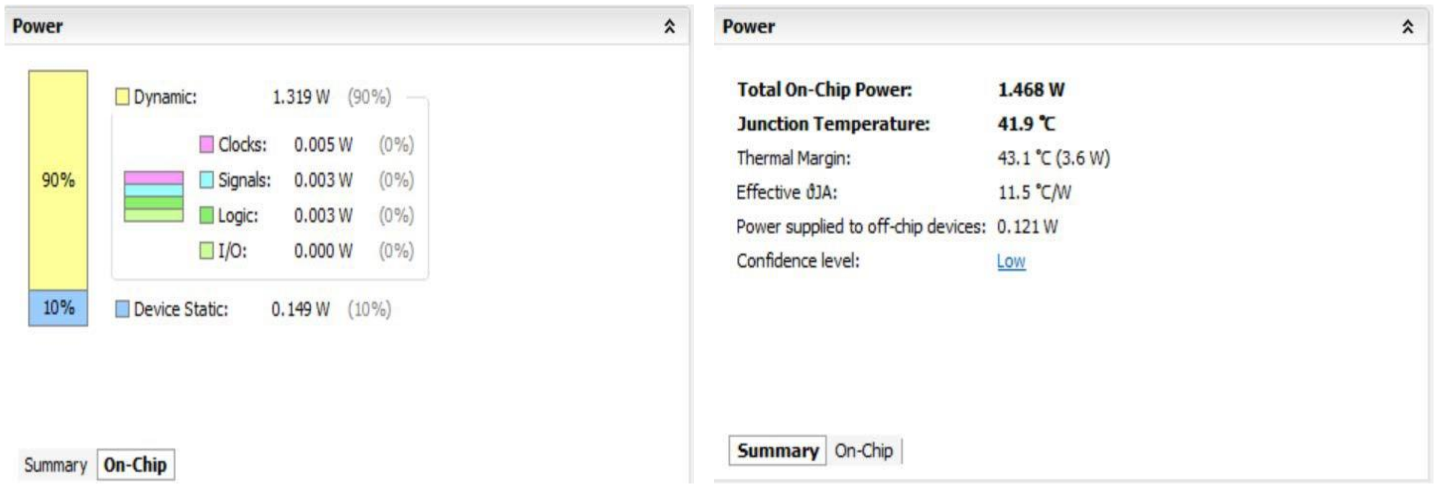
4.2 Hardware Resources Utilization:

The following table shows the hardware resources utilization. The design occupied nearly 2% of the PL section of the FPGA that includes the Timer, GPIO, Processor Reset, and Bus Interconnections blocks. The ARM processor is a hardcore processor that is already placed on the FPGA fabric and does not utilize any programmable blocks. The main tasks, which are running the Linux OS and the OpenCV applications, will be performed entirely by the ARM processor. The power consumption estimation is shown in Figure 4.28 below.



Resource	Utilization	Available	Utilization %
FF	966	106400	0.91
LUT	856	53200	1.61
Memory LUT	62	17400	0.36
I/O	2	200	1.00
BUFG	1	32	3.12

Figure 4. 27: Resources Utilization Graph and Table



Dynamic:	1.319 W	(90%)
Clocks:	0.005 W	(0%)
Signals:	0.003 W	(0%)
Logic:	0.003 W	(0%)
I/O:	0.000 W	(0%)
Device Static:	0.149 W	(10%)

Total On-Chip Power:	**1.468 W**
Junction Temperature:	**41.9 ℃**
Thermal Margin:	43.1 ℃ (3.6 W)
Effective θJA:	11.5 ℃/W
Power supplied to off-chip devices:	0.121 W
Confidence level:	Low

Figure 4. 28: Power Consumption Estimation

4.3 Algorithms Execution Times

Table 4.1 shows the obtained results of our implemented algorithms, the first and the second column identifies name of each algorithm, the next column presents the resolution of the processed images, the fourth column shows the execution time of the algorithms in both Host PC and ZedBoard. The fifth column reveals the ratio between the execution times in both platforms (ratio = ZedBoard execution time / PC execution time). Finally, the last column gives the number of Frames each platform can process in one second (fps).

	Algorithms	**Image Resolution**	**Execution Time in milliseconds**		**Ratio**	**FPS**	
			ZedBoard	Computer		ZedBoard	Computer
1	**Object/features Matching**	512x384 324x223	86.45	33.13	3	11.6	30
2	**Optical Flow**	640x480	136.87×10^3	65.7×10^3	2	0.015	0.3
3	**Image Smoothing**	3456x3132	50.23×10^3	1.75×10^3	29	0.02	0.57
4	**Edge Detection**	2448x3264 3456x3132	6.18×10^3	601.3	10	0.16	1.66
5	**MSER contour Extractor**	3264x2448	1491.95	336.68	5	0.7	3
6	**Fourier Transform**	334x305	170.00	16.263	10	5.88	61.5
7	**Color Inversion**	2448x3264	80.00	7.795	10	12	128
8	**Greyscale Filter**	2448x3264	170.00	11.293	15	5.88	88.5
9	**Background Substractor**	720x528	2.62×10^3	701.00	4	3.9	14.2

Table 4. 1: Execution times comparison results table

Discussion:

The obtained results demonstrate the execution time of each algorithm in ZedBoard and On Computer. It can be seen clearly that there is a huge difference between the PC and ZedBoard execution times with a maximum ratio of 29. It can also be noticed that the ratio is different from one algorithm to another. On average, the PC runs the selected applications 7 times faster than the ZedBoard with an average of 36 fps. The ZedBoard, however, achieved an average image processing speed of approximately 5 fps, which is sufficient for a number of real-time computer vision applications.

This could be explained by the fact that each algorithms uses different function and each function uses resources of both computer and ZedBoard differently. Most importantly, the current version that we have evaluated is a pure software implementation which does not contain any hardware acceleration, unlike the new xfOpenCV hardware accelerated library. Therefore, the main two resources that impact the performance of the applications are the processor clock speed and the available system memory.

The Computer is faster in execution than the ZedBoard because it runs at 2.6 GHz which is three time the clock of the ZedBoard (886MHz) [Figure 2.3]. Also the total memory (RAM) of the host PC is 8 Gb whereas the ZedBoard's memory is just 512 Mb. In our case the processor of the host PC is a Quad core with 6 Mb cache memory. It also supports hyperthreading which gives it the ability to create many virtual cores as they have physically, meaning each core has 2 processing threads comparing to the Zedboard which has dual ARM processor with a cache memory of 512Kb, so quad core CPU will offer more processing power than dual core.

In the computer, the data used by the program is loaded from the hard disk which is an SSD (Solid State Drive) to the RAM. Therefore, loading the input/result images data is fast comparing to the ZedBoard which uses an SD Card which is slower.

Furthermore, the operating system of the host PC (Linux Ubuntu 16.04) can manage the execution of applications on the processor using the full cores (Quad Core in our case), whereas in the ZedBoard the Linux kernel uses only one core to process the applications. This could also explain the difference between the execution times on both platforms.

In addition to the hardware parameters that impacts the execution time, we must talk also about the software parameters. We analyzed the algorithms to calculate their complexity (Big-O notation). It was noticed that the execution time depends heavily on the image input size (image resolution).

It is already known that image processing applications are computationally intensive tasks. This is because, typically, every pixel of an image must be processed. For example, in Edge detection, an image of 2448x3264 pixels needs 8 million pixels to be processed. In optical flow algorithm, we used as input two images of a 640x480 pixel, and in greyscale filter algorithm we used an image of 2448x3264 which has 8 Mega pixels. Subsequently, processing an image of 8 Mega pixels will take more time than an image with 4 Megapixels.

Among the algorithms that we implemented in our project, there are four of them that take a long execution time on both Host PC and ZedBoard, which are Edge Detection, Image Smoothing, Optical Flow and Background Substractor. This is mainly due to their use of OpenCV functions that

take long time such as convolution, matrix calculation. For example, Optical Flow algorithm uses many nested loops in the code which are of complexity of O(N^2), where N depends on size of the input image (Number of pixels). Also, for Image Smoothing algorithm, the code uses functions like gaussian blur and median Blur functions. The gaussian blur algorithm is implemented using a convolution kernel and a weight kernel will give a different blur effect on the image. So this algorithm can be slow as its processing time depends on the size of the image and the size of the kernel.

Conclusions and Future Work

Conclusions

In this project, we implemented a number of Image Processing OpenCV algorithms on the Xilinx ZedBoard. The execution times of these algorithms were compared to the execution times on a host PC. The results clearly showed, expectedly, that the host PC outperformed the FPGA board. This is mainly because the applications were based on a pure OpenCV software implantation with no hardware acceleration. Hence, the huge difference of clock speed and memory size between the two platforms certainly explains our results.

The work in this project will set the foundations for the upcoming FPGA-based real- time image/video processing projects based on the xfOpenCV library with hardware acceleration. We explored the capabilities of the ZedBoard FPGA board in terms of image processing, and we can say that it is a suited platform for such a task. However, the execution time is still the main limiting factor of such platform.

We believe that the project was successful and it achieved its main goals. However, a lot of work has to be done by the students and the researchers at our institute to expand it and use it to implement some advanced real-time image/video processing algorithms. This can also integrate a webcam, as a source of live video, that can be processed using the FPGA hardware. This allows the comparison between the real-time hardware capabilities of the FPGA and the software implementation.

Issues and suggestions

We faced many problems while working on this project due to many facts. The first one is the lack of references dedicated to hardware prototyping and development.

As our main goal was to execute Real Image/Video processing Algorithms, we faced a lack of additional components that we could use with the ZedBoard such as Camera module. In addition, the xfOpenCV library needs a paid license for Vivado 2017.4 and SoDC from Xilinx, which was not available to us at a time of the experimental work. Also, Xilinx only tested their xfOpenCV library on the ZCU201 board which is different and much more advanced than the ZedBoard that we had.

As expected, one academic term is not enough to make a real-time embedded image/video-processing framework. However, our work provided a foundation for the other project to come in the future.

As final suggestions, the following points need to be considered:

- Work with Accelerated version of OpenCV which is xfOpenCV as outlined in section 1.3.
- Implementing others algorithms like face detection using the available modules.
- Using a better camera, a one dedicated for hardware prototyping and development for computer vision applications such as OV7670 camera module.
- Using another Board which is Zybo Z7-10: Zynq-7000 ARM/FPGA SoC Development Board.

Future Work

In this section, we suggest several recommendations to further the research on the challenges addressed in this work.

Parallelism on ARM

As we mentioned before, Zynq-7000 series use a dual ARM Cortex-A9 as CPU and two Neon coprocessor. In this work, we only used one ARM core for our implementation. One recommendation is to exploit this parallelism by executing OpenCV application on multiple cores and co-processor which will further improve the performance issues of real-time computer vision applications. This certainly includes the analysis of each algorithm and exploring the possibility for parallelism that can be exploited.

Evaluation of power and energy

Obtaining the speed up and reducing the resource usage are significant challenges to improve the system. Another major problem is the power consumption. To achieve an efficient system, it is a good idea to evaluate the power effects in order to improve energy efficiency of the system.

Bibliography

[1] Xilinx OpenCV User Guide UG1233 (v2017.4) [online] viewed May 18, 2018. Available: www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1233-xilinx-opencv-user guide.pdf

[2] OpenCV 2.4.13.6 documentations “OpenCV tutorials” imgproc module. Image Processor Smoothing images (undated) [Online]. Viewed 2018 April 5. Available: https://docs.opencv.org/2.4/doc/tutorials/imgproc/gausian_median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html

[3] Zynq-7000 All Programmable SoC: Embedded Design Tutorial, A Hands-On Guide to Effective Embedded System Design, UG165(v2016.3) February 13, 2018

[4] OpenCV 2.4.13.6-dev documentation >> OpenCV Tutorials >>video module. Background Available:https://docs.opencv.org/2.0beta/doc/tutorials/video/background_subtraction/background_subtraction.htm

[5] Digilent Documentation Getting Started with Vivado (undated) [Online] viewed 2018 April

[6] Creating a custom IP block in Vivado Using ZedBoard: A Tutorial Embedded Processor Hardware Design, February 24th 2015.

[7] The OpenCV Tutorials, Release 2.4.13.6. [online] viewed May 21, 2018 Available:https://docs.opencv.org/2.4/doc/tutorials/tutorials.html

[8] Hindborg, Andreas Erik; Schleuniger, Pascal, Jensen, Nicklas Bo, Karlsson, sven: Hardware Realization of an FPGA Processor – Operating System Call Offload and Experiences Published

in: Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing (DASIP)

[9] P. Forss'en, D. Lowe, S-H Wang, 'MSER (Maximally Stable External Regions)', [online] viewed 2018 May 6.
Available :http://www.micc.unifi.it/delbimbo/wpcontent/uploads/2011/03/slide_corso/A34%20MSER.pdf

[10] Vivado Design Suite Tutorial High-Level Synthesis UG871 (new release for Vivado design suites 2014.2) May 30, 2018

[11] J. Noguera F. M. Vallina Daniele Bagni, A. Di Fresco. PS and PL Partitions in the Zynq-7000 APSoc.
Available: http://www.xilinx.com/support/documentation/ application_notes/xapp1170-zynq hls.pdf#G1204263.

[12] Xilinx. Znyq-7000 all programmable SoC. [online] http://www.xilinx.com/products/ silicon-devices/soc/zynq-7000.html, July 2014.

[13] The official Linux kernel from Xilinx.
Available: https://git: //github.com/Xilinx/linux-xlnx.git

[14] Xilinx xfOpenCV Library. [online] Available: https://github.com/Xilinx/xfopencv

[15] OpenCV documentations. https://docs.opencv.org/2.4.9/index.html

[16] Dreamland Fantasy Studio, "Image Processing Algorithms" [online] Viewed 2018 May 22
Available: http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/